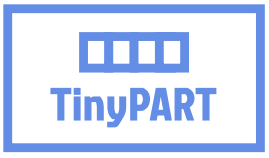

D2.1.1 : Prototype Ultra-low-power Script Microcontainer for Microcontrollers

---

Delivrable number: D2.1.1
Work package: WP2
Version: 1.0
Deliverable Lead Org: Inria
Deliverable type: R (Document, Report) | DEM (Demonstrator, pilot, prototype, plan designs) | OTHER (Software, technical diagram).
Dissemination Level: PU (Public)
Date: November 24, 2022

Authors:
Koen Zandberg (Inria), Samuel Legouix (Orange), Chrystel Gaber (Orange)

Reviewers:
David Nowak (University of Lille)

Abstract:
This deliverable presents multiple prototypes of embedded scripting environments and virtual machines geared towards ultra-low power microcontrollers, enabling secure execution and rapid prototyping of untrusted code on minimal hardware.

Keywords: Microcontainer, virtual machines

Acknowledgment The research leading to these results partly received funding from 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.
Disclaimer This report contains material which is the copyright of certain TinyPART Consortium Parties and may not be reproduced or copied without permission. All TinyPART Consortium Parties have agreed to publication of this report, the content of which is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. Neither the TinyPART Consortium Parties nor the ANR nor the VDE-VDI-IT warrant that the information contained in the Deliverable is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using the information.
CC BY-NC-ND 3.0 License - 2019-2021 TinyPART Consortium parties

Document Revision History

Document revision history			
Version	Date	Description of change	List of contributor(s)
v0.1	29/06/2022	Creation	Koen Zandberg
v0.2	20/09/2022	Add jerryscript and discussion	Samuel Legouix, Chrystel Gaber
v0.3	04/11/2022	Internal review	David Nowak
v1.0	07/11/2022	Corrections after steering committee review	Chrystel Gaber

Table 1: Document revision history

CC BY-NC-ND 3.0 License
TinyPART Consortium Parties

Contents

CC BY-NC-ND 3.0 License
TinyPART Consortium Parties

Introduction

Secure hosting of executable code is demonstrated in the prototypes of this package. We demonstrate multiple different execution environments, based on both virtual machines and script interpreters. Each environment must be generic and able to run on low power and minimal microcontroller-based hardware.

CC BY-NC-ND 3.0 License
TinyPART Consortium Parties

1 Execution environments

1.1 Femto-Containers

Femto-Containers is a variant of the eBPF virtual machine optimized for small microcontroller environments. Extended Berkeley Packet Filter (eBPF [1]) is a small in-kernel VM stemming from the Linux ecosystem, compatible with Unix-like operating systems.
eBPF is 64-bit register-based VM, using fixed-size 64bit instructions and a small ISA. eBPF uses a fixed-sized 512 byte stack and no heap, which limits VM memory overhead in RAM.
Femto-Containers is the latest port of the eBPF ISA.
The sandbox provided by Femto-Containers 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. Femto-Containers should not be vulnerable against hazardous memory access and code execution.

1.2 Micropython

MicroPython [2] is a Python3 execution engine for microcontrollers. Bare-metal operation of MicroPython is possible on some IoT hardware such as the pyboard. Variants of MicroPython such as CircuitPython [3] can run bare-metal on more hardware.
MicroPython execution itself is based on a two stage approach. In a first phase, based on an abstract syntax tree, a parser/lexer helps compile 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. This makes development of applications simple from a developer perspective. Thanks to the parsing phase which can be performed by MicroPython, a minimal toolchain is possible: the container developer only requires a text editor. 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.

1.3 WebAssembly

WebAssembly (Wasm [4]) is standardized by the World Wide Web Consortium (W3C). Initially aimed at portable web applications, Wasm has been adapted to microcontrollers.
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.
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. 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 [5]. WASI standarizes access to operating system facilities such as files, network sockets, clocks and random number etc.
Several interpreters have been developed such as WASM3 and WAMR. Other interpreters are integrated into existing applications, most notably web browsers. The WASM3 and WAMR virtual machines are specificatlly geared towards stand-alone execution and optimized for embedded devices.
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.

1.4 JerryScript

JerryScript is a light JavaScript engine which targets small IoT devices. It is written in C and had a binary footprint of about $160\,\mathrm{kB}$ of ROM and $30\,\mathrm{kB}$ of RAM. JerryScript complies with ECMAScript 1.5 (full implementation).

CC BY-NC-ND 3.0 License
TinyPART Consortium Parties

We have succefully run some small scripts including simple mathematical operations, driving the board leds, conditional tests, variable manipulations, and looping. Nevertheless, the possibility of scripting remain very limited because of memory contraints: the code had to be optimized to have enough memory to run these small scripts - typically with less than 10 instructions per script.
To go further, more optimization would be required. These optimization might be at memory management, network stack, or RIOT modules level. Nevertheless, it seems difficult to run large script with a firmware with a rich set of functionality.

CC BY-NC-ND 3.0 License
TinyPART Consortium Parties

2 Execution Environment Prototypes

We provide an open-source implementation of the execution environments integrated in RIOT [6], a common operating system for low-power IoT devices [7].
The prototypes all implement similar functionality in the execution environment. The prototypes provide a temperature and faked heart rate sensor over Bluetooth LE. The temperature is read from the internal CPU sensor, the heart rate sensor is faked with an up and down counting value. For the Bluetooth LE connectivity, the Nimble Bluetooth stack is used, running on RIOT.

2.1 Femto-Containers

The Femto-Containers prototype implements the previously described functionality using the Femto-Containers isolation mechanism. Two instances of Femto-Containers are running on the device, one for the temperature sensor and one for the heart rate. The temperature sensor is read via bindings to the operating system sensor functions. For the heart rate, the previous value and direction is passed to the Femto-Container instance.

2.2 Micropython

The Micropython prototype implements the functionality using two small python scripts. Again, one script implements the temperature sensor, using bindings to the RIOT sensor read functions. The formatted temperature is returned as value from the script.
For the heart rate sensor, two globals are used in the python application, these are retained between invocations of the python script. On the OS side, the global with the current heart rate is retrieved after the script run.

2.3 WebAssembly

The WebAssembly implementation uses WAMR as WebAssembly implementation for the application. The temperature uses the WASM-specific bindings to RIOT and returns the temperature as application result. The heart rate application stores the heart rate and direction as local variables in the application, which are retained between invocations. The current heart rate is returned as result of the application.

2.4 Application sizes

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 based on measurements from [8].

Runtime	Firmware Flash size	Firmware RAM size
Femto-Container	83.8 kB	11.7 kB
WebAssembly	137.8 kB	17.0 kB
MicroPython	189.1 kB	37.5 kB

Table 2: Sizes of the logic hosted in different runtimes on Cortex-M4.

The memory required on the IoT device differs wildly. In particular, techniques based on script interpreters (RIOTjs and MicroPython) require the biggest ROM memory budget, above 100 kB. 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.

CC BY-NC-ND 3.0 License
TinyPART Consortium Parties

3 Discussion

This document shows multiple solutions for runtime scripting and execution on embedded systems. While each solution presented here shows only a proof of concept application, multiple results can be derived from this.
First the memory size used by each solution differs wildly. Compared to the MicroPython runtime, Femto-Containers requires less than half the flash and RAM to run similar functionality. WebAssembly shows a middle ground, still significantly more than Femto-Containers. This results in potentially requiring a microcontroller with more resources when using MicroPython or WebAssembly.
The Javascript and MicroPython solutions have the advantage that the script does not have to be compiled before transferring it to the device which eases programming. Both WebAssembly and Femto-Containers require the application to be compiled into the final binary which improves performance due to lower memory footprint and an avoided compilation step. In addition, both allow for programming in multiple languages.
Introducing scripting capabilities on a constrained IoT is obviously a challenging task and after exploring different technologies for our Execution Environment, we found that the adoption of a technology should be balanced with the memory footprint and execution speed. We therefore consider Javascript and MicroPython not yet ready for efficient execution on a constrained device.
In the future we will use Femto-Containers and WebAssembly for the Execution Environment as they seem the most suitable for constrained devices.

CC BY-NC-ND 3.0 License
TinyPART Consortium Parties

4 Delivery source code location

The prototypes described in this document are listed in table 3.

Source Code delivered			
Prototype	License	Location	Entity
RIOT	OpenSource	TinyPART github D2.1.1 repository	INRIA
femtocontainers	OpenSource	TinyPART github D2.1.1 repository	INRIA
micropython	OpenSource	TinyPART github D2.1.1 repository	INRIA
WASM	OpenSource	TinyPART github D2.1.1 repository	INRIA
TinyContainer-Jerryscript	Proprietary	Orange internal gitlab	Orange

Table 3: Prototypes delivered

CC BY-NC-ND 3.0 License
TinyPART Consortium Parties

References

[1] M. Fleming, "A Thorough Introduction to eBPF," *Linux Weekly News*, 2017.

[2] "MicroPython." `https://micropython.org/`.

[3] "CircuitPython." `https://circuitpython.org/`.

[4] A. 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*, pp. 185–200, 2017.

[5] W3C, "WASI: libc Implementation for WebAssembly." `https://github.com/WebAssembly/wasi-libc`.

[6] "RIOT." `https://www.riot-os.org/`. Accessed: 2022-11-24.

[7] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot: An open source operating system for low-end embedded devices in the iot," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[8] K. Zandberg and E. Baccelli, "Femto-containers: Devops on microcontrollers with lightweight virtualization & isolation for iot software modules," *arXiv preprint arXiv:2106.12553*, 2021.

CC BY-NC-ND 3.0 License
TinyPART Consortium Parties