Femto-Containers: Lightweight Virtualization and Fault Isolation For Small Software Functions on Low-Power IoT Microcontrollers

Koen Zandberg
Inria
France
koen.zandberg@inria.fr

Emmanuel Baccelli
Inria
France
Freie Universität Berlin
Germany
emmanuel.baccelli@inria.fr

Shenghao Yuan
Inria
France
shenghao.yuan@inria.fr

Frédéric Besson
Inria
France
frederic.besson@inria.fr

Jean-Pierre Talpin
Inria
France
jean-pierre.talpin@inria.fr

ABSTRACT

Low-power operating system runtimes used on IoT microcontrollers typically provide rudimentary APIs, basic connectivity and, sometimes, a (secure) firmware update mechanism. In contrast, on less constrained hardware, networked software has entered the age of serverless, microservices and agility. With a view to bridge this gap, in the paper we design Femto-Containers, a new middleware runtime which can be embedded on heterogeneous low-power IoT devices. Femto-Containers enable the secure deployment, execution and isolation of small virtual software functions on low-power IoT devices, over the network. We implement Femto-Containers, and provide integration in RIOT, a popular open source IoT operating system. We then evaluate the performance of our implementation, which was formally verified for fault-isolation, guaranteeing that RIOT is shielded from logic loaded and executed in a Femto-Container. Our experiments on various popular microcontroller architectures (Arm Cortex-M, ESP32 and RISC-V) show that Femto-Containers offer an attractive trade-off in terms of memory footprint overhead, energy consumption, and security.

CCS CONCEPTS

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

KEYWORDS

IoT, Low-Power, Microcontroller, Middleware, Container, Virtual Machine, Function-as-a-Service, Security

ACM Reference Format:
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 *23rd ACM/IFIP International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3528535.3565242

Middleware '22, November 7–11, 2022, Quebec, QC, Canada

ACM ISBN 978-1-4503-9340-9/22/11…$15.00
https://doi.org/10.1145/3528535.3565242

1 INTRODUCTION

An estimated 250 billion microcontrollers are in use today [18]. An increasing percentage of these microcontrollers are networked and take part in distributed cyber-physical systems and the Internet of Things (IoT) we increasingly depend upon. For example, such low-power microcontrollers are at the core of hundreds of millions of connected machines such as sensors and actuators relied upon not only in smart homes, but also in other networked areas of the IoT and industrial contexts (Industry 4.0). On such hardware, the total memory available (for the whole system) is in the order of tens or hundreds of kBytes, without virtual memory management (MMU), often also without hardware memory protection (MPU). Neither Linux (or derivatives/equivalents) nor traditional hypervisor can be used as software platform on such hardware, and in effect the challenge of deploying and maintaining distributed low-power IoT software is exacerbated.

Recently, with the wider availability of low-power operating systems alternatives [15] and adequate network stacks, 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 workflow from the 1990s than today's common software practices. Simplistic application programming interfaces (APIs) offer basic performance and connectivity, but no additional comfort.

However, since the 1990s, networked software was revolutionized many times over. Networked software has entered the age of server-less, micro-services and agility. In the field, software modules that are deployed and running are expected to be quickly updatable in terms of functionalities, and in terms of bug fixes. Additional layers and primitives providing cybersecurity, flexibility and scalability became crucial: virtual machines, script programming (*e.g.* Python, Javascript), lightweight software containerization (*e.g.*

Docker, Function-as-a-Service [12] etc.). DevOps [7] workflow drastically shortened software development/deployment life cycles to provide continuous delivery of higher software quality.

In such a context, low-power IoT devices based on microcontrollers are the new 'weakest link' within distributed cyber-physical systems. Indeed, state-of-the-art primitives for 'serverless', including lightweight virtual machine (VM) runtimes [29] and container runtimes (*e.g.* Docker) are not applicable on low-power devices: they typically depend on operating systems and larger resources which don't fit microcontrollers. In particular, VMs are either too prohibitive in terms of hosting engine memory resource requirements (*e.g.* standard Java virtual machines), or restricted to very specific use cases (*e.g.* JavaCard). This lackluster creates bottlenecks that severely impact both flexibility and cybersecurity in the low-power IoT space.

Goals of this paper – We aim to design a middleware function runtime adequate for heterogeneous microcontrollers. Mainly enabling the secure deployment, execution and isolation of small virtual software functions on low-power IoT devices. What we aim for in priority is small start-up time for deployed functions, and negligible overhead w.r.t. memory footprint on the microcontroller when adding a hosted function runtime to the OS, compared to the same functionality implemented natively (in the OS).

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

- We propose Femto-Containers, a novel middleware for the abstraction, secure deployment, execution and isolation of (multiple, concurrent) software functions on heterogeneous microcontroller-based IoT devices. We design Femto-Containers as an extension of rBPF virtual machines;
- We benchmark ultra-lightweight virtualization techniques based on Python, WebAssembly, JavaScript and eBPF. We show that, comparatively, a Femto-Container runtime based on eBPF virtualization requires 10x less memory footprint;
- We provide an open source implementation of Femto-Containers, which we integrate in practice on a common, general-purpose operating system for low-power networked microcontrollers (RIOT);
- We formally verify key components of our Femto-Container implementation, guaranteeing fault-isolation amongst concurrent Femto-Containers, and the underlying OS;
- We evaluate the performance of Femto-Containers in a variety of use cases, on the most popular 32-bit microcontroller architectures: Arm Cortex-M, ESP32 and RISC-V.

2 MULTI-TENANT SOFTWARE SCENARIOS ON MICROCONTROLLERS

Software on low-power IoT devices is growing complexity, driven by cybersecurity, interoperability, and device management requirements. In practice, the development of embedded software components is therefore often delegated to distinct entities. For security and privacy reasons these distinct entities have limited mutual trust [36]. For example, in this context, prior work such as Amulet [16] aim to isolate multiple applications from each other on a microcontroller, and to protect the underlying OS from application code. Furthermore, maintenance of these low-power IoT devices typically requires on-the-fly instrumentation. Safety requires that hot insertion of instrumentation code cannot break running software (already deployed). For example, prior work such as TockOS [26] aims to enable safe multiprogramming of low-power microcontrollers.

In this context, we identify the following categories of use-cases, depicted in Figure 1:

(1) Use-case 1: Hosting and isolation of a high-level business function. This function can be updated securely, on-demand, remotely over the low-power network. The execution of this type of logic is typically periodic in nature, and has loose (non-real-time) timing requirements.
(2) Use-case 2: Hosting and isolation of debug and monitoring code functions at low-level. These are inserted and removed on-demand over the network. The functions must not interfere with existing code on the device. Comparatively, this type of function is short-lived and exhibits stricter timing requirements.
(3) Use-case 3: Hosting and isolation of several functions, managed by several different tenants.

Users in the above scenarios are provided with an event-driven programming model and fine-grained computational space, hosted on-demand on fleets of designated low-power IoT devices. Furthermore, the API available for a function fundamentally abstracts away most of the hardware and the OS. Conceptually, these scenarios thus partly mimic a Function-as-a-Service (FaaS) programming model [12].

As with FaaS, multiple functions provided by distinct stakeholders must run on a single device. For example OEM firmware may have to be completed/customised by separate components, e.g. different developers/tenants may provide drivers separately, which should be fault-isolated and restricted to using only driver-relevant resources. Meanwhile, OS maintainers can deploy/run debug snippets inserted elsewhere in the embedded code.

However, as a stable high-throughput network connection cannot be assumed, storage capability is restricted to device-local storage. Furthermore, scalability aspects of FaaS and container techniques (e.g. running thousands of containers on a single machine) do not play a significant role here. Instead, the scenarios we identify above require running just a handful of isolated functions on top of the embedded OS. However, considering potentially large fleets of IoT devices, the scenario may nevertheless involve a large number of containers (but across a large number of devices).

In this paper, we focus primarily on the middleware embedded in the devices: the runtime permitting to host, run and isolate functions, on-demand, on heterogeneous low-power IoT devices deployed in the field, based on popular 32-bit microcontroller architectures (such as the ARM Cortex-M class, RISC-V or ESP32).

3 THREAT MODEL

When a client deploys functions on a device operational in the field, the embedded environment has to ensure these functions are sandboxed. In our threat model, we consider both malicious tenants which can deploy malicious code and malicious clients which can maliciously interact with deployed code [34].

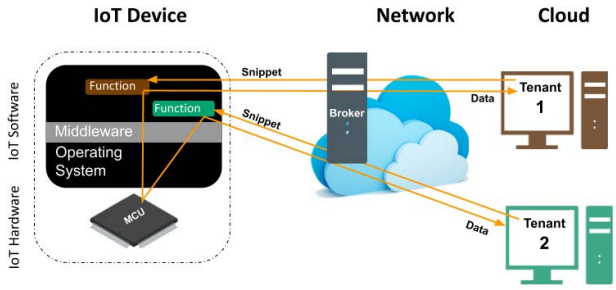

Figure 1: Container runtime use on IoT microcontrollers.

Malicious Tenant: The malicious tenant seeks to gain elevated permissions on the device it has already a set of permissions on. This tenant is already allowed to run code in the sandboxed environment, and the tenant might want to break free from the sandbox to either the host system or a different sandbox it doesn't have permissions for. While a tenant has to work within the permissions granted by the host service, it can make free use of the granted resources.

Malicious Client: The malicious client doesn't have any permissions for running sandboxed code on the device. The only access the malicious client has is access to networked endpoints exposed by the device, e.g. CoAP endpoints exposed by existing sandboxed environments. The malicious client seeks to gain any permission on the device to influence it or gain access to confidential data on the device. The malicious client could make use of an already vulnerable tenant function.

A number of attack vectors are considered in this work:

- *Install and update time attacks*: These attacks focus on modifying the application during the transport to the sandbox environment. This includes man-in-the-middle modifications to the applications.
- *Privilege escalation to a different sandbox*: This class of attacks focus on escaping the sandbox of the application to a different sandbox. The new sandbox could have different permissions.
- *Privilege escalation to the operating system*: This attack class attempts to escape the sandboxed environment altogether to the operating system.
- *Resource exhaustion attacks*: The devices considered here have very limited resources, both computational power and battery energy are limited. A denial of service vector can be to exhaust these resources.

Within the Femto-Container design we consider network attacks to exhaust resources on the system to be out of scope, this type of attack should be guarded against by the embedded operating system.

4 RELATED WORK

Typically, the fundamental building block for middleware embedded on devices allowing for generic function deployment and execution is a *virtual machine runtime*.

The vast majority of prior work on lightweight virtualization runtimes [29] does not target microcontrollers, but microprocessor-class computers. Recent examples include for instance AWS Firecracker [2] for serverless computing, WebAssembly [14] for process isolation in Web browsers, or eBPF [11, 28] for debug and inspection code inserted in the Linux kernel at run-time.

However, some ultra-lightweight virtualization approaches have been proposed for microcontrollers. For example, minimized WebAssembly runtimes adapted to run on 32-bit microcontrollers were proposed, such as WAMR [10] and WASM3 [35]. RapidPatch [17] uses an eBPF runtime to provide a hotpatching framework for RTOS firmwares.

VM runtimes for microcontrollers include also earlier examples such as Mate [25] or Darjeeling [9], a subset of the Java VM, modified to use a 16 bit architecture, designed for 8- and 16-bit microcontrollers. JavaCard [31] also uses a small Java virtual machine tailored for cryptographic purposes, running on smart cards.

Recently, tiny scripted logic interpreters and runtimes have also been proposed to provide a basic virtualization environment. For instance, MicroPython [27] is a very popular scripted logic interpreter used on microcontrollers. Small Python runtimes are used on ESP8266 microcontrollers in prior work such as NanoLambda [13]. Small Javascript runtimes are used on Cortex-M microcontrollers in prior work such as RIOTjs [4]. However, complementary mechanisms should however be used to guarantee mutual isolation between scripts (such as SecureJS [20]).

The most closely related work was published in [39] and in [37]. In [39], authors provide rBPF, a port of the eBPF instruction set in order to host a (single) VM on a microcontroller. In contrast, we extend rBPF's instruction set architecture and VM core with an adequate embedded loading and execution environment, which caters for well-defined, event-driven, short lived and isolated (concurrent) execution of (multiple) functions to be deployed on-the-fly on a networked microcontroller. On the other hand, while [37], concerns formal verification of the sole rBPF instruction interpreter, here we also verify the pre-flight instruction checker and we integrate both in our femto-container implementation – the performance of which we thoroughly evaluate on various microcontrollers. Our design and implementation are so small (a few hundreds lines of code) that formal verification was indeed realistic. Comparatively, software alternatives (WASM, MicroPython...) would require hundreds of thousands of lines of code, and magnitudes more lines of proof, all but voiding concrete perspectives of formal verification. Hardware alternatives (e.g. TrustZone [32]) are, to the best of our knowledge, not formally verified, and may require a software API to avoid unspecified hardware behaviours, hence faults.

To the best of our knowledge, our work provides the first formally verified middleware based on eBPF virtualization able to host multiple tiny runtime containers on a wide variety of heterogeneous low-power microcontrollers.

5 EMBEDDED RUNTIME ARCHITECTURE DESIGN

In this section, we introduce Femto-Containers, a new embedded runtime architecture tailored for constrained IoT devices, as described in the following.

Similarly to a FaaS runtime, Femto-Containers allow for the we deployment and execution of small logic modules. These modules, or functions, are hosted on top of a middleware offering isolation, abstraction and tight isolation with respect to the underlying OS and hardware. By combining isolation and hardware/OS abstraction, we retain the crucial properties of FaaS runtimes: code mobility and cyber-security. Differently from typical FaaS runtimes, however, Femto-Containers must be able to interact with specific hardware (*e.g.* sensor/actuators), and must drastically reduce the scope and the cost of virtualization to make do with IoT hardware constraints.

The Femto-Container architecture therefore relies on ultra-light-weight virtualization, as well as on a set of assumptions and features regarding an underlying RTOS, defined below.

Use of an RTOS with Multi-Threading. It is assumed that the RTOS supports real-time multi-threading with a scheduler. Each Femto-Container runs in a separate thread. Well-known operating systems in this space can provide for that, such as RIOT [5] or FreeRTOS [6] and others [15]. These can run on the bulk of commodity microcontroller hardware available. Note that RTOS facilities for scheduling enable simple controlling of how Femto-Containers interfere with other tasks in the embedded system.

No Assumptions on Microcontroller Hardware. To retain generality, we aim for a purely software-based isolation, which can also run on the least capable microcontrollers, without any assumptions on hardware architecture enhancements or security peripherals. If present, hardware-based isolation features could nevertheless be used to add layers of protection in-depth. For instance TrustZone software module isolation relies on enhanced Arm Cortex-M microcontroller architectures [32]. Other examples using hardware based protection mechanisms are TockOS [26] or Amulet [16], which rely on a hardware Memory Protection Unit (MPU) to isolate software modules.

Use of Ultra-Lightweight Virtualization. The virtual machine provides hardware agnosticism, and should therefore not rely on any specific hardware features or peripherals. This allows for running identical application code on heterogeneous hardware platforms. The virtual machine instances must have a low memory footprint, both in Flash and in RAM. This allows to run multiple VMs in parallel on the device. Note that, since we aim to virtualize less functionalities, the VM can in fact implement a reduced feature set. For instance, virtualized peripherals such as an interrupt controller are not required, and we give up the possibility of virtualizing a full OS. As exact virtualized hardware and peripherals is not a requirement, solutions around the interpretation of a scripting language are also suitable as target for Femto-Containers.

Use of OS Interfaces. A slim environment around the virtual machine (VM) exposes RTOS facilities to the VM. The container sandboxing a VM allows this VM to be independent of the underlying operating system, and provides the facilities as a generic interface to the VM. Simple contracts between container and RTOS can be used to define and limit the privileges of a container regarding its access to OS facilities. Note that such limitations must be enforced at run-time to safely allow 3rd party module reprogramming.

Isolation & Sandboxing through Virtualization. The OS and Femto-Containers must be mutually protected from malicious code, as described in section 3. This implies in particular that code running in the VM must not be able to access memory regions outside of what is granted via permissions. Here again, simple contracts can be used to define and limit memory and peripheral access of the code running in the Femto-Container. The strong isolation and security of the sandbox must prevent tenant escalation to the operating system and other tenants.

Slim Event-based Launchpad Execution Model. Femto-Containers are executed on-demand, when an event in the RTOS context calls for it. Femto-Container applications are rather short-lived and have a finite execution constraint. This execution model fits well with the characteristics of most low-power IoT software. To simplify containerization and enforce security-by-design, we mandate that Femto-Containers can only be attached and launch from predetermined launch pads, which are sprinkled throughout the RTOS firmware. Where applicable however, the result from the Femto-Container execution can modify the control flow in the firmware as defined in the launch pad.

Low-power Secure Runtime Update Primitives. Launching a new Femto-Container or modifying an existing Femto-Container can be done without modifying the RTOS firmware. However, updating the hooks themselves requires a firmware update. In our implementation, both types of updates use CoAP network transfer and software update metadata defined by SUIT [30] (CBOR, COSE) to secure updates end-to-end over network paths including low-power wireless segments [40]. Leveraging SUIT for these update payloads provides authentication, integrity checks and rollback options. Updating a Femto-Container application attached to a hook is done via a SUIT manifest. The exact hook to attach the new Femto-Container to is done by specifying the hook as a unique identifier (UUID) as a storage location in the SUIT manifest. A rapid develop-and-deploy cycle only requires a new SUIT manifest with the storage location specified every update. Sending this manifest to the device triggers the update of the hook after the new Femto-Container application is downloaded to the device and stored in the RAM. Using a secure update mechanism such as SUIT prevents a malicious client from intervening in the update and installation process of a Femto-Container application.

6 ULTRA-LIGHTWEIGHT VM MICRO-BENCHMARKS

In this section, we compare the performance of a proof of concept using RIOT [5] to extend with Femto-Container functionality, based on different ultra-lightweight isolation techniques: Python (MicroPython runtime), WebAssembly (WASM3 runtime), eBPF (rBPF runtime) and Javascript (RIOTjs runtime).

We run experiments using each virtualization candidate on an off-the-shelf IoT hardware platform representative of the landscape of modern 32-bit microcontroller architectures available: Arm Cortex-M4. Details of the benchmark setup are in Appendix A.

In the benchmarks we report on below, each implementation is loaded with a VM hosting logic performing a Fletcher32 checksum on a 360 B input string. We reason that this computing load roughly

mimics the instruction complexity of intensive sensor data (pre-)processing on-board.

Runtime	ROM size	RAM size
WASM3	64 KiB	85 KiB
rBPF	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 Femto-Container runtimes.

Runtime	code size	cold start overhead	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 Femto-Container runtimes on Cortex-M4.

Our benchmarks results are shown in Table 1 and Table 2. The startup time measures the time it takes for the runtime to load the application. This contains setup processing to parse the application or JIT compilation steps.

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 [39] the minimum required page size of 64 KiB to comply with the WebAssembly specification explains why WASM3 performs poorly in terms of RAM. One can envision enhancements where this requirement is relaxed. However the RAM budget would still be well above an order of magnitude more than the RAM budget we measured with 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 2.

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.

Considering VM architecture & security. 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 with mututal isolation of the VMs' memory between different tenants.

Furthermore, security guarantees call for a formally verified implementation of the hosting engine down the road. A typical approximation is: less lines of code (LoC) means much less effort to 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.

6.1 Choice of Virtualization

Our benchmarks indicate that in terms of memory overhead, startup time and LoC, Femto-Containers using eBPF virtualization is the most attractive, by far. We note that execution time with WebAssembly is 2x faster than with rBPF. However, we expect that a 2x factor in execution time will have no significant impact in practice for the use cases we target, such as small processing workloads. Since our priority is on memory footprint (recall our aim of ≈10% memory overhead for functionality containerization) we thus choose rBPF to flesh out our concept further.

7 FEMTO-CONTAINER RUNTIME IMPLEMENTATION

As proof of concept, we implemented the Femto-Container architecture, with functions hosted in the operating system RIOT and virtualization using an instruction set compatible with the eBPF instruction set. This implementation is open source (published in [22]). We detail below its main characteristics.

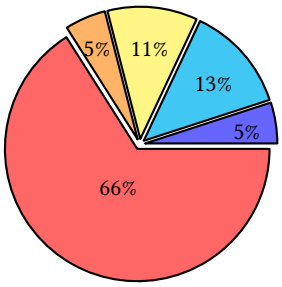

fig1: RIOT with MicroPython Femto-Container (154kBytes).

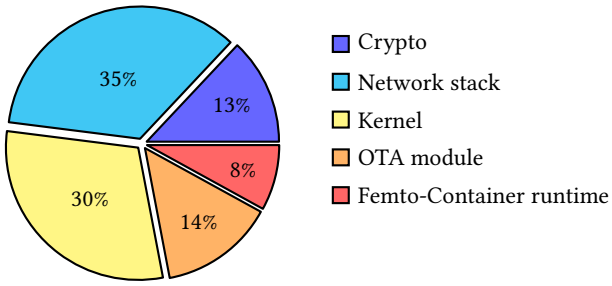

fig2: RIOT with rBPF Femto-Container (57kBytes).

Figure 2: Flash memory distribution with different Femto-Containers. RIOT is configured with 6LoWPAN, CoAP, SUIT-compliant OTA (totalling 53kBytes in Flash memory).

Simple Containerization. Simplified interfaces provide a uniform environment around the VM, independent of the RIOT operating system. Access from the Femto-Container to the required OS facilities is allowed through system calls to services provided by RIOT. These system calls can be used by the loaded applications via the eBPF native `call` instruction. Furthermore, the OS can share specific memory regions with the container.

Key-value store. In lieu of a file system, applications hosted in Femto-Containers can load and store simple values, by a numerical key reference, in a key-value 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. By default, two key-value stores are provided by the OS. The first key-value store is local to the application, for values that are private to the VM accommodated in the container. 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 facilitate sharing data across a set of VMs from the same tenant, while isolating this set of VMs from other tenants' VMs.

Use of RIOT Multi-Threading. Each Femto-Container application instance running is scheduled as a regular thread in RIOT. The native OS thread scheduling mechanism can simply execute concurrently and share resources amongst multiple Femto-Containers and other tasks, spread over different threads. A Femto-Containers instance requires minimal RAM: a small stack and the register set, but no heap. The host RTOS bears thus a very small overhead per Femto-Containers instance.

Figure 3 shows how Femto-Containers integrate into the operating system.An overview of how Femto-Containers integrate in the operating system is shown in Figure 3. It shows the flow within the operating system, with the Femto-Container triggered by an event in the operating system. The Femto-Container is started if it is present and gains access to it's own store and any bindings allowed by the operating system.

As the Femto-Container Instance does not virtualize its own set of peripherals, no interrupts or pseudo-hardware is available to the Femto-Container application. This also removes the option to interrupt the application flow inside a Femto-Container.

The hardware and peripherals available on the device are not accessible by the Femto-Containers instances. All interaction with hardware peripherals passes through the host RTOS via the system call interface.

Ultra-Lightweight Virtualisation using eBPF. Application code is virtualized using Femto-Containers, our enhancement of the rBPF virtual machine implementation. rBPF is again based on the Linux eBPF. The architectures of these virtual machines are similar enough that they all use the LLVM compiler with the eBPF target for compilation.

Register-based VM. The virtual machine operates on eleven registers of 64 bits wide. The last register (`r10`) is a read-only pointer to the beginning of a 512 B stack provided by the femto-container hosting engine. Interaction with the stack happens via load and store instructions. 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. Position-independent code is achieved by using the reference in `r10` and the offset field in the instructions.

Jumptable & Interpreter. The interpreter parses instructions and executes them operating on the registers and stack. The machine itself is implemented as a computed jumptable, 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.

Isolation & Sandboxing

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. Allow lists can be configured (attached in the hosting engine) to explicitly allow a VM instance access to other memory regions. These memory regions can have individual flags for allowing read/write access. For example, a firewall-type trigger can grant read-only access to the network packet, allowing the virtual machine to inspect the packet, but not to modify it. As the memory instructions allow for calculated addresses based on register values, memory accesses are

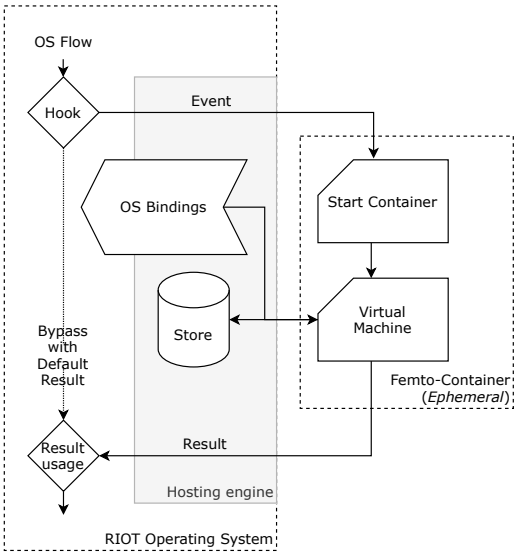

Figure 3: Femto-Container RTOS integration.

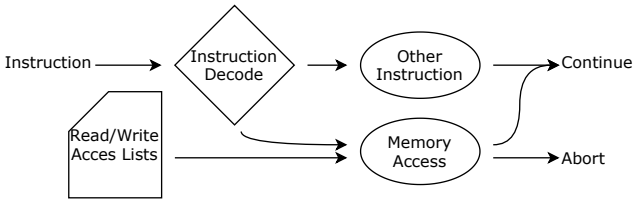

Figure 4: Interaction between memory instructions and the access lists.

checked at runtime with the access lists against the resulting address, as show in Figure 4. Illegal access aborts execution. While this relies on properly configured allow lists, it prevents access to memory outside the sandbox by a malicious tenant.

Pre-flight instruction checks. A Femto-Container verifies the application before it is executed for the first time. These checks include checks on the individual 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 are also checked 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. This prevents a tenant from jumping execution to the application code outside of the sandbox such as code from a different tenant or vulnerable code planted otherwise.

Finite execution is also enforced, by limiting both the total number of instructions N_i, and the number of branch instructions N_b that are allowed. In practice, this limits the total number of instructions executed to: $N_i \times N_b$. This puts a limit on the computational resource exhausted by a single execution by putting a hard limit on the total number of instructions executed.

Hooks & Event-based Execution

The Femto-Container hosting engine instantiates and runs containers as triggered by events within the RTOS. Such events can be a network packet reception, sensor reading input or an operating system scheduling events for instance. Business logic applications can be implemented either by directly responding to sensor input or by attaching to a timer-based hook to fire periodically.

Simple hooks are pre-compiled into the RTOS firmware, providing a pre-determined set of pads from which Femto-Containers can be attached and launched.

Listing 1: Example hook implementation.

```
sched_ctx_t context = {
    .previous = active_thread,
    .next = next_thread,
};

int64_t result;

f12r_hook_execute(FC_HOOK_SCHED, &context,
                  sizeof(context), &result);
```

An example of a hook integrated in the firmware is shown in Listing 1. The firmware has to set up the context struct for the Femto-Containers after which it can call the hosting engine to execute the Femto-Container instances associated with the hook.

8 USE-CASE PROTOTYPING WITH FEMTO-CONTAINERS

In this section, we describe and demonstrate the programming model exposed by Femto-Containers. We use Femto-Containers to prototype the implementation of several use cases involving one or more functions (applications). Where multiple functions are involved, these are hosted concurrently on a single microcontroller. The goal of these functions is to match the scenarios we targeted initially in section 2.

In the prototype implementation we show below, we used C to code the applications hosted on Femto-Containers engine. However, any other target language supported by LLVM could be used instead such as C++ and Rust, for instance.

8.1 Programming Model

Femto-Containers follow an event-driven programming model. Applications hosted are only executed when triggered by events in the operating system. The applications specify the entry point and to which hooks they are to be attached inside the operating system.

The logic that can be deployed in Femto-Containers is limited following the eBPF architecture. For instance, asynchronous operation is not supported: there is no option to interrupt the control flow inside a Femto-Container application from outside the virtual machine. This is mainly caused by the simple nature of the architecture: neither interrupts nor indirect jumps are available. This trade-off reduces flexibility but increases the security for the host operating system.

Our current implementation is also limited by the fixed, small size of the stack (512 Bytes) dictated by the eBPF specification. More memory-consuming tasks would need special handling to provide additional memory. An enhanced implementation could however allow the application to request more stack from the RTOS, for example via the contracts, which would alleviate this issue. More computation- and memory-intensive tasks could also make use of additional system calls provided by the RTOS, which could execute generic primitives at native speed.

The Femto-Container hosting engine is designed to be fast enough to start applications on a hot code path without affecting significantly normal operating system execution times. Small applications can thus be inserted and execute without substantial overhead or slowdown for the RTOS. This way small debug or inspection applications can be inserted into the RTOS at any point as long as the application does not cause a deadline to be exceeded in the RTOS.

8.2 Kernel Debug Code Example

The first prototype we display consists in a single application, which intervenes on a hot code path: it is invoked by the scheduler of the OS. It keeps an updated count of each threads' activations. The logic hosted in the Femto-Container is shown in Listing 2. A small C 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, incremented every time the thread is scheduled. External code can request these counters and provide debug feedback to the developer.

Listing 2: Thread counter code.

```
#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 */
} sched_ctx_t;

int pid_log(sched_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;
}
```

8.3 Networked Sensor Code Example

The second prototype we display adds two Femto-Containers from another tenant to the setup of the first prototype. Interaction between these two additional containers is achieved via a separate key-value store, as depicted in Figure 5. The logic hosted in the first Femto-Container, periodically triggered by the timer event, reads, processes and stores a sensor value. The code for this logic is shown in [23]. The second container's logic is triggered upon receiving a network packet (CoAP request), and returns the stored sensor value back to the requestor. The code for this logic is shown in [21].

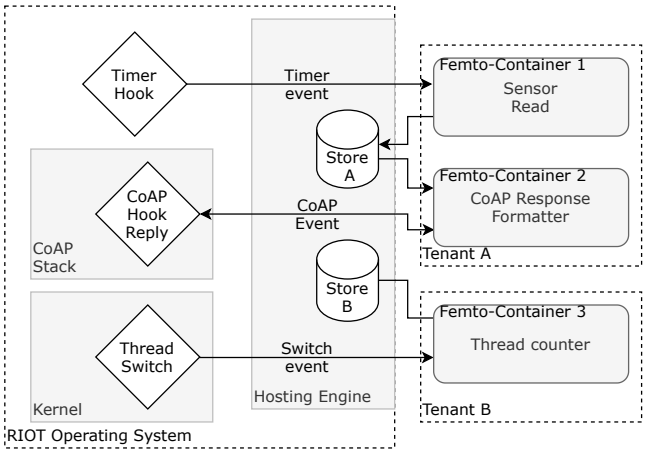

Figure 5: Event and value flow when hosting multiple containers from different tenants.

In this toy example, the sensor value processing is a simple moving average, but more complex post-processing is possible instead, such as differential privacy or some federated learning logic, for instance. This example sketches both how multiple tenants can be accommodated, and how separating the concerns between different containers is achieved (between sensor value reading/processing on the one hand, and on the other hand the communication between the device and a remote requester).

9 FEMTO-CONTAINER FORMAL VERIFICATION

The critical components of Femto-Containers in terms of cybersecurity are the rBPF interpreter and the pre-flight instruction checker. Since the implementation is conveniently small (500 lines of C code for the interpreter and the checker), we aimed at producing a formally verified implementation of these components. We will refer to CertFC (for Certified Femto-Container) as the runtime which uses the formally verified interpreter and checker.

Targeted requirements formalization. The security guarantees we wish to provide Femto-Containers with are essentially memory and fault isolation. More precisely, we want to prove it impossible for CertFC to access a memory location out of its app's register memory or to execute an instruction leading to an undefined behavior, and consequently heading the VM and/or its host to crash. Providing these guarantees further strengthens the security needed with the threat model to prevent access to memory outside of the sandbox, in turn preventing unprivileged access to the operating system or other virtual machines.

Formal verification approach. We have used the Coq proof assistant to mechanically and exhaustively verify these requirements by employing the design workflow depicted in Figure 6:

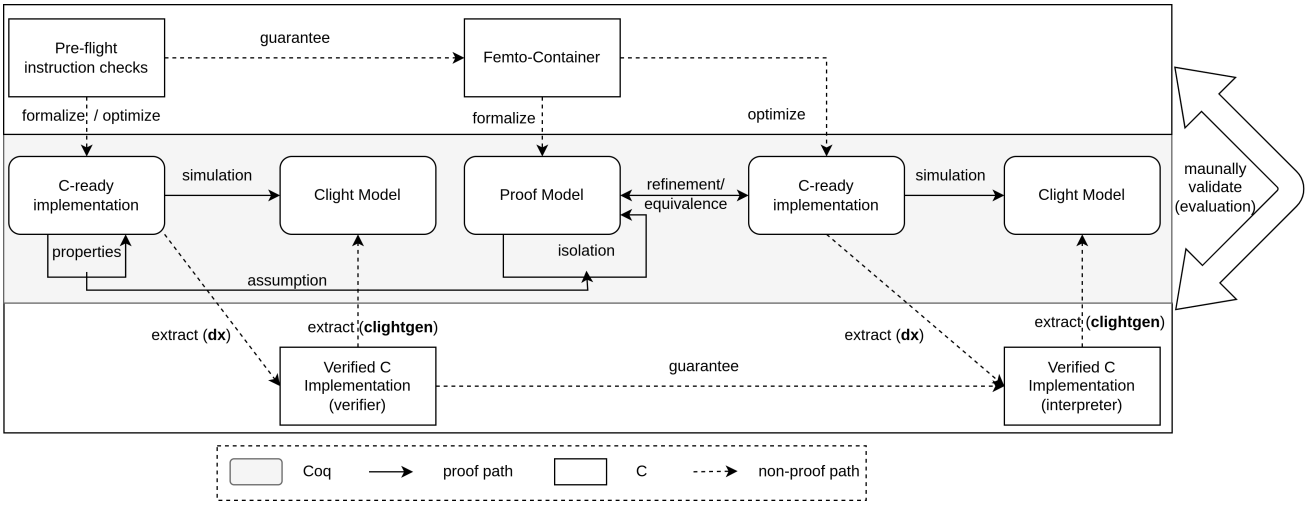

Figure 6: CertFC Formal verification workflow.

1) First, we provide a proof model and a C-ready implementation that formalize (resp. optimize) the native, vanilla, C implementations of the rBPF verifier and virtual machine in RIOT. Proof and "C-ready" models are proved semantically equivalent in Coq.
2) The verification of expected safety and isolation properties is performed by the Coq proof assistant on the VM's proof model. It relies on the formalized isolation guarantees provided by a) the CompCert C memory model [24] ii) the preflight runtime checks of the verifier, and iii) the defensive runtime checks of the virtual machine itself (for numerical and memory operations).
3) The verified C implementation is automatically extracted from the C-ready Coq model using the ∂x tool [19]. Based on a set of formalized translation rule from Coq to C, ∂x allows to craft a both reviewable and optimized C code from a functional Coq definition.
4) To ensure that the extracted C code refines the proof model, and hence satisfies the safety and isolation properties, the final simulation proof proceeds in two steps. First, a CompCert Clight model is extracted from the generated C code, using the VST-clightgen tool [3]. Second, proving that Clight model to simulate the C-ready model using translation validation [33].

Formal verification details. The Coq models and proofs used to obtain CertFC are presented in [37] and available from [38].

10 PERFORMANCE EVALUATION

In this section we evaluate and compare the performance of Femto-Containers with rBPF and CertFC runtimes. The comparison is done on a number of low-power IoT hardware platforms: Cortex-M4, RISC-V and ESP32 based microcontrollers.

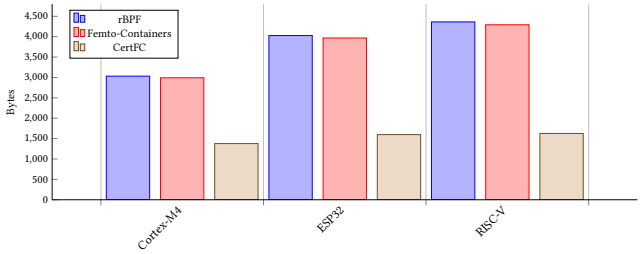

Figure 7: Flash requirement for the different implementations and platforms

10.1 Hosting Engine Analysis

We benchmark the Femto-Container implementation on a number of aspects. First, we compare the footprint of the hosting engine on the embedded device. This shows the impact of adding Femto-Containers to the applications Second, we compare the execution time of a number of individual instructions.

To compare the impact of adding the Femto-Containers to an existing firmware, we compare the memory footprint of the implementations. In general, each Femto-Container needs memory to

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

The impact on the required flash on the firmware is shown in Figure 7 and Table 3. In terms of required RAM for execution, both rBPF and Femto-Containers show comparable flash and RAM memory usage. In terms of Flash memory size, our measurements show that CertFC actually reduces the footprint by 55 % on Cortex-M4. The CertFC implementation requires slightly more memory, an increase of around 50 B per instance. This is caused by CertFC storing extra state of the virtual machine in the context struct and not on the thread stack.

The different implementations of Femto-Containers are compared in Figure 8 against a set of eBPF instructions, showing that the rBPF extensions incur minimal overhead on the virtual machine

	ROM size	RAM size
Femto-Containers	2992 B	624 B
rBPF	3032 B	620 B
CertFC	1378 B	672 B

Table 3: Memory footprint of a Femto-Container hosting minimal logic on Arm Cortex-M4.

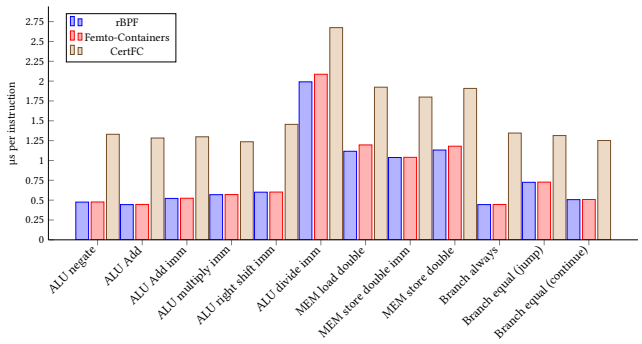

Figure 8: Time per instructions on the Cortex-M4 platform

and provide similar throughputs. Now, the performance of the formally verified CertFC is lagging behind the other implementations, revealing the trade off between the formally verified code and a natively optimized implementation.

10.2 Experiments with a Single Container

In this section we show the execution times of a number of Femto-Container applications. This shows the applicability of Femto-Containers in the different scenarios. we show the execution times in Figure 9.

The first example executes a Fletcher32 checksum over a data string of 360 B. It shows the time it takes for relative heavy processing within the Femto-Containers VM. Depending on the platform and the speed of the microcontroller it takes between 1.3 ms and 2.2 ms. For Femto-Containers the duration of this application is long.

The second example shown is the thread counter example previously shown in Listing 2. In normal operation it is inserted in the thread switch hook provided by the operating system, a hot path in the OS. As shown in the figure, adding this would increase the duration of a thread switch in the operating system by 10 µs to 27 µs. The impact on the operating system would not be negligible, but also not problematic during normal operation.

The last example shows the duration of the second stage of the networked sensor code example[21]. It depends heavily on system calls for formatting of the CoAP response, but still contains some processing inside the VM. It can be considered a representative example for business logic on the device. This example takes between 23 µs and 72 µs. For business logic programmed outside of the hot code path of the operating system itself, the overhead caused here by the VM is rather acceptable and doesn't impact the performance of the overall system.

10.3 Femto-Containers with Multiple Instances

Femto-Containers are optimized to run multiple containers on a single system in parallel. All state of an instance is kept local to the instance. Each new instance added takes 624 B of RAM to run, including the virtual machine stack. The other requirement is that the microcontroller must have a large enough storage for the all the application images.

We now measure the memory required to concurrently host multiple containers from multiple tenants on the same microcontroller, from the examples we described in section 8. As shown previously in Table 3, the minimal default memory footprint used by a Femto-Container amounts to 624 B, which is for storing the VM stack, housekeeping structs and information about memory regions. Furthermore, the key-value stores are also in RAM. In this case the total RAM used by the key value stores (and houskeeping) for different tenants was 340 B. Hence, the required RAM memory we measured so as to run the example with 3 containers and 2 tenants is 3.2 KiB. Beyond these examples, if we consider more containers hosting larger applications (*e.g.* ≈2000 B) an Arm Cortex-M4 microcontroller with 256 KiB RAM, the density of containers achievable would be of ≈100 instances, next to running the OS.

Different instances do not have access to each others resources by default. They are fully isolated and do not have access to each others memory, isolated by the memory protection mechanism. One way provided to communicate between the instances is the shared key-value store.

Multiple containers can be attached to the same launchpad hook inside the operating system. It depends on the hook how the return value from each instance is processed further. This allows for multiple tenants attaching to the same hook and process similar events.

10.4 Overhead Added by Hooks

One key question is how performance is affected by pre-provisioning launchpads (hooks) in the RTOS firmware. We measure in Table 4 the overhead caused by adding a hook to the RTOS workflow. This overhead amounts to ≈100 clock ticks on all the hardware we tested. Compared to the number of cycles needed for an average task in the operating system, this impact is low. Furthermore, this overhead is less than 10% of the number of cycles needed to execute the logic hosted in a Femto-Container. From this observation, we can conclude that, even if this hook is on a very hot code path (as for the Thread Counter example) the performance loss is tolerable. Conversely, the perspective of adding many hooks sprinkled in many places in the RTOS firmware is realistic without incurring significant performance loss.

	Empty Hook	Hook with Application
Cortex-M4	109	1750
ESP32	83	1163
RISC-V	106	754

Table 4: Hook overhead in clock ticks for the thread switch example

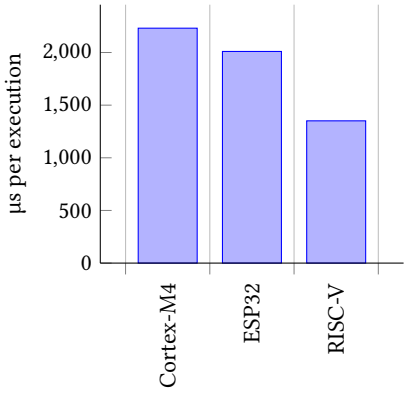

fig1: Fletcher32 checksumming algorithm application.

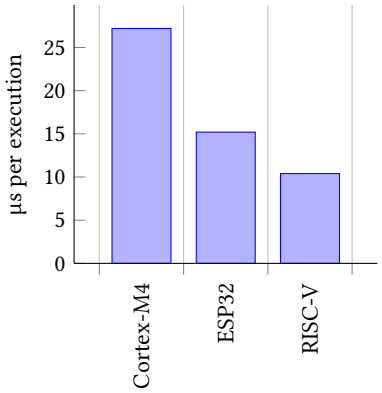

fig2: Thread log example application.

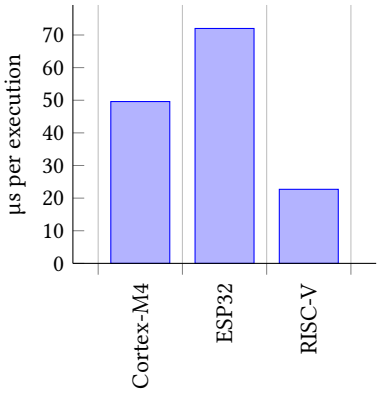

fig3: CoAP response formatter application.

Figure 9: Execution duration of different examples running on Femto-Containers.

11 DISCUSSION

Virtualization vs *Power-Efficiency.* Inherently, virtualization causes some execution overhead, due to interpretation of the code. Thus Femto-Containers increase power consumption for functionality executed within the VM, compared 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, power consumption and downtime are saved if software updates modify a Femto-Container instead of the full firmware.

Controlling Tenant Priviledges. Controlling and granting access to specific RTOS resources to different containers or tenants is a complex challenge. Our design includes a basic permission system based on preprovisionned hooks, system calls, and simple contracts between the hosting engine (on behalf of the OS) and a given container. Basically: the OS restricts the set of priviledges that can be granted, the container specifies the set of priviledges it requires, and the hosting engine grants the intersection of these sets. One limitation of our current simplified design is that there is only one fixed set of priviledges possible per hook. In case 2 tenants have different priviledges, a second hook must be made available. Additional mechanisms would be required to overcome this limiation and/or to enable dynamic priviledge levels.

Install Time vs *Execution Time.* As mentioned before, one limitation due to virtualization 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. To avoid the issues describe before on complicating the run-time security checks, this compilation into native code has to be done at run-time by the device deploying the code.

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.

12 CONCLUSION

In this paper we have introduced Femto-Containers, a new middleware runtime architecture we designed, which enables FaaS capabilities embedded on heterogeneous low-power IoT hardware. Using Femto-Containers, authorized (3rd party) maintainers of IoT software can deploy and manage via the network mutually isolated software modules embedded on a microcontroller-based device. We provided an open source implementation of the Femto-Container runtime, which uses the eBPF instruction set ported to microcontrollers, as well as integration in a common low-power IoT operating system (RIOT). We formally verified a fault-isolation guarantee which ensures that RIOT is shielded from arbitrary logic loaded and executed in a Femto-Container – and such, without requiring any specific hardware-based memory isolation mechanism. We then demonstrated experimentally the performance of the Femto-Container runtime on the most common 32-bit microcontroller architectures: Arm Cortex-M, RISC-V, ESP32. We show that Femto-Containers significantly improve state of the art, by providing FaaS-like capabilities with strong security guarantees on such microcontrollers, while requiring negligible Flash and RAM memory overhead (less than 10%) compared to native execution.

APPENDIX A: BENCHMARK CONFIGURATION

Hardware – We carry out our measurements on popular, commercial, off-the-shelf IoT hardware, representative of the landscape of the modern 32-bit microcontroller architectures that are available.

More precisely, we build and run the code on the following boards, all configured to run at 64 MHz:

- **Arm Cortex-M:** 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),
- **ESP32:** a WROOM-32 board, using an Espressif 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.
- **RISC-V:** a Sipeed Longan Nano GD32VF103CBT6 Development Board, which provides a RISC-V 32-bit microcontroller with 32 KiB RAM and 128 KiB Flash.

Note that an open-access testbed such as IoT-Lab [1] also provides some of this hardware, for reproducibility.

Software – In all benchmarks, the embedded software platform (OS) hosting the Femto-Containers is RIOT [5]. As base, we took RIOT Release 2022.01, configured to be IoT-ready. More precisely, we configured RIOT to provide standard low-power networking connectivity, leveraging the board's IEEE 802.15.4 radio chip and a ressource-efficient IPv6-compliant network stack (6LoWPAN, UDP, CoAP), as well as providing secure software update capability, enabling the update of Femto-Containers over the low-power network, in compliance with SUIT [8] specifications (using CBOR, COSE, ed25519 signatures and SHA256 hashes as primitives).

ACKNOWLEDGEMENTS

The research leading to these results partly received funding from the RIOT-fp project, and from the TinyPART project (within the MESRI-BMBF German-French cybersecurity program under grant agreements no ANR-20-CYAL-0005 and 16KIS1395K). The paper reflects only the authors' views. MESRI and BMBF are not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Cedric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele, et al. 2015. FIT IoT-LAB: A large scale open experimental IoT testbed. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, 459–464.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[3] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program logics for certified compilers.* Cambridge University Press.

[4] 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)*. IEEE, Athens, Greece, 504–507.

[5] 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.

[6] R. Barry. 2022. FreeRTOS, a FREE open source RTOS for small embedded real time systems. http://www.freertos.org.

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

[8] Henk Birkholz Brendan Moran, Hannes Tschofenig and Koen Zandberg. 2021. *CBOR-based Firmware Manifest Serialisation Format for the Software Updates for Internet of Things (SUIT) Manifest.* Internet-Draft draft-ietf-suit-manifest-16. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-suit-manifest-16 Work in Progress.

[9] Niels Brouwers et al. 2009. Darjeeling, a Feature-Rich VM for the Resource Poor. In *ACM SenSys*. Association for Computing Machinery, New York, NY, USA, 169–182. https://doi.org/10.1145/1644038.1644056

[10] Bytecode Alliance. 2020. WebAssembly Micro Runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime.

[11] Matt Fleming. 2017. *A Thorough Introduction to eBPF.* https://lwn.net/Articles/740157/

[12] Geoffrey C Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2017. Status of serverless computing and function-as-a-service (faas) in industry and research. (2017). https://doi.org/10.13140/RG.2.2.15007.87206 arXiv:arXiv:1708.08028

[13] Gareth George, Fatih Bakir, Rich Wolski, and Chandra Krintz. 2020. Nanolambda: Implementing functions as a service at all resource scales for the internet of things.. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, Virtual Event, Online, 220–231.

[14] Andreas Haas et al. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 185–200.

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

[16] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. 2018. Application Memory Isolation on Ultra-Low-Power MCUs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 127–132.

[17] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. 2022. RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2225–2242.

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

[19] Narjes Jomaa, Paolo Torrini, David Nowak, Gilles Grimaud, and Samuel Hym. 2018. Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base. In *18th International Workshop on Automated Verification of Critical Systems (AVOCS 2018)*, Vol. 76. Electronic Communications of the EASST Open Access Journal, Oxford, United Kingdom, 0–20.

[20] Yoonseok Ko, Tamara Rezk, and Manuel Serrano. 2021. SecureJS Compiler: Portable Memory Isolation in JavaScript. In *SAC 2021-The 36th ACM/SIGAPP Symposium On Applied Computing*. Association for Computing Machinery, New York, NY, USA, 1265–1274. https://doi.org/10.1145/3412841.3442001

[21] Koen Zandberg. 2022-05. Femto-Containers CoAP sensor value handler. https://anonymous.4open.science/r/middleware2022-femtocontainers-BB19/snippets/counter_fetch_gcoap.c.

[22] Koen Zandberg. 2022-05. Femto-Containers RIOT Implementation. https://github.com/future-proof-iot/middleware2022-femtocontainers/tree/main/femto-containers.

[23] Koen Zandberg. 2022-05. Femto-Containers sensor readout application. https://anonymous.4open.science/r/middleware2022-femtocontainers-BB19/snippets/sensor_process.c.

[24] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[25] Philip Alexander Levis and David E. Culler. 2002. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, California, USA, October 5-9, 2002*, Kourosh Gharachorloo and David A. Wood (Eds.). ACM Press, New York, NY, USA, 85–95.

[26] Amit Levy et al. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *ACM SOSP*. Association for Computing Machinery, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786

[27] George Robotics Limited. 2022. MicroPython. https://micropython.org/.

[28] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX*, Vol. 46. USENIX Association, San Diego, CA, 2.

[29] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. 2018. Consolidate IoT edge computing with lightweight virtualization. *IEEE network* 32, 1 (2018), 102–111.

[30] Brendan Moran, Milosch Meriac, Hannes Tschofenig, and David Brown. 2021. *A firmware update architecture for internet of things devices.* RFC 9019. RFC Editor. https://www.rfc-editor.org/rfc/rfc9019.txt

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

[32] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.
[33] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Bernhard Steffen (Eds.). Vol. 1384. Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166. https://doi.org/10.1007/BFb0054170 Series Title: Lecture Notes in Computer Science.
[34] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. 2020. Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds. In *USENIX*. USENIX Association, Boston, MA, USA, 927–942.
[35] Volodymyr Shymanskyy. 2020-10. WASM3: A high Performance WebAssembly Interpreter Written in C. https://github.com/wasm3/wasm3.
[36] Ian Thomas, Shinji Kikuchi, Emmanuel Baccelli, Kaspar Schleiser, Joerg Doerr, and Andreas Morgenstern. 2018. Design and Implementation of a Platform for Hyperconnected Cyber Physical Systems. *Internet of Things* 3 (2018), 69–81.
[37] Shenghao Yuan, Frédéric Besson, Jean-Pierre Talpin, Samuel Hym, Koen Zandberg, and Emmanuel Baccelli. 2022. End-to-End Mechanized Proof of an eBPF Virtual Machine for Micro-controllers. In *International Conference on Computer Aided Verification*. Springer, Haifa, Israel, 293–316.
[38] Shenghao Yuan, Frédéric Besson, Jean-Pierre Talpin, Samuel Hym, Koen Zandberg, and Emmanuel Baccelli. 2022-09. CertFC artifact. https://github.com/future-proof-iot/CertFC/tree/MIDDLEWARE22.
[39] Koen Zandberg and Emmanuel Baccelli. 2020. 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)*. IEEE, Berlin / Virtual, Germany, 1–6.
[40] 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.