

Enabling Post-quantum Secure Software Reconfiguration of Heterogeneous Resource-Constrained Networked Devices

Dissertation

zur Erlangung des Grades eines Doktors der Naturwissenschaften

am Fachbereich Mathematik und Informatik der
Freien Universität Berlin

vorgelegt von

Koen Zandberg

Berlin, 2025

Advisor Prof. Dr. Emmanuel Baccelli
Reviewer Prof. Dr. Mesut Güneş

1	INTRODUCTION	7
1.1	SCOPE OF THIS WORK	9
1.2	RESEARCH QUESTIONS	10
1.3	THESIS CONTRIBUTIONS	10
1.4	PUBLISHED RESULTS	12
1.4.1	CODE CONTRIBUTIONS	14
1.5	OUTLINE	14
2	BACKGROUND	15
2.1	HARDWARE CONSTRAINTS	15
2.2	NETWORK CONNECTIVITY	16
2.2.1	IEEE 802.15.4	16
2.2.2	LoRa	17
2.3	FIRMWARE AND OPERATING SYSTEMS	17
2.3.1	RIOT	18
2.3.2	CONTIKI-NG	18
2.3.3	FREERTOS	18
2.3.4	ZEPHYR	19
2.3.5	NUTTX	19
2.3.6	MONGOOSE OS	19
2.3.7	Tock	19
2.4	SECURITY PRIMITIVES	19
2.4.1	SECURITY ASPECTS	20
2.4.2	SYMMETRIC KEY ENCRYPTION	20
2.4.3	PUBLIC KEY CRYPTOGRAPHY	21
2.4.4	HASH FUNCTIONS	21
2.5	SOFTWARE UPDATES FOR CONSTRAINED DEVICES	22
2.5.1	EMBEDDED SOFTWARE DESIGN ON LOW-END IoT DEVICES	22
2.5.2	UPDATE FRAMEWORK BACKEND	23
2.5.3	NETWORK TRANSPORT TO THE FIRMWARE TOWARDS THE IoT DEVICES	24
2.5.4	OPEN STANDARDS FOR SECURE CONSTRAINED FIRMWARE UPDATES	24
2.6	AUTHENTICATION THROUGH DIGITAL SIGNATURES	26
2.6.1	POST-QUANTUM SIGNATURE SCHEMES	27
2.6.2	PRE-QUANTUM ALGORITHMS	28
2.7	EMBEDDED SOFTWARE VIRTUALISATION AND SANDBOXING	29
2.7.1	SCRIPT ENVIRONMENTS	29
2.7.2	VIRTUAL MACHINES	31
3	COMPARATIVE EVALUATION OF POST- AND PRE-QUANTUM DIGITAL SIGNATURES FOR CONSTRAINED DEVICES	34
3.1	IMPACT OF CRYPTOGRAPHIC PRIMITIVES ON FIRMWARE	34
3.1.1	FIRMWARE UPDATE SIZES AND POST-QUANTUM SIGNATURES	35
3.2	CRYPTOGRAPHIC LIBRARY SELECTION	36
3.2.1	PRE-QUANTUM SIGNATURE SCHEMES	36
3.2.2	POST-QUANTUM SIGNATURE SCHEMES	37
3.2.3	HASH FUNCTIONS	38

3.3	BENCHMARKS	39
3.3.1	BENCHMARK HARDWARE SETUP	39
3.3.2	PRE-QUANTUM SIGNATURE BASELINE	39
3.3.3	POST QUANTUM CRYPTOGRAPHY PRIMITIVES	40
3.3.4	HASH FUNCTION BENCHMARKS	41
3.4	IMPACT OF POST-QUANTUM PRIMITIVES ON EMBEDDED DEVICES	42
3.4.1	THE COST OF POST-QUANTUM SECURITY	42
3.4.2	THE COST OF POST-QUANTUM ALGORITHMS WITH FIRMWARE UPDATES	43
3.4.3	REAL-WORLD USABILITY OF POST-QUANTUM DIGITAL SIGNATURES	44
3.5	DISCUSSION	45
3.5.1	COMPARISON TO PRE-QUANTUM DIGITAL SIGNATURES	45
3.5.2	IMPACT ON REAL WORLD SCENARIOS	45
3.6	CONCLUSION	46
4	SECURE FIRMWARE UPDATE FRAMEWORK FOR LOW-POWER INTERNET OF THINGS	47
4.1	UPDATE ARCHITECTURE	48
4.1.1	DEVICE UPDATE NOTIFICATION	49
4.1.2	MANIFEST RETRIEVAL	49
4.1.3	MANIFEST AUTHENTICITY VERIFICATION	49
4.1.4	FIRMWARE UPDATE APPLICABILITY CHECKS	49
4.1.5	FIRMWARE RETRIEVAL	49
4.1.6	FIRMWARE AUTHENTICITY VERIFICATION	50
4.1.7	FIRMWARE INVOCATION	50
4.2	FIRMWARE REQUIREMENTS	50
4.3	MANIFEST DESIGN	51
4.4	IMPLEMENTATION OF SECURE FIRMWARE UPDATES	52
4.4.1	SCENARIO SETUP	52
4.4.2	COMPONENTS AND FUNCTIONAL OVERVIEW	53
4.5	CONFIGURABILITY OF THE PROTOTYPE	55
4.5.1	BASELINE	55
4.5.2	BASIC-OTA	56
4.5.3	IPv6-OTA	56
4.5.4	SUIT-OTA	56
4.5.5	LWM2M-OTA	57
4.6	PERFORMANCE EVALUATION	57
4.7	RELATIVE IMPACT OF CRYPTOGRAPHIC LIBRARIES	58
4.8	EVALUATING THE COST OF THE OTA UPDATE FUNCTIONALITY	59
4.8.1	THE COST OF OTA	59
4.8.2	THE COST OF STANDARDS COMPLIANCE FOR OTA	60
4.9	SECURITY ASSESSMENT	61
4.9.1	FIRMWARE TAMPERING	61
4.9.2	FIRMWARE REPLAY	62
4.9.3	OFFLINE DEVICE ATTACK	62
4.9.4	DEVICE FIRMWARE MISMATCH	62
4.9.5	FLASH MEMORY LOCATION MISMATCH	62
4.9.6	UNEXPECTED PRECURSOR IMAGE	63
4.9.7	FIRMWARE REVERSE ENGINEERING	63
4.9.8	RESOURCE EXHAUSTION	63
4.10	DISCUSSION	64
4.10.1	MAKING THE FIRMWARE UPDATE RELIABLE IS KEY	64
4.10.2	USE DELEGATION CAPABILITIES WITH CARE	64

4.10.3	SHIELDING AGAINST RESOURCE EXHAUSTION AND BEST-BEFORE VULNERABILITIES	64
4.10.4	REAL-WORLD REQUIREMENTS MAKE FIRMWARE UPDATES COMPLEX	65
4.10.5	IoT SOFTWARE UPDATES ARE NOT JUST FOR CRITICAL INFRASTRUCTURE	65
4.11	CONCLUSION	65
5	RBPF: A TINY SOFTWARE-ONLY VIRTUAL MACHINE FOR INTERNET OF THINGS FIRMWARE	67
5.1	DESIGN GOALS & REQUIREMENTS	68
5.1.1	MINIMAL MEMORY FOOTPRINT	68
5.1.2	NO RELIANCE ON HARDWARE-SPECIFIC MECHANISM FOR MEMORY PROTECTION	68
5.1.3	TOLERABLE CODE EXECUTION SLUMP	68
5.1.4	SMALL APPLICATION CODE SIZE	68
5.2	VIRTUAL MACHINE DESIGN	69
5.2.1	EXECUTION HOOKS	69
5.2.2	ARCHITECTURE	70
5.2.3	MEMORY PROTECTION	71
5.3	EXPERIMENTAL EVALUATION	72
5.3.1	COMPUTING BENCHMARK SETUP	73
5.3.2	NETWORKED BENCHMARK SETUP	73
5.3.3	VIRTUAL MACHINE MEMORY REQUIREMENT	73
5.3.4	APPLICATION SIZE COMPARISON	74
5.3.5	RBPF WITH LOGIC INVOLVING IoT NETWORKING	74
5.3.6	APPLICATION FLASH REQUIREMENT	75
5.3.7	RUNTIME MEMORY REQUIREMENT	75
5.4	DISCUSSION	75
5.4.1	INHERENT LIMITATIONS WITH A VM	75
5.4.2	DECREASING WASM RAM USAGE	76
5.4.3	IMPROVING RBPF EXECUTION TIME OVERHEAD	76
5.4.4	DECREASING RBPF SCRIPT SIZE OVERHEAD	76
5.4.5	EXTENDING RBPF SANDBOXING GUARANTEES	77
5.5	CONCLUSION	77
6	SANDBOXED FUNCTION EXECUTION FOR DEVOPS-STYLE RECONFIGURATION OF CONSTRAINED DEVICES	78
6.1	THREAT MODEL	78
6.1.1	MALICIOUS TENANT	79
6.1.2	MALICIOUS CLIENT	79
6.1.3	ATTACK VECTORS	79
6.2	EMBEDDED RUNTIME ARCHITECTURE DESIGN	80
6.2.1	USE OF AN RTOS WITH MULTI-THREADING	80
6.2.2	NO ASSUMPTIONS ON MICROCONTROLLER HARDWARE	80
6.2.3	USE OF ULTRA-LIGHTWEIGHT VIRTUALISATION	80
6.2.4	USE OF SIMPLE CONTAINERIZATION	81
6.2.5	ISOLATION & SANDBOXING THROUGH VIRTUALISATION	81
6.2.6	EVENT-BASED LAUNCHPAD EXECUTION MODEL	81
6.2.7	LOW-POWER SECURE RUNTIME UPDATE PRIMITIVES	81
6.3	ULTRA-LIGHTWEIGHT VM MICRO-BENCHMARK	82
6.3.1	CONSIDERING SIZE	82
6.3.2	CONSIDERING SPEED	83
6.3.3	CONSIDERING VM ARCHITECTURE & SECURITY	84
6.3.4	CHOICE OF VIRTUALISATION	84
6.4	FEMTO-CONTAINER RUNTIME IMPLEMENTATION	84

6.4.1	USE OF RIOT MULTI-THREADING	84
6.4.2	ULTRA-LIGHTWEIGHT VIRTUALISATION USING eBPF INSTRUCTION SET	85
6.4.3	ISOLATION & SANDBOXING	86
6.4.4	HOOKS & EVENT-BASED EXECUTION	87
6.5	USE-CASE PROTOTYPING WITH FEMTO-CONTAINERS	88
6.5.1	PROGRAMMING MODEL	88
6.5.2	KERNEL DEBUG CODE EXAMPLE	88
6.5.3	NETWORKED SENSOR CODE EXAMPLE	89
6.6	FORMAL VERIFICATION	93
6.6.1	TARGETED REQUIREMENTS FORMALIZATION.	93
6.6.2	FORMAL VERIFICATION APPROACH.	93
6.7	PERFORMANCE EVALUATION	94
6.8	HOSTING ENGINE ANALYSIS	94
6.9	EXPERIMENTS WITH A SINGLE CONTAINER	95
6.9.1	FEMTO-CONTAINERS WITH MULTIPLE INSTANCES	96
6.10	OVERHEAD ADDED BY HOOKS	97
6.11	DISCUSSION	98
6.11.1	VIRTUALISATION VS POWER-EFFICIENCY	98
6.11.2	CONTROLLING TENANT PRIVILEGES	98
6.11.3	INSTALL TIME VS EXECUTION TIME	98
6.11.4	TENANT-LOCAL STORAGE OF VALUES	99
6.11.5	SECURITY VS LONG-RUNNING APPLICATION SUPPORT	99
6.11.6	FIXED- VS VARIABLE-LENGTH INSTRUCTIONS	99
6.12	CONCLUSION	100
7	CASE STUDY: SECURE SOFTWARE RECONFIGURATION ON NANOSATELLITE	101
7.1	THINGSAT	101
7.1.1	SYSTEM ARCHITECTURE DESIGN	101
7.1.2	COMMUNICATION CHARACTERISTICS OVERVIEW	103
7.1.3	INTERMITTENT COMMUNICATION AND POWER SUPPLY	104
7.1.4	HOSTED PAYLOAD UPDATE REQUIREMENTS	104
7.2	SOFTWARE UPDATE IMPLEMENTATION	104
7.2.1	SECURITY REQUIREMENTS	104
7.2.2	TRUST ANCHOR	105
7.2.3	CUBEDATE SOFTWARE LIFE-CYCLE PHASES	105
7.2.4	SUPPORTING NETWORK TRANSPORT HETEROGENEITY	106
7.2.5	SUPPORTING UPDATED SOFTWARE HETEROGENEITY	107
7.2.6	LOW-POWER END-TO-END SECURITY USING SUIT	108
7.3	PERFORMANCE EVALUATION	109
7.3.1	MEMORY FOOTPRINT OVERHEAD	109
7.3.2	NETWORK TRANSFER OVERHEAD	110
7.4	DISCUSSION	110
7.4.1	PORTABILITY	110
7.4.2	NETWORK STACK SIMPLIFICATION & STANDARDIZATION	110
7.4.3	ALTERNATIVE CRYPTOGRAPHIC PRIMITIVES	111
7.5	CONCLUSION	111
8	CONCLUSION	112
8.1	SUMMARY	112
8.2	PERSPECTIVES	114
A	BIBLIOGRAPHY	116

ENABLING POST-QUANTUM SECURE SOFTWARE RECONFIGURATION OF HETEROGENEOUS RESOURCE-CONSTRAINED NETWORKED DEVICES	6
B LIST OF ACRONYMS	126
C CURRICULUM VITAE	128
D DEUTSCHE ZUSAMMENFASSUNG	130

CHAPTER 1

INTRODUCTION

The number of small constrained embedded devices deployed around in our personal environment keeps growing. Gartner [74] estimates a 8.4 billion internet connected devices in use in 2017. This growth continued with over 18.8 billion devices connected in 2024[95]. In contrast to general purpose computer systems such as desktops, laptops and servers, this class of devices can be significantly more constrained in resources such as in an industrial control system or in a vehicle [115, Definition 1.1]. These embedded devices serve a specific tailored purpose in the system they are embedded in, often a cyber-physical purpose where physical systems are operated by the embedded device. Software bugs and vulnerabilities can influence physical operations and might be able to cause physical harm when triggered or exploited. The cyber-physical nature of these devices can make these embedded devices an attractive target for attackers, where the constrained nature makes it harder to defend against attackers and the physical aspect can cause direct harm. For these reasons it is paramount to either prevent security vulnerabilities, or when bugs are discovered, have a mechanism in place to patch the bug in the software. Without the ability to resolve security issues in the field, it is only a matter of time until the device becomes a security liability.

With the emergence of the Internet of Things (IoT) [97], embedded systems are more connected than ever. These devices gather data measurements about their surrounding environment, process information and control physical systems. In addition, these devices often transmit their measurements to centralized server-based systems. The interconnected nature of these devices increases the attack surface of these devices and makes them an attractive target for malicious actors. With the internet connectivity of IoT devices, it must be assumed attackers are able to directly communicate with the devices. Relying on an unconstrained gateway device to guard access to the constrained devices is no longer sufficient. These constrained devices must thus be sufficiently secure to prevent damage caused by such malicious actors.

The added internet connectivity, provided by state-of-the-art network architectures, increases the software complexity on IoT devices. The network stack must be able to both transmit the measurement data to centralized systems over the internet, as well as receive instructions and configuration from these centralized systems. As such the handling routines for these devices must be able to handle arbitrary input

received via the network connection. Given the complexity of network protocols used in the constrained embedded space, these network handler routines are susceptible to vulnerabilities [142]. The issue is exacerbated by the lack of common software isolation mechanisms present on general purpose computers, as the hardware features to mitigate or isolate common vulnerabilities are not present on the constrained devices [169].

A large fraction of IoT devices are based on microcontrollers, highly integrated devices and optimized for power consumption, production cost or chip size. Conventional approaches to security used with general purpose operating systems, such as relying on security features of the hardware platform, are not available on these systems. The modular multi-processing approach used by conventional operating systems, by splitting software components into multiple isolated processes, relies on functionality provided by platform hardware such as a memory management unit (MMU) not available on microcontrollers. Instead, only a few kilobytes of memory are available for the operating system. Additional hardware features for protection are fully optional and often unavailable. Thus the impact of vulnerabilities in the software can be used to gain full access to the system. Therefore, protecting the software must fit within the limited memory and processing power available, and cannot assume the presence of hardware security features.

The ability to isolate processes and run multiple different software components from different stakeholders, a solved issue on general purpose computers through virtualisation and containerization, is a challenging subject area in constrained devices. Here again the lack of hardware security and virtualisation features available on general purpose computers, hampers the deployment of similar solutions in the constrained IoT space. While isolating sensitive data processing from the attack surface provided by network handlers can curb entire classes of vulnerabilities, deploying such isolation mechanisms is not trivial on constrained devices [174].

However, the ability to resolve vulnerabilities in a device is not the only use-case for over-the-air updates. Without updates, the features provided by a device are fixed at deployment time. Adding new functionality and enhancing existing features of a device allows deployments to be more flexible in their purpose, and allows stakeholders to adapt existing devices to new developments. The scope of these adaptations can range from simple configuration updates to adjustments of single software components to the deployment of a completely new operating system on the devices. This flexibility can extend the deployment lifetime of the device and allow for adjusting to new and unexpected developments during the device lifetime.

With the complexity of current software, the discovery of new bugs is just a matter of time. While techniques exist to discover and mitigate bugs before software is deployed in production systems, the ability to resolve bugs while the device is already deployed in the field is a mandatory feature. The European Union mandates the ability to update software running on devices for the full life cycle of the device [68]. Given the constrained nature of the IoT devices in the field,

approaches to software updates used with general purpose operating systems are not always possible. Instead a novel, holistic approach suitable for constrained devices is required. This approach must allow for updating software components on the device without requiring personnel to physically manipulate the device. Devices can be deeply embedded in existing structures, or simply unreachable without significant effort. For example with satellite-based systems [111], while still a significant target for cyberattacks, are not accessible by conventional means when an issue arises.

Furthermore, the isolation of software components within the operating system, without any reliance on the optional security hardware available, can significantly enhance the flexibility and security of the constrained system. While the lack of resources in these constrained devices must be taken into account here and poses a challenge, the isolation of software modules provides both security benefits and allows for modularization of the software components during updates.

The advent of quantum computers further complicate the matter for the state-of-the-art solutions. Conventional cryptographic algorithms used to secure network communication, have limited time remaining before they can no longer be considered secure. While research on post-quantum cryptographic algorithms is on-going, current devices require a level of crypto-agility, being able to adapt to changes in the cryptographic stack and switch algorithms when novel algorithms are developed. Furthermore, conventional (pre-quantum) cryptographic algorithms benefit from the availability of implementations optimized for constrained embedded systems. Given the novel nature of currently available post-quantum cryptographic algorithms, these have not yet had the multiple development cycles available to older algorithms. For the constrained embedded devices, algorithms and implementations must be made available and suitable for the limited resources available on these platforms.

1.1 SCOPE OF THIS WORK

When considering embedded systems in general, the full scope can range any computational device integrated into a larger system. Within these systems embedded processors have a well-defined design purpose. The processing power and memory available to these systems can range from a few kilobytes to multiple gigabytes of memory, running a monolithic software stack or a full blown general purpose operating system tailored to the application of the system.

However, in this work the focus is on the constrained embedded systems, restricted to class 0 and 1 type of devices [41]. These systems are severely limited in computational power and memory. For these devices, the processor is often running between 10 MHz to 500 MHz. Memory on these devices is 64 kB to 1024 kB of flash and 10 KiB to 100 KiB of RAM.

This type of devices is unable to provide the hardware functionality required to run Linux or other general purpose operating system.

Instead these device run a single monolithic software system called firmware.

Similar to the processing power and memory, the network links available to the devices considered is also constrained. Network throughput is limited to 10 kbps to 1024 kbps and potentially asymmetric in nature. The network links provided on these devices can have widely different upload and download throughputs.

Auxiliary security hardware components are assumed to be absent from the platforms considered. Some microcontrollers contain extra peripherals aimed at security. The memory protection unit (MPU) or Physical Memory Protection (PMP) provides granular permissions for memory access and allows segmenting different software modules.

While solutions can be designed to rely on these components, a considerable number of devices do not have access to these components but still require robust security and update solutions.

1.2 RESEARCH QUESTIONS

The main research question of this work is:

How can we enable post-quantum secure software reconfiguration of heterogeneous resource-constrained network devices using open solutions?

This research question can be divided into multiple sub questions.

- RQ1** Which post-quantum primitives are suitable in practice on resource-constrained microcontroller-based devices?
- RQ2** How can resource-constrained device secure software updates be generalized and democratized?
- RQ3** How can individual software components be isolated and executed securely (sandboxed) on resource-constrained devices, with or without auxiliary hardware security components?
- RQ4** How can a multi-tenant (sandboxed) environment be supported on a resource-constrained device, analogous to a cloud paradigm e.g. `aclFaaS`.

1.3 THESIS CONTRIBUTIONS

In this thesis, I present my work on secure software reconfiguration on small embedded systems in a post-quantum, low-throughput network scenario.

In the space of post-quantum security I evaluate a number of signature schemes on small microcontrollers. This work is published in “Quantum-Resistant Software Update Security on Low-Power Networked Embedded Devices” [4] whereby I survey post-quantum signature schemes suitable for embedded systems. I compare the selected cryptographic signature schemes experimentally on a number

“Quantum-Resistant Software Update Security on Low-Power Networked Embedded Devices”[4]

“Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check”[3]

A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest[7]

“Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF”[1]

“End-to-End Mechanized Proof of an eBPF Virtual Machine for Microcontrollers”[8]

“Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers”[2]

“Cubedate: Securing Software Updates in Orbit for Low-Power Payloads Hosted on CubeSats”[6]

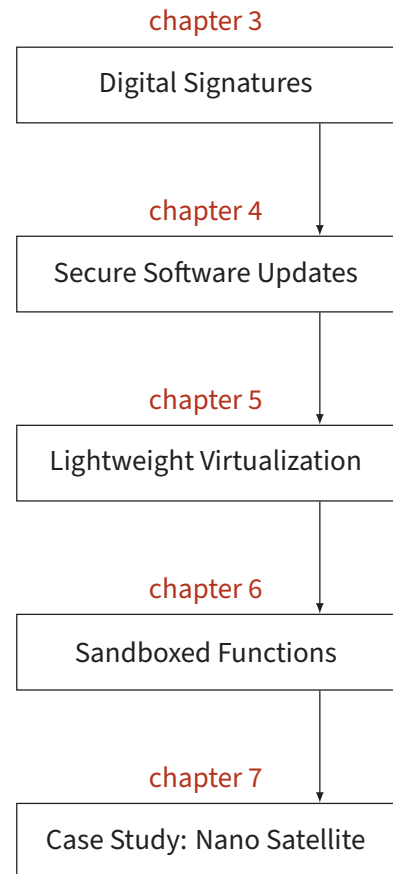


Figure 1.1: Contributions of this thesis in relation to the academic output

of microcontrollers against the main pre-quantum signature schemes. I show that while post-quantum-security is indeed doable on these devices, large differences exist between the different cryptographic primitives, their implementation, and the burden they put on the device.

To achieve secure firmware updates on small embedded systems, I contribute to the design and standardization of a secure metadata format for payloads. This work has been published in *A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest*[7] which specifies the SUIT standard to which I contribute. Furthermore, I present the first results of experiments using SUIT for firmware updates on a common IoT operating system, for both pre-quantum and post-quantum security levels [3, 4]. I published an open source implementation of SUIT able to deliver arbitrary firmware updates for the RIOT operating system. I show that firmware updates secured with SUIT can be achieved on a large variety of devices, including the smallest of microcontrollers.

In the domain of small virtual machines for microcontrollers: I design rBPF, a small software-only virtual machine optimized for small virtualised applications on embedded systems. I present the design

and implementation of rBPF as lightweight modular sandbox in RIOT. I compare WebAssembly and rBPF on embedded systems and show the minimal impact rBPF has on the total firmware size. I evaluate the capabilities of rBPF for running small business logic applications. I show that rBPF can be deployed with minimal impact on the firmware size. This work has been published before as [1].

Aiming to provide an environment for executing functions: I build Femto-Container on top of rBPF as a Functions-as-a-Service (FaaS)-like runtime for debugging and enhancing firmware on small embedded systems. This work has been published as “Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers” [2] whereby I comparatively evaluate Femto-Container against rBPF and WebAssembly. The implementation of Femto-Containers is so small that, in collaboration with formal verification specialists, we provide an implementation of the hosting engine providing formal guarantees on memory- and fault-isolation. I thus demonstrate how Femto-Container provides an attractive alternative as hosting engine for multi-tenant functions on embedded systems.

Finally, aiming to verify the design of update components, I provide a case-study: ThingSat, a low-power, low-cost payload hosted on a nano-satellite (CubeSat) launched in Low-Earth Orbit (LEO) in 2023. In this context I survey open standard protocols for secure over-the-air software (re-)configuration on ThingSat. I then define Cubedate, a generic architecture combining several protocols, SUIT and the work I present in this thesis to enable various levels of software updates for ThingSat. I evaluate an open source implementation of Cubedate [6].

1.4 PUBLISHED RESULTS

The outlined research results have led to publications of the research results in the form of peer-reviewed papers and an open specification under active review. The following work has been published in scope of this thesis:

- [1] Koen Zandberg and Emmanuel Baccelli. “Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF”. In: *9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks, PEMWN 2020, Berlin, Germany, December 1-3, 2020*. IEEE, 2020, pp. 1–6. DOI: [10.23919/PEMWN50727.2020.9293081](https://doi.org/10.23919/PEMWN50727.2020.9293081). URL: <https://doi.org/10.23919/PEMWN50727.2020.9293081>.
- [2] Koen Zandberg, Emmanuel Baccelli, Shenghao Yuan, Frédéric Besson, and Jean-Pierre Talpin. “Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers”. In: *Middleware ’22: 23rd International Middleware Conference, Quebec, QC, Canada, November 7 - 11, 2022*. Ed. by Paolo Bellavista, Kaiwen Zhang, Abdelouahed Gherbi, Saurabh Bagchi, Marta Patiño, Giuseppe Di Modica, and Julien Gascon-Samson. ACM, 2022, pp. 161–173.

- DOI: [10.1145/3528535.3565242](https://doi.org/10.1145/3528535.3565242). URL: <https://doi.org/10.1145/3528535.3565242>.
- [3] Koen Zandberg, Kaspar Schleiser, Francisco Acosta Padilla, Hannes Tschofenig, and Emmanuel Baccelli. “Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check”. In: *IEEE Access* 7 (2019), pp. 71907–71920. DOI: [10.1109/ACCESS.2019.2919760](https://doi.org/10.1109/ACCESS.2019.2919760). URL: <https://doi.org/10.1109/ACCESS.2019.2919760>.
 - [4] Gustavo Banegas, Koen Zandberg, Emmanuel Baccelli, Adrian Herrmann, and Benjamin Smith. “Quantum-Resistant Software Update Security on Low-Power Networked Embedded Devices”. In: *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. Lecture Notes in Computer Science. Springer, 2022, pp. 872–891. DOI: [10.1007/978-3-031-09234-3_43](https://doi.org/10.1007/978-3-031-09234-3_43). URL: https://doi.org/10.1007/978-3-031-09234-3_43.
 - [5] Zhaolan Huang, Koen Zandberg, Kaspar Schleiser, and Emmanuel Baccelli. “RIOT-ML: toolkit for over-the-air secure updates and performance evaluation of TinyML models”. In: *Annals of Telecommunications* (2024), pp. 1–15.
 - [6] François-Xavier Molina, Emmanuel Baccelli, Koen Zandberg, Didier Donsez, and Olivier Alphand. “Cubedate: Securing Software Updates in Orbit for Low-Power Payloads Hosted on CubeSats”. In: *12th IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks, PEMWN 2023, Berlin, Germany, September 27-29, 2023*. IEEE, 2023, pp. 1–6. DOI: [10.23919/PEMWN58813.2023.10304910](https://doi.org/10.23919/PEMWN58813.2023.10304910). URL: <https://doi.org/10.23919/PEMWN58813.2023.10304910>.
 - [7] Brendan Moran, Hannes Tschofenig, Henk Birkholz, Koen Zandberg, and Øyvind Rønningstad. *A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest*. Internet-Draft draft-ietf-suit-manifest-25. Work in Progress. Internet Engineering Task Force, Feb. 2024. 101 pp. URL: <https://datatracker.ietf.org/doc/draft-ietf-suit-manifest/25/>.
 - [8] Shenghao Yuan, Frédéric Besson, Jean-Pierre Talpin, Samuel Hym, Koen Zandberg, and Emmanuel Baccelli. “End-to-End Mechanized Proof of an eBPF Virtual Machine for Microcontrollers”. In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022, pp. 293–316. DOI: [10.1007/978-3-031-13188-2_15](https://doi.org/10.1007/978-3-031-13188-2_15). URL: https://doi.org/10.1007/978-3-031-13188-2_15.

1.4.1 CODE CONTRIBUTIONS

In addition to academic papers, contributions have been made to the operating system RIOT [30] to incorporate the results from the research into public code. Most notably the inclusion of SUIT firmware update capabilities [9, 13] in RIOT as default over-the-air firmware update mechanism.

- [9] Schleiser, Kaspar and Zandberg, Koen and Abadie, Alexandre and Molina, François-Xavier. *sys/suit: initial support for SUIT firmware updates*. 2019. URL: <https://github.com/RIOT-OS/RIOT/pull/11818>.
- [10] Zandberg, Koen. *libcose: Constrained node COSE library*. 2022. URL: <https://github.com/bergzand/libcose>.
- [11] Zandberg, Koen. *NanoCBOR: CBOR library aimed at heavily constrained devices*. 2024. URL: <https://github.com/bergzand/NanoCBOR>.
- [12] Zandberg, Koen. *rBPF: Initial include of small virtual machine*. 2021. URL: <https://github.com/RIOT-OS/RIOT/pull/19372>.
- [13] Zandberg, Koen. *SUIT: Introduction of a payload storage API for SUIT manifest payloads*. 2020. URL: <https://github.com/RIOT-OS/RIOT/pull/15110>.
- [14] Zandberg, Koen and Baccelli, Emmanuel. *Femto-Containers: Femto-Containers RIOT Implementation & Hands-on Tutorials*. 2022. URL: https://github.com/future-proof-iot/Femto-Container_tutorials.

During the span of this work I maintained and contributed more than 900 code commits to open source software projects, including RIOT [30], libcose [10] and NanoCBOR [11].

1.5 OUTLINE

This thesis starts with a chapter providing background and related work on the fundamental building blocks at play, including a primary on cryptography, low-power hardware and software, low-power networking, virtualisation and sandboxing (chapter 2). In the following chapters, the contributions of this thesis are developed. Post-Quantum digital signatures are evaluated and compared against pre-quantum digital signatures in chapter 3. Next the over the air update mechanism leveraging SUIT manifests are presented in chapter 4. Following is chapter 5 describing the rBPF virtual machine (VM). The design of rBPF is explained and measurements to show the runtime overhead and impact of adding the rBPF to a typical firmware are presented. In chapter 6, Femto-Container is presented as extension on top of rBPF. Finally a case-study using the Cubedate satellite firmware management system is presented in chapter 7.

CHAPTER 2

BACKGROUND

The embedded devices connected to the internet, known as the **IoT**, are growing and tend towards instrumenting all aspects of our environment. To this end, billions of new devices are gradually being deployed on the one hand, and on the other hand, retro-fitting supplements legacy devices with similar capabilities for communication and on-board computation. Managing the firmware and other software components on these devices requires a coherent system addressing multiple challenges:

- Secure authentication of messages through digital signatures.
- Flexible update mechanisms for firmware.
- Secure execution of potentially untrusted code on constrained devices.
- Integration of sandbox environments into the operating system.

For each of these challenges, there exist previous work to address the challenges. This chapter presents background information on the scope of this work and presents a number of existing solutions to the challenges involved. First the constraints presented by the hardware involved and the hardware and software capabilities are presented. As one of the requirements with firmware management heavily involves security, a quick security primer with relevant security primitives is given. Following this security primer, firmware updates themselves are discussed with different existing solutions and challenges involved. Given the need for digital signatures with firmware updates, a number of relevant existing post-quantum and pre-quantum digital signatures are presented. At last virtualisation and sandboxing environments are presented, with a deeper focus on WebAssembly and eBPF given their relevance to this work.

2.1 HARDWARE CONSTRAINTS

The type of devices deployed here are constrained in processing resources. Microcontrollers deployed with a cyber-physical purpose can be classified [41] based on the resources available on the device. The resources of these devices range between 10 kB to 250 kB in ROM with RAM between 1 kB to 50 kB. While it is expected that the resource boundaries of constrained devices move over time, gains available

with personal (desktop) computer hardware will not directly translate to the embedded space. Increases in computational power will more likely be invested in power requirement reductions and not necessarily in increased computer power.

Devices in the constrained cyber-physical systems space use small processor cores. Popular examples of these are the ARM Cortex-M class processor cores [24], RISC-V processors [94] with limited feature set, and the Cadence Xtensa processor cores [53]. RISC-V cores in particular are extremely configurable and, depending on the hardware implementation, can be configured for everything between large hardware platforms and small low-power devices. In the scope of this work the RISC-V-based microcontrollers are limited in capabilities and consist of a basic RV32I core with limited extensions.

This class of devices is not only constrained in raw processing power, available peripherals is also limited. Memory protection mechanisms ubiquitous in personal computers, the MMU, are not available, or only available with very limited capabilities.

2.2 NETWORK CONNECTIVITY

The devices considered here have network connectivity to interact with other devices. When considering network connectivity, similar constraints apply as with the processing power on the devices. High-throughput network connectivity requires power and is often not required for the functionality of the device. The types of devices considered in this work often connect via low power networking alternatives such as IEEE 802.15.4 [91] or LoRA-based [112] networks. Each of these network connections bring their own advantages and restrictions, which will be elaborated on below.

2.2.1 IEEE 802.15.4

IEEE 802.15.4 network links are optimized for bidirectional communication with low power in mind. In contrast to WiFi, frame sizes are limited to 127 B, and theoretical throughput limited to 250 kbps. Furthermore, IEEE 802.15.4 provides mechanisms to create mesh networks. On top of IEEE 802.15.4, different protocols can be used to provide rich network connectivity. One of the options is Zigbee [19], a full mesh network protocol for high-level communication with the aim to create personal area networks for devices such as small low-power home automation devices. Another option is 6LoWPAN [122], which defines a frame format for the transmission of IPv6 packets over IEEE 802.15.4 networks. Via 6LoWPAN, devices can be connected transparently to the rest of the internet via a so called border router. While the restriction of 127 B frame size still applies, further header compression and protocols optimized for constrained networks further lessen the burden on the network and devices.

One of the protocols used in this space is Constrained Application Protocol (CoAP) [147], a specialized web transfer protocol optimized for constrained devices and networks. CoAP is specifically optimized for

machine- to-machine applications such as home automation devices and supports networks limited in throughput. Another mechanism in CoAP provides discovery of application endpoints. A large number of extensions are available for CoAP to further enhance the capabilities:

- An observe mechanism [86] that allows clients to monitor an endpoint on a server for changes in a lightweight manner.
- Block-wise transfers [45] to support transfer sizes efficiently beyond the frame size limitations of the link layer.
- Patch and fetch methods [157] to support partial access to resources on a server.
- Object security [145] to protect resources provided via end-to-end encryption.
- Echo option [21] to mitigate security issues and force clients to demonstrate reachability at its claimed network address.
- Resource directories [22] for CoAP to publish available resources to a central resource directory.

Both 6LoWPAN and CoAP are developed as open standards freely available by the Internet Engineering Task Force (IETF).

2.2.2 LORA

The LoRa specification, together with the LoRaWAN MAC layer provide a low-power and long-range communication standard. Communication ranges of more than 10 km are possible with data rates up to 50 kbps. LoRaWAN frames are received by multiple gateways, which forward the frames to a centralized network server. The network server then forwards the frame to an application server provisioned by the device owner. The network itself is reliable for moderate loads, but can show performance issues with sending acknowledgements [32].

OTHER NETWORK CONNECTIVITY STANDARDS

Other network connectivity standards are: Bluetooth Low Energy [171], Narrowband IoT [144] and Sigfox [105, 152], among others. A select number of microcontrollers also have a WiFi network peripheral on board, with the extra power consumption associated [172] when not carefully tuned for efficiency.

2.3 FIRMWARE AND OPERATING SYSTEMS

Given the constrained nature and the limited space available in the flash of the devices, specialized operating systems have been developed for these devices. The nature of these devices put severe restrictions on the firmware running on them [57]. Paradigms ubiquitous to commodity desktop and server hardware are not always applicable to the constrained devices. Traditional operating systems such as Linux or

BSD are not applicable to microcontrollers as they cannot run on the constrained capabilities provided by the microcontrollers.

The purpose operating systems specialized for microcontrollers is to manage the limited resources on these devices in a power-efficient way [83, 153]. Usually such operating systems provide a simple task scheduler, providing a notion of parallelization of different task on the system. Furthermore, access to the different hardware peripherals available on microcontrollers is usually provided through abstractions provided by the operating system.

Depending on the operating system a more extensive feature set is available. A network stack, tailored for constrained devices can be shipped with the operating system, providing connectivity out of the box. Drivers for different peripherals such as common sensors and actuators can be integrated into the operating system. Furthermore, rich services such as over-the-air update capabilities and high level language scripting support is also within reach for some operating systems. Depending on the enabled features within the firmware, the required ROM and RAM by the firmware can increase significantly, potentially exceeding what is provided by the small class 0 type of devices.

Multiple operating systems exist in active use on microcontrollers, each with their own goals and feature sets:

2.3.1 RIOT

RIOT [30] was developed with the requirements for constrained embedded and networked devices in mind. It aims to provide a developer-friendly programming model and API, providing a microkernel with multi-threading and a full 6LoWPAN network stack. RIOT is written in C, with support for C++ for libraries. RIOT has a strong modular approach to firmware and can be compiled in many different configurations. A number of configurations with their ROM and RAM usage are shown in Table 2.1

2.3.2 CONTIKI-NG

Contiki-NG [65, 131] and the Contiki precursor are both operating systems targeting constrained devices. The operating systems use an event-driven approach, relying on a cooperative scheduling mechanism approach using protothreads. Protothreads provide a lightweight pseudo-threading mechanism. As the operating system uses cooperative scheduling, priorities are not supported and the operating system relies on each process to yield voluntarily at some point during execution.

2.3.3 FREERTOS

FreeRTOS [20] is a popular RTOS used for Real-Time tasks and ported to multiple platforms. The preemptive microkernel supports multi-threading. FreeRTOS does not provide a network stack, multiple third-party network stacks can be used for internet connectivity with

Table 2.1: RIOT RAM and ROM usage for various RIOT configurations on a 32 bit Cortex-M0+ microcontroller [30]

RIOT Configu- ration	ROM	RAM
Basic RTOS	3.2 kB	2.8 kB
6LoWPAN	38.5 kB	10.0 kB
JavaScript	166.2 kB	29.1 kB
OTA-enabled	111.0 kB	17.5 kB

FreeRTOS. FreeRTOS is currently developed by Amazon Web Services and is available under a modified GPL license allowing commercial use.

2.3.4 ZEPHYR

Zephyr [173] also follows a microkernel approach with multi-thread support. Zephyr provides its own network stack, including support for links oriented towards low-power devices such as IEEE 802.15.4 and Bluetooth LE. Zephyr is developed under the Linux Foundation as operating system for resource-constrained systems.

2.3.5 NUTTX

NuttX [160] is a RTOS under the Apache foundation. Emphasis is on adherence to technical standards and small footprint to scale from 8 bit to 64 bit systems. Main governing standards for NuttX are the Portable Operating System Interface (POSIX) and American National Standards Institute (ANSI) standards. Unix and other RTOS interfaces are adopted and adapted where needed to for the constrained embedded environment.

2.3.6 MONGOOSE OS

Mongoose OS is an IoT firmware development framework for constrained devices. It provides built-in integration for cloud providers such as AWS IoT, Google IoT Core, Microsoft Azure, Adafruit IO and other generic MQTT servers. Over-the-air firmware updates and remote management is supported out of the box and it supports cryptographic accelerators and mbed TLS [110] for security.

2.3.7 TOCK

Tock [108] emphasises security on constrained embedded systems, leveraging the Rust programming language and hardware security features. Tock isolates different software components into capsules to provided memory protection and parallelization.

The lack of MMUs on microcontrollers restricts operating systems often in terms of isolation capabilities, the process memory isolation available on commodity hardware operating systems is not a given on microcontrollers. While the MPU provides an alternative way of protecting the memory space of microcontrollers, it has been shown to be difficult to apply for operating systems [174]. The result of this is that address space separation is not commonly used, or must be explicitly designed for in the operating system. For example as used with the previously mentioned Tock embedded operating system [109].

2.4 SECURITY PRIMITIVES

Management of software on a device over the network must be secured to prevent malicious actors from interfering with the device. Within IoT this requirement can be difficult to ensure, as adding extra

security measurements after deployment is impossible without update capabilities, and extra care is required to address the security of devices [73]. This starts with the goals of security themselves.

2.4.1 SECURITY ASPECTS

Three main aspects are to be considered with security [141]:

- Confidentiality: ensuring the transported data is secret and private.
- Integrity: ensuring the data is trustworthy accurate, complete, and uncorrupted.
- Availability: ensuring the machines are accessible for the relevant actors.

Different types of cryptographic primitives have been developed to address the challenges posed by these aspects [139].

2.4.2 SYMMETRIC KEY ENCRYPTION

Symmetric key encryption provides confidentiality based on a single key shared between senders and recipients. Two types of symmetric key encryption primitives can be distinguished: block ciphers and stream ciphers.

Block ciphers operate on fixed-length groups of bytes, termed blocks. Every type of block cipher uses a mode of operation for encrypting messages longer than a single block size: counter (CTR), cipher block chaining (CBC) or counter with MAC (CCM) among others. As opposed to block ciphers, Stream ciphers operate on individual bits, one at a time, and the transformation varies during the encryption. The distinction between these two modes is not always as clear as some modes used with block ciphers act effectively as stream cipher, such as counter mode with AES.

The main symmetric key encryption cipher used is AES [138], a block cipher with 128 bit blocks. The block cipher is always combined with different modes to allow for larger messages. Commodity desktop hardware often has specialized hardware available for this block cipher, and microcontrollers sometimes have a specialized peripheral for AES operations.

Another common symmetric key cipher is the ChaCha [34], a 256 bit stream cipher. ChaCha is standardized with 20 quarter rounds and the Poly1305 MAC code [36] to protect both confidentiality and integrity in a single cipher [128]. The advantage of ChaCha is the simple integer operations required per round, restricted to addition, bitwise exclusive OR and bitwise rotations. This ensures that, even with the limited processing capabilities of microcontrollers, high-throughput implementations are possible [61].

2.4.3 PUBLIC KEY CRYPTOGRAPHY

Public key cryptography provides cryptographic primitives with pairs of related keys. One of the keys of a pair can be public and openly distributed without compromising the security of the algorithm. Public key cryptography is used mainly in two ways, digital signatures and public key encryption.

Digital signature primitives allow the owner of the secret key to generate a signature for a specific message. Anyone with access to the public key can verify that the message was indeed signed by the holder of the private key and was not tampered with.

Public key encryption allows for encrypting payloads with the public key of the pair. Only the holder of the private key can decrypt the message.

Public key cryptography relies on mathematical problems termed one-way functions. Current development aims to provide a new set of primitives resistant to quantum computers, so-called post-quantum cryptography [37].

Digital signatures are used to verify the authenticity of a message, where public key encryption is used to ensure both confidentiality and authenticity of the message. Examples of primitives used for public key are the elliptic curves using the P-256 prime fields defined by National Institute of Standards and Technology (NIST) [156] and the Curve25519 [35] elliptic curve, which are discussed more in [subsection 2.6.2](#).

2.4.4 HASH FUNCTIONS

A hash (or digest) function is any function which maps an arbitrary sized input to a fixed-size output value. As such the resulting hash can serve as a representative image of the input data. Hash functions are used when a fixed-size representation of arbitrary input data is required, for example with digital signatures the hash of the payload is signed. Hash functions are often relative easy to compute, while the inverse operation is nearly impossible. Cryptographic hash functions must have special properties desirable for cryptographic applications:

- Preimage resistance: For all possible outputs, it is computationally infeasible to find an input that hashes to that output.
- Second preimage resistance: It is computationally infeasible to find any second input which has the same output as any specified input.
- Collision resistance: It is computationally infeasible to find any two distinct inputs which hash to the same output.

SHA-256

SHA-256 [85, 155] is one of the most actively used hash functions currently relevant. It provides a 256 bit hash output based on arbitrary input.

SHA-3

SHA-3 is the latest hash algorithm standardized by NIST [66, 129] hash algorithm. The hash algorithm is internally completely different from SHA-256, it is based on the Keccak [38] permutation. This provides an alternative in case the internal structure of SHA-256 is broken.

2.5 SOFTWARE UPDATES FOR CONSTRAINED DEVICES

An IoT firmware update solution is a special case of software update, and requires special care to take the earlier mentioned constraints of the device and connectivity into account. Mainly, three areas of work [51] are identified, namely:

- Embedded software design on low-end IoT devices.
- Back-end update framework to describe the firmware update.
- Network transport of the firmware towards the IoT devices.

2.5.1 EMBEDDED SOFTWARE DESIGN ON LOW-END IOT DEVICES

The software on an IoT device has to be prepared to support a firmware update mechanism. The device requires a bootloader, the logic that is executed first when the device boots and determines which firmware it launches. Sometimes devices are equipped with multiple bootloaders; for example, a stage 1 bootloader in the ROM and a stage 2 bootloader that can be updated. The reason for such designs is security-related as an update of the bootloader can lead to a bricked device. Whenever a bootloader is present on a device, the memory layout of the hardware has to be considered, for example the firmware must be linked in the correct position (with offset) for the device.

A typical firmware update requires a number of steps:

1. A developer recompiles the code and generates an entirely new firmware image, which is then distributed to the device.
2. The flash memory of the IoT device is split into memory regions (slots) containing (i) the bootloader and (ii) firmware images (with some metadata).
3. The new firmware is stored into one of the available slots.
4. The IoT device is then reset so that the bootloader can boot the new firmware image [15].

This approach is used, for example, by MCUboot [161] and ESPer [72]. On top of these steps, additional features can be added to reduce the size of the network transfer or increase the granularity of the software update. These options are not mutually exclusive and can be deployed together for additional gains.

PARTIAL UPDATES THROUGH DYNAMIC LOADING

One option to reduce the update size is by increasing the granularity of the targeted firmware binary. This allows for updating only part of the binary, instead of the full firmware binary. Multiple different approaches to increase the granularity of the firmware image exist.

One way to enable partial updates is via dynamic loading of binary modules [64, 140]. The firmware must support dynamic loading of parts of the firmware. This allows for targeting specific areas of the binary via the updates, targeting only the part where the update is required.

Another simpler option is using component-based programming [175, 176] aim to simplify dynamic modification and reconfigurability of the system on constrained IoT devices by enforcing black-box-style interactions between system modules.

Partial updates of software can also use scripts instead of binaries [28], whereby pieces of interpreted language are updatable on devices. Only the content of the scripts are updated and the firmware responsible for interpreting the scripts is not updated. Popular environments for this use JavaScript, WebAssembly or Python.

DIFFERENTIAL UPDATES

Another approach to reduce size of the firmware update is to use differential binary patching [98]. This allows for patching only part of the binary during an update, decreasing the size of the update. Compared to partial updates, this does not require separate modules in the binary, and the full firmware can be updated with a single update. A requirement for this is that the exact running firmware used as base for the differential update is known.

BINARY COMPRESSION

Lightweight compression schemes such as [27] can be used to apply binary compression to the firmware update [158]. This decreases the size of the transported binary and shifts part of the burden to the target device, as the binary needs to be decompressed before it can be written to the memory of the device.

2.5.2 UPDATE FRAMEWORK BACKEND

The second aspect of IoT firmware updates concerns the backend framework and securing the supply chain of IoT software. The Internet Engineering Task Force (IETF) Software Updates for Internet of Things (SUIT) working group specifies a simple back-end architecture [126] for IoT firmware updates. In addition to authentication and integrity protection, even when updates are stored on untrusted repositories, the SUIT specifications enable encrypting the firmware image, to protect against attacks based on reverse engineering. SUIT followed previous work such as FOSE [63] which proposed firmware encryption

and signing using JavaScript Object Notation (JSON) and Javascript Object Signing and Encryption (JOSE).

The Update Framework (TUF) [162] and Uptane [104], designed for use in connected cars, aim to ensure the security of a software update system, even against attackers who compromise the repository or signing keys. ASSURED [26] builds on TUF to improve support for constrained IoT devices by leveraging a trusted intermediate controller between the update repository and IoT device. CHAINIAC [127] is another approach that uses a blockchain-like mechanism to attest to the history of prior updates, even without central authority.

2.5.3 NETWORK TRANSPORT TO THE FIRMWARE TOWARDS THE IOT DEVICES

The third aspect of IoT firmware updates concerns the dissemination of software through the network. As mentioned earlier, the class of devices is often connected via a constrained type of connectivity to a network. The transport used to disseminate the updates must take this into account as not to put a too great burden on the network. The variety of approaches to this topic, as presented in recently published literature, includes protocols that optimize the dissemination of updates through multiple paths in a multi-hop, low-power wireless network [90]; updating network stack modules to reconfigure the network on the fly [176]; and using the Message Queuing Telemetry Transport (MQTT) protocol to disseminate software updates to a fleet of IoT devices [72]. 6LoWPAN protocols [148] enable end-to-end network connectivity from constrained IoT devices to anywhere on the internet. The IETF Trusted Execution Environment Provisioning (TEEP) working group [92] is standardizing a transport mechanism to update trusted applications running in trusted execution environments (TEEs), such as Arm TrustZone and Intel SGX.

2.5.4 OPEN STANDARDS FOR SECURE CONSTRAINED FIRMWARE UPDATES

When considering open standards applicable for implementing firmware updates in the constrained device space, multiple standards are available or in active development. Some of these, such as CoAP mentioned already, provide a transport for payloads suitable for constrained devices.

The SUIT specifications include an architecture document [126], an information model description [125], and a proposal for a manifest specification [7].

To achieve its goals, SUIT builds upon a number of other open standards that provide generic building blocks. In particular, the Concise Binary Object Representation (CBOR) [42] specification is used as a data format for serialization. CBOR is a schema-less format optimized for a small message size using a binary encoding. Furthermore, the CBOR Object Signing and Encryption (COSE) [143] specification is used to

cryptographically secure data serialized with **CBOR**. **COSE** defines a variety of structures, among them the `sign` structure, which specifies how to protect a payload against tampering by using a cryptographic signature.

When taking TUF/Uptane [104] as a reference, for instance, the **SUIT** manifest format could provide Uptane-compliant (custom) metadata about firmware images. The TUF standards neither target interoperability, nor specify concrete metadata formatting, contrary to the **SUIT** standards.

STANDARDS FOR IOT FIRMWARE TRANSPORT

A number of protocols provide specifications for transferring a firmware update over the network. Basic transport schemes enable a so-called Device Firmware Upgrade (**DFU**) over a specific low-power Media Access Control (**MAC**) layer technology, such as Bluetooth, or over a specific bus technology, such as USB. On the other hand, to transport firmware over several hops, or over heterogeneous low-power networks, the **IETF** suite of protocols standardized a network stack combining **CoAP** over UDP [147] and **CoAP** over TCP/TLS [148]. **CoAP** offers features equivalent to HTTP but tailored to constrained **IoT** devices. The 6LoWPAN specification was designed to offer an adaptation layer for networks that cannot directly use IPv6. To provide communication security, DTLS and TLS profiles [163] were standardized for use in **IoT** deployments.

FIRMWARE UPDATE METADATA

The firmware update requires extra information that describes instructions for the target device on how the new firmware must be installed on the device. The **IETF SUIT** working group is currently standardizing a format for describing firmware updates. The **SUIT** group defines a so-called manifest, which provides:

1. Information about the firmware required to update the device.
2. A security wrapper to protect the metadata end-to-end.

STANDARDS FOR REMOTE IOT DEVICE MANAGEMENT

One of the most prominent open standard for **IoT** device management is the Lightweight Machine-to-Machine (**LwM2M**) protocol [132, 133, 134] developed by OMA SpecWorks, a merger between the Open Mobile Alliance (OMA) and the IP Smart Object (IPSO) Alliance. To transfer data, **LwM2M** v1.1 uses **CoAP**, which can be secured with DTLS [163] or TLS [44]. The **LwM2M** specifications define a simple data model and several RESTful interfaces for remote management of **IoT** devices. The interfaces enable devices to register to a server, provide information updates, and obtain keying material. A large number of objects and resources have already been standardized to support commonly used sensors, actuators, and other resources. Among the standardized objects is the firmware update object.

The CoAP Management Interface (**CORECONF**) [166] is a more recent design and standardized by the **IETF**. **CORECONF** uses **CoAP** and a data model based on the Yet Another Next Generation (**YANG**) modeling language [40], and aims to reuse existing Simple Network Management Protocol (**SNMP**)-defined objects and resources. **CORECONF** is still in development, and a firmware update mechanism has not yet been defined. Such extension might however be defined in a future extension.

The Open Connectivity Foundation, the result of a merger between the UPnP Forum, the Open Interconnect Consortium (OIC), and the AllJoyn Alliance, standardizes an **IoT** device management protocol operating on top of **CoAP** and TLS/DTLS for communication, similarly to **LWM2M**. The OCF defines a data model with RESTful API Modeling Language (RAML) as the data modeling language. While initially targeting bigger **IoT** devices in smart home environments, the OCF is now also considering other industry verticals.

Earlier work on device management for **IoT** devices use remote procedure calls instead of a RESTful design. For instance TR 69 [48], also known as the CPE WAN Management Protocol (CWMP) developed by the Broadband Forum, formerly known as the DSL Forum, offers firmware update functionality on higher-end **IoT** devices, such as Internet-connected printers. The successor of TR 69, called User Services Platform (USP) [49], was recently released by the Broadband Forum.

2.6 AUTHENTICATION THROUGH DIGITAL SIGNATURES

Authenticating a specific payload for verification on the receiver side is possible via digital signatures. Providing this security guarantee for updates is mandatory as otherwise the authenticity of the firmware update cannot be guaranteed. The payload is signed with the private key stored and protected by the sender, and the receiver can verify that the message has not been tampered with the public key provisioned or received earlier. For this, the message, signature and public key are combined in a function that returns a result indicating a correct or tampered message.

Multiple different schemes exist providing digital signatures each with their own properties, including key sizes and signature sizes. One main difference between digital signature algorithms is whether the underlying problem they are based on is resistant against a quantum computer.

As with pre-quantum cryptography, post-quantum cryptography is designed to run on regular hardware. However, post-quantum cryptographic primitives are designed to resist adversaries with access to both classical and quantum computers. For pre-quantum cryptography, adversaries with access to quantum computers can leverage specific algorithms [78, 150] to break these cryptographic primitives. When considering post-quantum digital signatures, these signatures must

provide a similar security level, while providing resistance against attacks from both type of computers.

2.6.1 POST-QUANTUM SIGNATURE SCHEMES

Multiple different types of post-quantum digital signature schemes have been developed [149]. These can be classified based on their underlying hard problems that guarantee their security.

HASH-BASED SIGNATURES

Hash-based signatures are a form of post-quantum digital signatures based on the security of hash functions. They are among the oldest digital signature schemes available. The security of hash-based signatures relies on the difficulty of inverting cryptographic hash functions. Hash-based signatures in general provide very fast verification at the cost of very large signatures. One of the hash-based signatures has been standardized as Hierarchical Signature System (HSS)/ Leighton-Micali Signature (LMS) [118]. The main issue of hash-based signatures is the stateful nature of the private key, the private key must be update after every signature and can be used a limited number of times.

LATTICE-BASED SIGNATURES

The lattice-based signatures are based on hard problems in Euclidean lattices. In general, these schemes offer fast signing and verification as advantages, but in turn they generate very large signatures. Examples of Lattice-based signature schemes are the CRYSTALS-Dilithium [113], NTRUSign [89] and Falcon [71].

MULTIVARIATE-BASED SIGNATURES

Multivariate signature schemes are based on the complexity of solving certain low-degree polynomial systems in many variables. Analysis in the field of multivariate cryptography [39] questioned the security level of some of these signature schemes.

ISOGENY-BASED SIGNATURES

Isogeny-based cryptography is based on the difficulty of computing unknown isogenies between elliptic curves. Isogeny-based signature schemes inherit small parameter sizes from pre-quantum elliptic-curve cryptography, for example SQISign [60]. This property makes them interesting for microcontroller applications. On the other hand they also inherit and increased the computational burden caused by the heavy algebraic calculations of ECC.

CODE-BASED SIGNATURES

Code-based cryptosystems rely on the difficulty of hard problems of the theory of error-correcting codes. For example the McEliece key exchange scheme [117] is among the oldest of all public-key

cryptographic systems. Code-based signature schemes however, are much less well-established.

ZERO-KNOWLEDGE-BASED SIGNATURES

A relative new category of post-quantum digital signatures use zero-knowledge-based techniques. They combine algorithms from symmetric cryptography with technique known as Multi-Party Computation In The Head [96].

Table 2.2: Overview of post-quantum signature candidates. “Security analysis” reflects the maturity of analysis of the scheme: the age of the scheme, recent attacks, and how well-studied the underlying hard problem is, are considered.

Paradigm	Scheme	Security Analysis	Signature	Public Key	Private key
Hash	LMS	Mature	4756 B	60 B	64 B
	SPHINCS+-128	Mature	17 088 B	32 B	64 B
lattice	Dilithium	Less Mature	2528 B	1312 B	2420 B
	Falcon	Less Mature	1281 B	897 B	666 B
Multivariate	Rainbow I	Not Mature	66 B	157 800 B	101 200 B
	GeMSS	Not Mature	417 416 B	14 520 B	48 B
Isogeny	SQISign	Not Mature	204 B	64 B	16 B
Code	WAVE	Not Mature	1625 B	13 MB	—
Zero-knowledge	Picnic3-L1	Not Mature	13 802 B	34 B	17 B

2.6.2 PRE-QUANTUM ALGORITHMS

Multiple types of pre-quantum cryptographic digital signature schemes are available. For this comparison only digital signature schemes suitable for implementation on small microcontrollers are considered.

ELLIPTIC CURVE

Elliptic curve cryptography makes use of the algebraic structures of elliptic curves over finite fields. This provides relative small key and signature sizes with acceptable verification speed. Elliptic curves have been studied extensively with multiple different elliptic curves available for selection. In the field of IoT the P-256 curve [156] is often used, and available in implementations such as TinyCrypt [93, 114].

EDWARDS-CURVE DIGITAL SIGNATURES

EdDSA [35] is a digital signature scheme using a variant of Schnorr signatures based on twisted Edwards curves. The goal is to be faster than existing digital signature schemes while retaining the same level of security. Two signature schemes are defined for EdDSA:

- Ed25519, using Curve25519 with SHA-512 hash function.
- Ed448, using Curve448 with SHAKE256 hash function.

2.7 EMBEDDED SOFTWARE VIRTUALISATION AND SANDBOXING

A sandbox provides a controlled environment in which software can be executed in full isolation from the host system. The main purpose is to restrict access to critical operating resources while still being able to execute the software. This limited access provides a mechanism to run software without fully trusting the sandboxed application, which could be malicious or flawed. It provides a safe space to test and analyze software without putting the system at risk. One possible approach to sandboxing are VMs.

The vast majority of prior work on lightweight virtualisation run-times [123] does not target microcontrollers, but microprocessor-class computers. Recent examples include for instance AWS Firecracker [18] for serverless computing, WebAssembly [82] for process isolation in Web browsers, or eBPF [69, 116] for debug and inspection code inserted in the Linux kernel at run-time.

Roughly two types of sandbox environments can be distinguished that each provide a sandbox environment to the host system. The first is the script environment, with the second one being the VM.

2.7.1 SCRIPT ENVIRONMENTS

Script environments on microcontrollers allow for interpreting and execution of scripted applications. The applications are written by a developer and directly loaded on the microcontroller. An optional in-between step is minification of the script, where size and execution overhead caused by optional tokens in the script is removed.

PYTHON

One popular language running on microcontrollers is Python. For example MicroPython [77] is a very popular scripted logic interpreter used on microcontrollers. Another alternative is CircuitPython [16] runtime. Both options are geared towards hobbyists and aim for an easy and rapid development flow, while abstracting away the specifics of the microcontroller used.

Small Python runtimes are used on ESP8266 microcontrollers in prior work such as NanoLambda [76]. This runtime provides a scheduler to intelligently place functions across multi-scale IoT deployments according to resource availability and power constraints.

JAVASCRIPT

Multiple implementations of JavaScript environments are also available for constrained devices [79]. One advantage of JavaScript is the expressiveness of the language, trading a decrease in script size for complexity in the interpreter.

Another advantage of original design of JavaScript, as scripting language in browsers, is the event-based nature. This makes it a good fit for power-efficient microcontrollers where tasks can run on-demand when an event needs to be processed. The rest of the time the microcontroller can be reduced to a power-efficient sleep mode.

Multiple engines for executing JavaScript on microcontrollers are available. Usually these engines require additional code, written in the host language to access functionality from the hardware and host platform.

JerryScript [75] is one of these engines. It was initially started by Samsung in 2014, but now supervised and sponsored by the OpenJS Foundation. The JerryScript engine is provided as a library and integration into a project is required before it can be used. One of such project, provides an Over-the-air updatable environment on top of the RIOT operating system [29]. However, complementary mechanisms should be used to guarantee mutual isolation between scripts (such as SecureJS [103]).

A similar project is Duktape [165], an embeddable JavaScript engine, with a focus on portability and compact footprint. The engine also requires a platform or operating system to host it, but it requires only minimal functionality from the host. Duktape is fully compliant with ECMAScript version 5 and partly version 6.

mJS [54] is another embeddable engine for JavaScript. Where other interpreters try to support the full ECMAScript standard, mJS intentionally does not implement the full language. The main advantage is a reduction in required flash and RAM, only 50 kB of flash and 1 kB of RAM is required to host the engine. mJS is mainly used in the Mongoose OS framework [55] for IoT systems.

Espruino [170] is a stand-alone JavaScript interpreter with a very active community and actively developed ecosystem. When used on a microcontroller, applications can be written entirely in JavaScript without requiring any C or C++ knowledge.

A different approach is used by the Moddable SDK [88, 121]. Instead of running a JavaScript interpreter on the device, the SDK compiles the JavaScript into small optimized bytecode. The resulting bytecode can be flashed on the device and executed by the bytecode interpreter from Moddable.

LUA

A Lua code interpreter is sufficiently small to embed on microcontrollers. Prior work applied Lua as a scripting environment to support dynamic orchestration of multiple networked microcontrollers [80]. Another approach leverages a Lua VM for partial updates through LoRaWAN-connected devices [130].

2.7.2 VIRTUAL MACHINES

Virtual machines are ubiquitous on large server deployments and are used to isolate different tenants and systems from each other. This allows large cloud providers to rent out compute resources without risk. Server hardware includes hardware support for virtual machines (VMs) operations to allow for minimal overhead when virtualising software.

On constrained embedded microcontrollers, specialized instructions and support for virtual machines is not available. As mentioned before, even memory isolation can be challenging. The environment in which the VM is used, provides less isolation by default to the VM runtime and protection of the host system must fully happen from the VM runtime. The limited resources provided by the class of devices puts additional strain on the resources provided to the VM.

JAVA VIRTUAL MACHINES

Java VMs execute Java bytecode. A large number of Java Virtual Machines [25, 50, 67, 107, 137, 146] are available for constrained devices.

CapeVM [137] is a Java VM implementation offering a full sensor node abstraction. The VM bytecode is compressed to reduce transmission costs and to save power. The bytecode itself is compiled ahead of time to native instructions.

Darjeeling [50] uses an adapted Java VM modified to use a 16 bit architecture. It is designed for 8- and 16-bit processors for use on sensor nodes. Only a subset of the Java VM is implemented to reduce code complexity and memory usage.

A very specific application of Java VMs is Java Card. It allows for running Java-based applications securely on smart cards and other cryptographic tokens. Choupi OS is an MPU-hardened Java Card Virtual Machine [46]. The MPU-hardened Java Card Virtual Machine project aims to secure JavaCard applets running on shared hardware by utilizing the MPU hardware. The MPU safeguards executing contexts, each hosting a Java Card byte code interpreter, confining each applet within its dedicated context and interpreter. The MPU configuration is established at compile-time and dynamically reconfigured to align with the executing context.

WEBASSEMBLY

WebAssembly (Wasm) [82] is a virtual instruction set architecture developed and standardized by the World Wide Web Consortium (W3C) and primarily aimed at portable web applications. The instruction set allows for binaries small in size, to minimize transfer time to the client. The sandbox provided by implementations offers strong guarantees on memory access. Both of these properties aim to ensure security while requiring only limited memory footprint on the platform target.

The WebAssembly VM specifies the use of both a stack and a flat heap for memory storage. The stack is required by the architecture, and can

be configured to any size. An interface for allocating heap memory is provided by the standard. Note that the specification mandates memory allocations in chunks of 64 KiB (pages) which is not always possible on smaller IoT platforms.

TOOLCHAIN & SDK A software development kit is available to write applications for WebAssembly. The full workflow for development and executions of applications is depicted in Figure 2.1. Wasm uses the LLVM compiler: it is thus possible to write applications in any language supported by LLVM, such as C/C++, D, Rust, and TinyGo among others. Note that for C and C++, the WebAssembly binaries are created using the emcc toolchain, which combines the EmSDK with LLVM. A standardized interface is specified for host access in a POSIX-like way is provided by the WebAssembly System Interface (WASI) standard [167]. It offers standardized access to operating system facilities such as files, network sockets, clocks and random number.

INTERPRETER Once the Wasm binary is created, it can be transferred to the IoT device and on which it can be interpreted and executed, as shown in Figure 2.1. Several interpreters exist, for example, minimized WebAssembly runtimes adapted to run on 32-bit microcontrollers were proposed, such as WAMR [52] and WASM3 [151].

WASM3 uses a two-stage approach, whereby the loaded application is first transpiled to an efficient and optimized executable form which then can be executed in the interpreter. WAMR has multiple compile-time tunables to configure which execution strategy must be used. Both a Just-in-Time (JIT), where the WebAssembly bytecode is optimized to native execution, and an interpreted execution strategy are available.

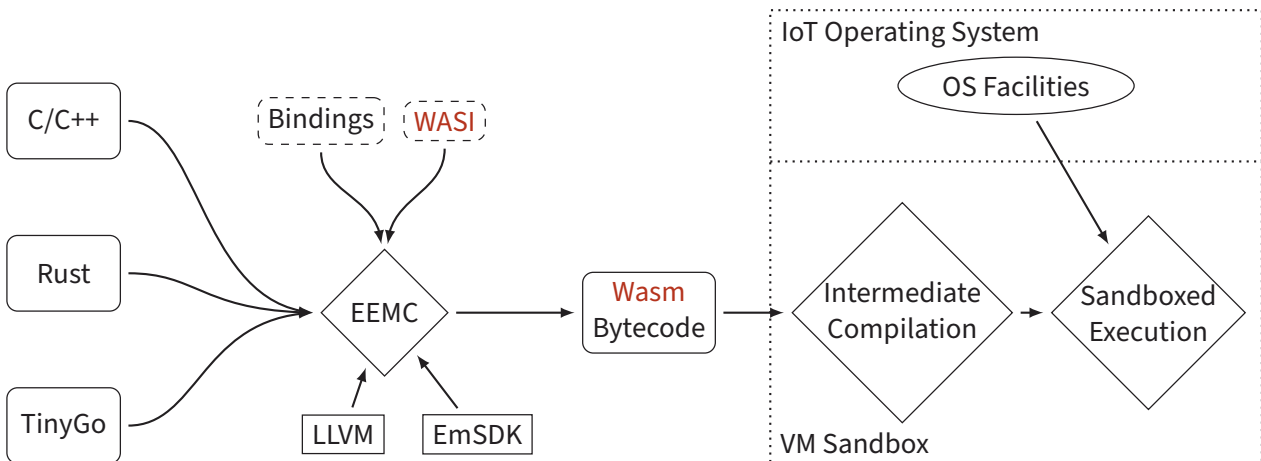


Figure 2.1: Wasm code development and execution workflow with the WASM3 VM.

EXTENDED BERKELEY PACKET FILTERS

Berkeley Packet Filter (BPF) [69] is a small in-kernel VM specifically for sandboxing small applications transferred from user-space into the kernel. Originally its purpose was for filtering network packets, for example: only passing packets to user space matching a set

of requirements. However, some ultra-lightweight virtualisation approaches have been proposed for microcontrollers. RapidPatch [87] uses an eBPF runtime to provide a hot patching framework for RTOS firmwares. Within the Linux kernel the VM is extended and renamed to eBPF to allow for various purposes not necessarily related to networking. It provided a small and efficient facility for running custom code inside the kernel, hooking into various subsystems.

The state-of-the-art eBPF architecture is 64-bit register based VM with a fixed stack. 10 general purpose registers are available together with a read-only stack pointer. The stack itself is specified as fixed at 512 B. A heap is not contained in the specification, as an alternative the Linux kernel provides an interface to key-value maps as persistent storage between invocations. These maps can also be interfaced with from user space applications. The limited stack size and absence of a heap put only minimal requirements on the RAM a platform has to provide for the VM.

The VM is suitable for isolating the operating system from the virtualised application: all memory access, including to the stack, happen via load and store instructions. Moreover, branch and jump instructions are also limited, the application has no access to the program counter and a jump is always relative to current program counter. The VM does not provide facilities to directly influence the program counter. Both of these can be implemented with the necessary checks in place to limit access and execution.

Interfacing with the operating system facilities can be done by providing the necessary bindings on the device. These can then be accessed during execution from the VM.

CHAPTER 3

COMPARATIVE EVALUATION OF POST- AND PRE-QUANTUM DIGITAL SIGNATURES FOR CONSTRAINED DEVICES

Any communication with networked devices for delivering adjustments to the device must at least be authenticated with digital signatures. This is required to prevent unauthorized parties from modifying the communication to the device and in turn the behaviour of the device. Without any proof of origin, the device will not be able to verify the origin of the request.

In the space of constrained IoT devices a requirement for secure authenticated communication is also present. Protecting the communication with digital signatures allows the device to verify the origin of the communication. Without the option to communicate securely with constrained embedded devices, devices cannot be securely updated and can quickly become a liability when they become part of a botnet.

With the device lifetime of small embedded constrained devices measured in years, protecting communication of these devices against future development is a must. Post quantum signatures are a strict requirement with respect to the future of quantum computing. This allows for secure communication with these devices even when pre-quantum cryptography is no longer an option.

In this chapter I present benchmarks on both post-quantum and pre-quantum digital signatures and show the trade-offs required when deploying one of these primitives on the devices. This collaborative work has been presented previously in “Quantum-Resistant Software Update Security on Low-Power Networked Embedded Devices” [4].

3.1 IMPACT OF CRYPTOGRAPHIC PRIMITIVES ON FIRMWARE

Adding cryptographic digital signature primitives to constrained devices is not without cost and careful consideration is required with

the type and implementation of the library. Cryptographic primitives implemented for these devices must be written with the resource constrained nature of the devices in mind. This implies that the implementation must not require large amounts of memory and must not put a too large computational burden on the processor. Often a trade-off must be made between the cryptographic strength of the primitive and the resources required for the operation. Different optimizations are possible for implementations and with unconstrained devices usually memory is traded in favour of a decrease in computational cycles. These type of optimizations are not always possible on memory constrained devices.

Furthermore, other considerations are possible with cryptographic libraries. For example, a generic big number library can be reused by multiple cryptographic primitives implemented by a single library, or the library implements the operation as a specialized function. When only a single primitive is required on the device, a big number library will require more memory on the device.

The extra computational load required by some implementations does not only increase the response time of the devices. Extra cycles spent on cryptographic operations also increases the overall power consumption of the device as the device is active for longer durations.

Another concern is the size of the signature generated by the primitive. Depending on the algorithm used, signatures can grow to sizes that dwarf the size of the payload protected. This in turn can put significant strain on the network connectivity of the device. Furthermore, the device must hold the full signature in memory after the transfer to verify the origin of the message.

3.1.1 FIRMWARE UPDATE SIZES AND POST-QUANTUM SIGNATURES

When considering the impact of post-quantum signatures and their impact on firmware updates, multiple aspects of the signature algorithm must be considered. On the embedded devices, only signature verification is relevant as that is the only operation required for the verification of the firmware binary. Based on the estimates from [Table 2.1](#), four broad categories of updates for low-power embedded IoT can be distinguished

1. Small software module update, of ≈ 5 kB.
2. Small firmware update without cryptographic libraries, ≈ 50 kB.
3. Small firmware update including cryptographic libraries, ≈ 50 kB.
4. Large firmware update including cryptographic libraries, ≈ 250 kB.

When considering the update protocol itself, a manifest based on the SUIT specification can have a real world size of 419 B, excluding the signature.

3.2 CRYPTOGRAPHIC LIBRARY SELECTION

The cryptographic libraries compared here implement one of the existing digital signature schemes. Both post-quantum and pre-quantum digital signatures are compared, where the pre-quantum digital signatures are used as baseline for the comparison. Especially the post-quantum cryptographic implementations must be selected based on the following properties:

- *Key size*: The public key must fit in the memory of the microcontroller.
- *Signature size*: The signature size puts a burden both on the memory of the microcontroller and the network transfer size.
- *Runtime performance*: As mentioned before, computational burden of the signature verification in turn influences the battery life and responsiveness of the microcontroller application.
- *Maturity*: The post-quantum digital signature schemes have varying levels of analysis and the security level of some of the algorithms is still subject of debate.

The **NIST** PQC project has dominated research in the post-quantum cryptography in recent years, with multiple digital signature algorithms submitted. The candidates resulting from this project are a natural first selection for credible post-quantum digital signature algorithms, as these have had extensive analysis from the cryptographic community. However also older post-quantum schemes can be considered such as the hash-based signature schemes.

3.2.1 PRE-QUANTUM SIGNATURE SCHEMES

For the baseline measurements to compare post-quantum digital signature schemes, a number of pre-quantum digital signature schemes and implementations have been selected.

ECDSA IMPLEMENTATIONS

The ECDSA implementation used in this work is the TinyCrypt library. This library is designed by Intel to provide cryptographic standards for constrained devices, including the **NIST** standard P-256 curve.

ED25519 IMPLEMENTATIONS

For Ed25519, two libraries are used, both providing constant-time finite-field arithmetic based on public-domain implementations.

- *C25519*: A very small public domain implementation of the Ed25519 digital signature scheme and the x25519 key exchange.
- *Monocypher*: Another implementation of cryptographic primitives including Ed25519 digital signatures. The main difference

with C25519 is the use of precomputed tables in Monocypher to speed up the computation of elliptic curve points.

3.2.2 POST-QUANTUM SIGNATURE SCHEMES

While a large number of post-quantum signature schemes are available as shown in subsection 2.6.1, a subset of these are considered for the evaluation performed in this chapter. When choosing candidate signature schemes, key and signature sizes, runtime performance, and maturity with respect to security analysis must all be considered. While the relatively compact parameters of some isogeny- and code-based signature schemes may make them interesting for future work targeting microcontrollers, at present these schemes are far from theoretical maturity. The true security level of the NIST multivariate and ZK-based candidates is a subject of current debate, though their extremely large keys and/or signatures would likely eliminate them from consideration for constrained embedded devices. The NIST PQC project has dominated research in post-quantum cryptography in recent years. Its candidate cryptosystems are a natural first port of call for credible post-quantum signature algorithms, since they have had the benefit of concerted analysis from the cryptographic community — especially the Round 3 proposals, which are candidates for standardization in the coming years.

However, these are not the only algorithms that should be considered. For example, among hash-based signature schemes, a comparison between the older LMS scheme, which is not a NIST candidate, with the newer SPHINCS+ scheme, which is a NIST Round 3 alternate. LMS has smaller computational requirements, but the signer must maintain some state between signatures; SPHINCS+ is a heavier scheme, but it is stateless. Statelessness is an advantage for general applications, as tracking the key state increases complexity on the usage of the signature scheme. However around the use case of firmware updates, statefulness is natural, as it corresponds naturally to the version number on the firmware update. As the constrained embedded device only requires signature verification, tracking state of the key is not relevant on there, so the lighter LMS is a more natural choice.

For the reasons above, the focus of the benchmarks is on three post-quantum signature algorithms: LMS, Dilithium, and Falcon, representing the hash-based and lattice-based categories. LMS has 60B public keys and 4756-byte signatures. Dilithium II, targeting NIST security level 2, has 1312B public keys and 2420B signatures. Falcon-512, targeting NIST security level 1, has 897B public keys and 666B signatures.

LMS

For LMS, the Cisco implementation [58] is used with a small modification, removing calls to malloc since it can lead to memory fragmentation. This change might lead to some small improvements in performance, since the kernel already knows the address at compile-

Table 3.1: Public key sizes of selected signatures

Algorithm	Public Key Size
LMS	60 B
Dilithium II	1312 B
Falcon	897 B
Ed25519	32 B
ECDSA P-256	32 B

Table 3.2: Signature sizes of selected signatures

Algorithm	Signature Size
LMS	4756 B
Dilithium II	2528 B
Falcon	1281 B
Ed25519	64 B
ECDSA P-256	64 B

time rather than only at runtime. For the benchmark, the smallest parameters proposed in [118, Section 5] is used: that is, SHA-2 with 256-bit output for the hash function (to keep the code as close as possible to the original implementation [58]) with tree height 5, and 32 bytes associated with each node. For the LMOTS, 32 bytes and 4 bits of width for Winternitz coefficients is used. The OpenSSL call from the original code is removed and change for a implementation of SHA2-256 provided in their repository [58]. Furthermore, HSS is used with 2 layers. These parameters satisfy the life cycle of updates: in particular, the key lifetime will never be surpassed by the amount of updates.

FALCON

The Falcon implementation provided by PQClean [101, 102] is used without any significant structural modifications.

DILITHIUM

Two Dilithium implementations based on PQClean are prepared [101, 102].

- *Dynamic Dilithium* is the basic PQClean implementation. The first step in signing and verifying is to expand a random seed given in the public key into a large matrix.
- *Static Dilithium* modifies the PQClean implementation to pre-compute the matrix and store it in the flash memory of the microcontroller. This makes both signing and verification faster, at the cost of using more flash and reducing flexibility. Only the flashed key can be used to verify signatures against.

3.2.3 HASH FUNCTIONS

One aspect of digital signature algorithms is that they often require a digest function to operate on. Ed25519 uses the SHA-512 digest function, ECDSA has a configurable digest function. Furthermore, hash functions are often used as image and signed instead of signing the actual payload.

When considering post-quantum attacks on digest functions, the outlook is positive. Only a few quantum attacks against SHA-2 and SHA-3 exist. Grover's algorithm may be parallelized to find hash pre-images [31]. This applies to both Merkle Damgård hashes (SHA-2) and Sponge-based hashes (SHA-3). For collision resistance, the state-of-the-art in quantum collision search does not drastically reduce the complexity with respect to classical algorithm [56]. However, classical attacks for SHA-2 might become a reality [62] at some point, still causing a decrease in security with SHA-2.

When considering the hash functions in the scope of low power embedded microcontrollers, the main concern is the memory usage and run time of the hash function implementation. One aspect to keep in mind while selecting a hash function is a synergy with the selected

signature algorithm. Most post-quantum signature algorithms use SHA-3 in their construction, as candidates for the **NIST** post-quantum signature standard are required to be SHA-3/SHAKE compatible, the current US standard. As space on the microcontroller devices is very limited, factorization of code is typically desirable. Selecting a single hash function for both hashing for images and as hash function used by the digital signature algorithms is desirable. When using post-quantum signature algorithms, matching these with SHA-3 hash functions used in other modules on the firmware decreases the overall flash footprint.

3.3 BENCHMARKS

3.3.1 BENCHMARK HARDWARE SETUP

The testbed used for the benchmarks here consists of popular, commercial, off-the-shelf hardware. The boards selected are representative of the modern 32-bit microcontroller architectures available. For the memory footprint of the libraries, only the total flash usage of the library itself was measured. For the timings, the running time of the operation and the number of kiloticks, based on the hardware clock and the time spent, was measured.

- **ARM Cortex-M4:** The Nordic nRF52840 Development Kit. This board provides a typical ARM Cortex-M4 microcontroller running at 64 MHz, with 256 KiB of RAM, 1 MiB flash and a 2.4 GHz radio transceiver compatible with both IEEE 802.15.4 and Bluetooth Low-Energy.
- **Espressif ESP32:** The WROOM-32 board. This is a small development board containing the ESP32 module with the ESP32-D0WDQ6 chip on board. It provides two low-power Xtensa 32-bit LX6 microprocessors with integrated Wi-Fi and Bluetooth, operating at 80 MHz, with 520 KiB of RAM, 448 KiB of ROM.
- **RISC-V:** The Sipeed Longan Nano GD32VF103CBT6 Development Board. This provides a RISC-V 32-bit core running at 72 MHz, with 32 KiB of RAM, 128 KiB of ROM and no wireless connectivity.

RIOT is used as a base firmware setup for these benchmarks, providing hardware abstraction and timers to benchmark the implementations.

3.3.2 PRE-QUANTUM SIGNATURE BASELINE

The pre-quantum digital signature algorithms act as a baseline metric to compare the post-quantum algorithm against.

MEMORY FOOTPRINT

In figure **Table 3.3** the flash usage of the pre-quantum digital signature algorithms is presented. As visible, the Monocypher implementation requires the most flash on all platforms, with TinyCrypt in the middle and C25519 requiring the least amount of flash.

	Algorithm	Cortex-M4	ESP32	RISC-V
C25519	Ed25519	5106 B	5608 B	6024 B
Monocypher	Ed25519	13 852 B	17 238 B	17 328 B
TinyCrypt	ECDSA P-256	6498 B	6869 B	7452 B

Table 3.3: Flash usage for pre-quantum digital signature algorithms.

SIGNATURE GENERATION SPEED

The signature generation speed is shown in [Table 3.4](#). Timings are shown both in milliseconds and in kiloticks for every platform. Looking at the signature generation speed, Monocypher is the fastest implementation on all platforms. C25519 is the slowest implementation here, with TinyCrypt again in the middle.

Table 3.4: Signature generation speed for pre-quantum digital signature algorithms, measured is the time in milliseconds and the number of clock ticks for signature generation.

	Algorithm	Cortex-M4		ESP32		RISC-V	
C25519	Ed25519	845 ms	54 111 kT	921 ms	73 690 kT	956 ms	68 883 kT
Monocypher	Ed25519	17 ms	1136 kT	21 ms	1709 kT	16 ms	1194 kT
TinyCrypt	ECDSA P-256	294 ms	18 871 kT	333 ms	26 696 kT	270 ms	19 489 kT

VERIFICATION SPEED

The verification speed numbers in general show a similar picture as the signature speed numbers, as visible in [Table 3.5](#). The verification speed is shown again both in milliseconds and in kiloticks for every platform benchmarked. Both algorithms need more time for the verification, with TinyCrypt only slightly slower on verification, but the Ed25519 verification is a bit over twice as slow.

Table 3.5: Signature verification speed for pre-quantum digital signature algorithms.

	Algorithm	Cortex-M4		ESP32		RISC-V	
C25519	Ed25519	1953 ms	125 012 kT	2165 ms	173 205 kT	2242 ms	161 475 kT
Monocypher	Ed25519	40 ms	2599 kT	60 ms	4864 kT	41 ms	3013 kT
TinyCrypt	ECDSA P-256	313 ms	20 037 kT	374 ms	29 948 kT	308 ms	22 192 kT

Looking at the results in general, the measurements show a large difference between the C25519 library and the Monocypher implementation, even though these implement the same signature algorithm. It clearly shows that the C25519 library is optimized for low memory usage, where the Monocypher library is build around optimizations to decrease the algorithm speed. The TinyCrypt library implements a different signature algorithm and strikes a middle ground in memory usage and algorithm speed.

3.3.3 POST QUANTUM CRYPTOGRAPHY PRIMITIVES

The selected Post-quantum digital signature schemes are deployed on the same hardware as the pre-quantum algorithms. As the RISC-V

board selected for the benchmarks is too limited in flash storage, the benchmark application for Dilithium in dynamic mode is unable to fit on the flash of the microcontroller and could not be benchmarked on this architecture.

MEMORY FOOTPRINT

The flash memory footprints of the post-quantum digital signature algorithms is shown in Table 3.6. As visible all algorithms require more than 10 KiB to fit on the devices.

	Cortex-M4	ESP32	RISC-V
Falcon	57 613 B	60 358 B	11 122 B
Dilithium-dynamic	11 664 B	12 397 B	—
Dilithium-static	26 672 B	27 197 B	25 148 B
LMS	12 864 B	15 177 B	15 889 B

Table 3.6: Flash usage for post-quantum digital signature algorithms.

SIGNATURE GENERATION SPEED

The signature generation of the post-quantum algorithms is shown in Table 3.7. LMS is clearly visible as outlier here, requiring multiple seconds to generate the signature on all platforms. Dilithium in both modes is relative fast among the post-quantum algorithms.

Table 3.7: Signature generation speed for post-quantum digital signature algorithms.

	Cortex-M4		ESP32		RISC-V	
Falcon	1172 ms	75 020 kT	1172 ms	93 824 kT	—	—
Dilithium-dynamic	465 ms	29 788 kT	87 ms	7036 kT	—	—
Dilithium-static	135 ms	8655 kT	121 ms	9694 kT	—	—
LMS	9224 ms	590 354 kT	7583 ms	606 674 kT	9105 ms	655 614 kT

VERIFICATION SPEED

The verification speed of the post-quantum digital signatures is shown in Table 3.8. All signature schemes are fast to verify, with only LMS requiring more than 100 ms on the different platforms.

Table 3.8: Signature verification speed for post-quantum digital signature algorithms.

	Cortex-M4		ESP32		RISC-V	
Falcon	15 ms	1004 kT	16 ms	1322 kT	13 ms	975 kT
Dilithium-dynamic	53 ms	3407 kT	43 ms	3508 kT	—	—
Dilithium-static	23 ms	1510 kT	21 ms	1706 kT	17 ms	1237 kT
LMS	123 ms	7908 kT	101 ms	8141 kT	122 ms	8808 kT

3.3.4 HASH FUNCTION BENCHMARKS

Table 3.9 compares three hash function implementations on the memory usage and speed on an ARM Cortex M4 microcontroller:

- RIOT's default implementation of SHA2-256.
- A compact implementation of SHA3-256 optimized to minimize flash memory footprint.
- An implementation of SHA3-256 optimized for speed on Cortex-M4 ARMv7M architectures.

Stack is roughly equivalent across the different implementations, but speed and flash vary widely: SHA3-256 can offer slightly faster execution than SHA2-256, but at the price of a 10× larger flash footprint. For a flash footprint similar to SHA2-256, the comparative speed of SHA3-256 diminishes drastically for larger inputs.

Table 3.9: SHA-2 and SHA-3 performance on the nRF52840 ARM Cortex-M4 microcontrollers

	Flash Usage	Stack Usage	64 B	Ticks to hash input		
				100 B	1024 B	10 240 B
SHA2-256 (RIOT)	1008 B	384 B	277 kT	278 kT	1943 kT	17 933 kT
SHA3-256 Compact	1692 B	404 B	1336 kT	1342 kT	10 402 kT	98 448 kT
SHA3-256 fast-ARMv7M	11 548 B	284 B	220 kT	228 kT	1672 kT	15 732 kT

3.4 IMPACT OF POST-QUANTUM PRIMITIVES ON EMBEDDED DEVICES

3.4.1 THE COST OF POST-QUANTUM SECURITY

A toe-to-toe comparison between pre-quantum and post-quantum signatures must consider all of the public key and signature sizes, running time, and memory requirements. All post-quantum algorithms have significant larger public key and signature sizes, by well over an order of magnitude. Compared with standard elliptic-curve signature schemes, Falcon's public keys are 28× larger and its signatures are 10.4× larger; Dilithium's public keys are 41× larger than elliptic-curve keys, and its signatures are 38× larger. **LMS** avoids this spectacular growth in public key sizes, with keys only 1.875× larger than elliptic-curve public keys; but its signatures are a massive 74.3× larger than elliptic-curve signatures.

When comparing the running time of the signature primitives, the post-quantum signatures have their advantages and disadvantages. Signature verification is generally considerably faster across all the devices tested, signing is generally slower however. The comparison of signing algorithms shown in [Table 3.7](#) and [Table 3.4](#) shows that the fastest post-quantum algorithm runs in 135 ms which is 7.94× slower than the Monocypher Ed25519 implementation. However, the reverse is true when comparing the algorithms on signature verification. The fastest pre-quantum algorithm runs in 40 ms, which is 2.65× slower than post-quantum Falcon. Efficient verification is a required and

valuable feature for constrained embedded devices, however in this setting it comes at the price of an increase in stack and flash memory.

3.4.2 THE COST OF POST-QUANTUM ALGORITHMS WITH FIRMWARE UPDATES

Considering a real world firmware updates using these post-quantum digital signature algorithms, the impact of changing the pre-quantum digital signatures to post-quantum alternatives can be measured. In practical terms the SUIT manifest as proposed in [subsection 3.1.1](#) being 419 B large without the signature, increases in size when switching from pre-quantum signatures to post-quantum.

- Falcon: $419 \text{ B} + 666 \text{ B} = 1085 \text{ B}$, a $\approx 2.24\times$ increase;
- Dilithium: $419 \text{ B} + 2420 \text{ B} = 2839 \text{ B}$, $\approx 5.87\times$ increase; and
- LMS: $419 \text{ B} + 4756 \text{ B} = 5175 \text{ B}$, a $\approx 9.84\times$ increase.

Now consider the crucial aspect of network transfer costs, and the memory resources required to actually apply the firmware update on the IoT device. s our measurements to evaluate the relative cost of the entire SUIT software update process. Visible is that the impact of switching to quantum-resistant security in SUIT varies widely in terms of network transfer costs, ranging from negligible increase ($\approx 1\%$) to major impact ($3\times$ more), depending on the software update use case.

Algorithm	Flash	Stack	Data Transfer	
			2	3
Ed25519 / SHA2-256	52.4 kB	16.3 kB	47 kB	53 kB
Falcon / SHA3-256	+120 %	+18 %	+1.1 %	120 %
LMS / SHA3-256	+34 %	+1.2 %	+9 %	43 %
Dilithium / SHA3-256	+30 %	+3407 %	+4.3 %	34 %

Table 3.10: Relative costs for SUIT with quantum resistance on the ARM Cortex M4

There are many possible deployments of IoT, and several possible scenarios for IoT software updates. It is safe to assume that the authorized maintainer, responsible for updating the firmware, has powerful hardware. Hence, the computational burden of signing is not the main concern here. On the other hand, a constrained device will be responsible for signature verification of the update.

As seen above, the cryptography package does not run standalone in the board: it must coexist with several other modules (including kernel, network stack, and libraries), and the application itself. One challenge that was faced in deploying the schemes was sharing stack memory (and RAM memory). For example, on the RISC-V platform used, the total RAM memory budget available was only 32 kB for the whole system, a small but not uncommon amount of RAM for this class of devices. It was not possible to run Dilithium to sign or verify within these constraints as the library required more stack than available. In fact the default stack configuration required adaptation for all of the post-quantum algorithms used.

Execution speed is another challenge. Slow signature verification may impact real-time applications if special care is not taken. Typically, on low-power IoT devices, there is no parallel computing. For instance, RIOT OS uses a preemptive multithreading paradigm, where a single thread is running at any given time. If signature verification takes a long time, running in a high-priority thread, then the system blocks on this task until completion. It is therefore necessary to carefully tune the priority of the crypto verification thread so as not to stop other functionally essential tasks, especially if signature verification is slow.

3.4.3 REAL-WORLD USABILITY OF POST-QUANTUM DIGITAL SIGNATURES

When considering the four firmware updates from subsection 3.1.1 (as shown in Table 3.11) and the choice of post-quantum digital signature for each.

In the case of option 1, a small module update of 5 kB, and option 2, the small firmware update without the cryptographic library, the package contains the software update and the signature. In this case speed and signature size are the most important factors. In these cases, Falcon has a large advantage over LMS and Dilithium.

When option 3 is considered, the firmware includes the cryptographic library, the situation is more complicated. Both the signature size, but also the flash impact of the cryptographic library must be considered. Both of these are transferred over the network when updating the firmware. As the size of the network transfer also impacts the duration of the firmware update, a small difference in verification speed is quickly dwarfed by a large increase in network transfer time due to more flash usage. As shown in Table 3.10, when these factors are considered, LMS presents itself as the best trade-off between flash size, network transfer cost, verification time and stack usage.

In the case of option 4, the large network transfer costs overwhelm the other costs, reducing the comparative advantages of one post-quantum signature over another. From the point of view of cryptographic maturity, LMS is the safest choice. As noted in subsection 2.6.1, hash-based problems have received extensive cryptanalysis from the cryptographic community, while the security of structured lattice-based schemes like Falcon is less well-understood. Nevertheless, compared to the pre-quantum state of the art, LMS imposes a significant increase in signature size and running time, which has a major impact on firmware update performance. Thus, despite its relative lack of maturity, the performance characteristics of Falcon make it extremely tempting for applications with smaller updates.

Table 3.11: Firmware sizes of the different update options

	Firmware	Size
1	small module	5 kB
2	Firmware	50 kB
3	Firmware + crypto	50 kB
4	Large Firmware	250 kB

3.5 DISCUSSION

3.5.1 COMPARISON TO PRE-QUANTUM DIGITAL SIGNATURES

Comparing the post-quantum digital signatures currently considered, there exist a large differences with current state-of-the-art pre-quantum digital signature schemes. When comparing the two different categories, public key sizes and signature sizes together with the memory requirements and running time must be considered. Current post-quantum digital signature schemes all have larger public key and signature sizes, generally by well over an order of magnitude. Comparing current elliptic-curve signature schemes, Falcon's public keys are 28 times larger and signatures are 10 times larger. Dilithium's public keys are 41 times as large as elliptic curve public keys and signatures are 38 times larger. **LMS** avoids the growth in public key sizes, with a slightly below 2 times increase in public key size, however the signatures are 74 times larger than current elliptic curve sizes.

When considering the computational time of the post-quantum digital signatures, the comparison is more mixed. Signature verification is generally relative fast compared to the elliptic curve signatures on the tested platforms. The fastest post-quantum algorithm, Falcon, requires 13 ms to 16 ms on the tested platforms, where the fastest pre-quantum signature requires between 40 ms to 60 ms. However, the fast signature verification of Falcon comes at the cost of requiring significant flash and moderate amount of stack for the implementation to work. Signature generation is in general slower with post-quantum signatures compared to the pre-quantum signatures. The fastest post-quantum algorithm sits between 121 ms to 135 ms, which is almost 8 times slower than Monocypher at 16 ms to 21 ms. Considering the memory requirements, the post-quantum digital signature schemes require significantly more flash and stack memory. Even the stack usage can grow over 11 times compared to the pre-quantum implementations.

3.5.2 IMPACT ON REAL WORLD SCENARIOS

Considering real-world scenarios where a payload must be authenticated on a constrained device, the public key must be provisioned on the device. While considerably larger than pre-quantum algorithms, all post-quantum algorithms have public key sizes that can be accommodated on these devices.

A larger issue is the signature size of the algorithm as this has to be transferred to the device over a constrained and lossy link. Furthermore the signature must fit in the memory of the target device to verify it. Depending on the exact deployment scenario in which the digital signature is required, the extra cost of the transfer of the post-quantum signature can be dominant compared to the actual protected payload.

3.6 CONCLUSION

This chapter provided an experimental study of available post-quantum digital signatures and the cost of transitioning from pre-quantum signatures to post-quantum signatures on constrained embedded devices. I compared the performance of standard pre-quantum cryptography to selected post-quantum digital signatures in the same constrained environment on three low-power IoT platforms, representative of the current landscape of 32 bit microcontrollers. I show that it is possible indeed to upgrade from classical 128 bit security to **NIST** Level 1 post-quantum security on these platforms. However I show that, based on the measurements, both in memory requirements and computational burden, the performance varies significantly between algorithms and implementations. Between implementations ROM usage can vary between 1 to 30× and processing time can vary between 1 to 2000×.

The work here shows which future-proof digital signatures can be used to secure network communication. These algorithms can be used to secure firmware updates and other communication even in a post-quantum scenario. While practical quantum computers are not viable yet, with the average lifetime of embedded devices around 5 to 10 years, the concern is realistic and must be protected against.

However, digital signature do not provide added value in isolation, and exist to protect a payload. They provide the required protection needed for authenticated communication with networked devices, for example to provide a firmware payload. Depending on the exact payload protected, the relative increase on the network burden can be more or less significant. Based on this, a careful consideration is required, as to which post-quantum algorithm puts the least burden on the requirements for the device.

CHAPTER 4

SECURE FIRMWARE UPDATE FRAMEWORK FOR LOW-POWER INTERNET OF THINGS

Similar to unconstrained networked devices, the Internet of Things requires a secure update mechanism to adjust the firmware running on devices. In the previous chapter, cryptographic primitives for authenticating messages are evaluated. These can be used to protect the firmware updates against malicious actors.

Over-the-air firmware updates are an essential part of networked embedded devices. Bugs, including vulnerabilities, in the firmware can be resolved via firmware updates and new features can be introduced. This makes incorporating an update mechanism in any accessible device a fundamental requirement.

Within the heterogeneous space that is the Internet of Things, a generic mechanism to deliver firmware updates to a large number of devices is required. This facilitates the ability to update multiple different devices via the same infrastructure, without the need for dedicated logic for different device types. As different devices use different network and communication technologies, a need exists for a transport independent mechanism for delivering the update.

To facilitate these requirements, a manifest fully describing the firmware update is designed. This manifest describes the steps required to apply the update, and includes checks to validate the applicability of the update itself.

This work is mainly based on previous work “Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check” [3]. Furthermore the work is described as open standard in *A CBOR-based Firmware Manifest Serialisation Format* [124], which has been superseded by a new version: *A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest* [7]. The work here is used for the RIOT-ML toolkit [5]. Furthermore, this work resulted into a number of contributions to the RIOT operating system, spread over multiple pull requests and commits listed in [Table 4.1](#), of which a number have been merged into the software base.

Table 4.1: SUI-related contributions to RIOT

Description	Pull Request
pkg: add support for libcoarse	#8895
suit: Initial minimal CBOR -based SUI manifest parser	#10315
SUIT: provide manifest validation module	#11118
nrf52: use cortexm.ld script when applicable	#11127
suit: cleanup of TinyCBOR to NanoCBOR refactor	#13354
suit: Remove non-standard hello handler	#13385
tests/suit_v4_manifest: Add test for manifest parsing	#13440
SUIT: Update to draft-ietf-v3	#13486
suit/transport/coap: Make use of exposed tree handler function	#13688
SUIT: Upgrade to draft-ietf-suit-manifest-09	#14436
suit: Introduce per-component flags	#15092
suit: Move policy check to before fetch	#15093
SUIT: fail fetch if the image size doesn't match expected	#15094
SUIT: Introduction of a payload storage API for SUI manifest payloads	#15110
suit: Move common storage.c to module directory	#15136
suit/storage/flashwrite: use riotboot_slot_offset	#15306
stm32f{2,4,7}: Initial flashpage support	#15420
examples/suit_update: Add compatibility with native	#15994

4.1 UPDATE ARCHITECTURE

A firmware update is a multi-step process that, given the heterogeneous space, does not follow a fixed process. Different types of devices have different requirements on network links and storage options. Furthermore, different types of updates exist, not limited to firmware and configuration. To allow for this type of flexibility, the update process is described by a manifest, which is parsed and processed by the target device. This manifest describes the steps required by the device to retrieve and install the firmware update.

The overall update process follows the following steps:

1. The device is notified via a push or polling mechanism that a new update is available for installation.
2. The manifest for the update is retrieved for processing on the device.
3. The device verifies the authenticity of the manifest.
4. The device parses the manifest and checks whether the firmware update is applicable to the device.
5. The new firmware is retrieved by the device based on information from the manifest.
6. The firmware is verified and installed according to instructions in the manifest.
7. Based on the manifest, the new firmware is invoked.

4.1.1 DEVICE UPDATE NOTIFICATION

The first step provides a mechanism for the device to be notified that a new manifest is available for consumption. This requires the device to have some network connectivity to a central repository or orchestration server. The mechanism can be either a push-based mechanism, actively notifying the device of the available manifest, or a poll based mechanism. For example, for firmware updates it could be sufficient to check only daily for updates with a central authority.

4.1.2 MANIFEST RETRIEVAL

When a new manifest is available, it must be downloaded and processed on the device. This requires network connectivity to the device and limited memory to store the manifest on the device for processing. A protocol such as CoAP [147], suitable for constrained devices, can be used here to retrieve the manifest.

4.1.3 MANIFEST AUTHENTICITY VERIFICATION

The manifest itself is protected from malicious actors by authenticating it via a digital signature or Message Authentication Code (MAC). This requires the device to contain a root of trust via which to authenticate the update. By itself the cryptographic signature only protects against modification to a manifest. However, an old manifest can still be submitted to a device to force it to downgrade to a vulnerable firmware. To protect against this attack, the manifest contains a serial number which must be incremented for every new manifest. The device is only allowed to process a manifest if this serial number is greater than any previously applied manifest.

4.1.4 FIRMWARE UPDATE APPLICABILITY CHECKS

As the environment consists of multiple heterogeneous devices, not every firmware is applicable to every device. To prevent an accidental installation of invalid firmware, the manifest can provide a set of checks to verify whether the firmware is applicable to the device. This allows a device to reject a firmware when it must not be applied to the device.

4.1.5 FIRMWARE RETRIEVAL

The manifest contains the location where the firmware can be retrieved by the device, and a location where it must be stored on the device. The device retrieves the firmware by itself from the location, which can again be a CoAP-based URL.

4.1.6 FIRMWARE AUTHENTICITY VERIFICATION

The authenticity of the firmware itself is protected via a digest contained and protected by the manifest. This allows the device to guarantee the integrity of the new firmware. After verification, the new firmware can be installed by the device in the final location. Depending on the exact architecture used by the device, the new firmware can be downloaded directly to the installation location and wiped if the authenticity could not be verified.

4.1.7 FIRMWARE INVOCATION

Finally, after installing the new update, it must be invoked by the device. Usually this involves a reboot of the device to start the new firmware. The manifest can instruct the device to invoke the new firmware immediately after installation, or wait for a specific event or instruction before invoking the new firmware.

While this architecture provides the option to also update the firmware via out-of-band mechanisms and is not restricted to networked devices, in this work the focus lies with networked devices. However it is possible to deliver the manifest and firmware over interfaces such as USB or RS-232 or similar mechanism.

4.2 FIRMWARE REQUIREMENTS

The steps involved to parse and apply a manifest and in turn update to a new firmware involves a number of services and components to be available on the device. These must be provided by the software components handling the manifest, usually the currently running firmware.

A network stack must be provided by the firmware to receive both the manifest and the firmware. Firmware images are often multiple kilobytes in size, sometimes exceeding 100 KiB. This requires a network stack on the device capable of handling such transfers. On the constrained device side, a protocol such as **CoAP** can handle this via the block-wise transfer mechanism, using the `block1` and `block2` options.

The manifest itself must be parsed on the device. This involves a parser for **CBOR** [43] object to parse both the manifest and the cryptographic envelope protecting the manifest.

Furthermore, the device must be able to verify the authenticity of the manifest. For this a root of trust and the necessary cryptographic primitives must be available on the device to verify the digital signature in the manifest.

The running firmware must also contain the capability to write the received firmware to persistent storage, often the flash memory of the device itself. When the bootloader is involved in writing the new

update to the final location, the bootloader must have this capability instead.

Depending on the exact architecture, a bootloader must also be present on the device. The bootloader must either install the new firmware over the previous firmware, or boot the newest firmware from multiple available slots. This depends on whether a single firmware slot is used where the bootloader installs the new firmware. A second option is to use multiple slots. In this case, the running firmware would install the new firmware in a different slot. The bootloader then boots the newest firmware available among the slots. In the latter case the bootloader does not require persistent storage write capabilities.

4.3 MANIFEST DESIGN

The manifest describes the metadata involved in obtaining the payload, the devices to which it applies, and the cryptographic information protecting the manifest. The manifest is encoded using the **CBOR** data format and is structured based on several key components elaborated on below.

First is the outer wrapper structure. This contains the authentication block, the manifest itself and a number of optional elements for extensions. The envelope ensures that processing can be done in a modular way without substantial complexity.

The authentication block inside the envelope contains a **COSE** authentication block, using either a signing or **MAC** type **COSE** object. The **COSE** [143] authentication blocks, consisting of either a sign or Mac type **COSE** object, provide the cryptographic authentication required for the manifest. Via this authentication container the full manifest, and in turn the update payload, is protected against tampering by unauthorized parties.

The manifest itself, inside the envelope container, contains the full information for applying the update. This starts with a structure version for indicating compatibility. The next value is a sequence number. This sequence number ensures protection against replay attacks, where old manifest are resubmitted to a device to force a firmware downgrade. This sequence number must always be higher than previously decoded manifest, ensuring protection against replay attacks.

The rest of the manifest structure, inside the envelope, contains information on how the payload should be applied to the receiving system. This contains the set of payloads, any dependencies, a set of pre-installation instructions, installation instructions and post-installation instructions. Furthermore some human readable text can be included and a Concise Software Identification Tag can be included.

The pre-, post- and regular installation instructions consist of a number of condition checks and directives controlling the full installation process. The condition checks, as visible in **Table 4.2**, allow the receiving device to reject the update based on these conditions. If one of the preconditions do not succeed, the manifest is rejected. This for

Table 4.2: SUIT pre-installation condition checks

Id	Name	Argument
1	Vendor Identifier	UUID
2	Class Identifier	UUID
3	Device Identifier	UUID
4	Image Match	Digest
5	Image Not Match	Digest
6	Use Before	Time
7	Minimum Battery	Integer
8	Update Authorised	Integer
9	Version	Integers
10	Component Off-set	Integer

Table 4.3: SUIT pre-installation directives

Id	Name	Argument
1	Wait Until	Time
2	Day of Week	Day
3	Time of Day	Time
4	Battery Level	Integer
5	External Power	-
6	Network Disconnect	-

example allows for rejecting a manifest received by the incorrect device hardware. The directives, as shown in Table 4.3 contain more active instructions for the device to act upon. This includes waiting for an external event or time to happen, for example waiting for the device to be plugged into an external power source.

The installation info contain a number of elements to help the receiving device to retrieve and process the binary for the update. For each payload component in the manifest to be installed on the receiving device, this element supplies the critical information needed for that payload. This includes the size of the payload and the location where to retrieve the payload from. Another essential element is the digest over the payload, which protects the payload from tampering via this digest, which is in turn protected by the COSE element in the outer wrapper.

Finally the post installation info contains directives and conditions which can be used for further actions for the device after a successful installation. For example the device can be instructed to reboot after the manifest has been applied. The post installation conditions can be used to verify the state of the device after the update has completed.

4.4 IMPLEMENTATION OF SECURE FIRMWARE UPDATES

4.4.1 SCENARIO SETUP

[todo] Rework

Prior work [126] outlines requirements for firmware updates of IoT devices, and lists various common deployment scenarios. The common scenario used here is that of a low-power IoT device as target for the firmware update. The IoT device is connected through a low-power low-throughput wireless network to a device management server, which runs on the internet.

The scenario assumes an IoT maintainer, or firmware developer, in charge of maintaining the firmware and of updating the device when required. Over the lifetime of this IoT device, an authorized IoT maintainer should be able to:

1. Produce firmware updates that are integrity-protected and authenticated;
2. Trigger the device to fetch (via push or pull) and verify the integrity and authenticity of a firmware image, and then reboot;
3. Delegate authorization to another maintainer, in case of new ownership or change of contracts, the same technique is used to switch trust anchor when it expires or has to be revoked;
4. Reconfigure the device so that cryptographic algorithms can be upgraded, if needed.

There are several aspects not explored in the prototype:

- Only the case where the entire firmware is replaced is considered, differential updates are out of scope
- The focus is on the use of asymmetric cryptography for digital signatures, even though a symmetric key solution is also possible.
- Firmware encryption is not supported.
- Proprietary protocols are avoided, the focus is on open source software and open protocols. Therefore, the optimization potential is not explored. The results should therefore be interpreted as representing the "lower bar".

The prototype is designed such that multiple configurations are possible. For example, to switch crypto algorithms, crypto libraries, and network stacks. This code can be executed on IoT hardware from different vendors. This provides with a good basis for comparing different features.

4.4.2 COMPONENTS AND FUNCTIONAL OVERVIEW

The prototype utilizes the following building blocks:

- The firmware metadata format based on the IETF SUIF manifest.
- The 6LoWPAN, IPv6, and CoAP transport stack present in RIOT.
- The LwM2M IoT device management solution.
- Digital signature algorithms based on Ed25519 and ECDSA P-256r1.

The RIOT operating system is used for this prototype, but the results can be transferred to other real-time operating systems. Within RIOT, both the build system and the code have been adapted to incorporate the prototype. The remainder of this section provides a functional overview of the prototype.

IOT DEVICE COMMISSIONING

From the embedded software point of view, the prototype firmware layout is based on the design shown in Figure 4.1. The flash layout consists of:

1. A minimalistic bootloader, invoking the newest firmware between the two slots.
2. Two firmware image slots in flash memory, each prefixed with space for their respective metadata structures.
3. A basic firmware update module, also implemented on top of RIOT, integrated into each firmware image.

The RIOT build system is extended to enable a maintainer to simultaneously build and flash (through the serial or USB port) the bootloader and the initial firmware in the first slot. The initial firmware includes a

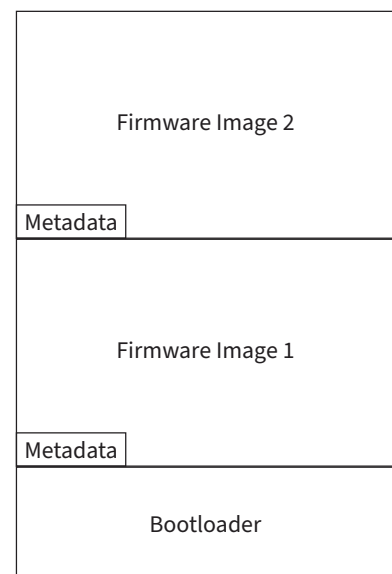


Figure 4.1: Microcontroller flash memory layout

software module for firmware updates, configured with the necessary trust anchor of the maintainer.

TRUST ANCHOR

Our model is based on a single trust anchor, namely of the authorized maintainer. This trust anchor is used to verify the authenticity of the signed firmware image. If an attacker manages to trick the maintainer into handing out the private key associated with the trust anchor, the attacker can load malicious firmware images onto the IoT device. An attacker could make the compromised maintainer sign malicious firmware images. Alternatively, the compromised maintainer could relinquish authorization to the attacker. There is no mitigation when the only trust anchor used is compromised. This prototype relies on the maintainers' ability to keep their private keys secure. Extensions using a full public key infrastructure, potentially with a hierarchy of keys, is possible but out of scope for this prototype.

PRODUCING AND UPLOADING AN AUTHORIZED FIRMWARE UPDATE

The existing build system of RIOT is extended so that a maintainer, a software developer, can simultaneously build a new firmware image and produce the corresponding metadata, signed with the private key of the maintainer. The firmware and signed metadata can then be uploaded to the IoT software update server, using an HTTP-based API. The update server is a web server, which can speak both HTTP and CoAP. It interfaces with the maintainer of the firmware and with the IoT device.

FIRMWARE UPDATE MODULE

The firmware update module's main tasks are to retrieve the firmware image and manifest from the update server, to parse and verify the manifests, and to store the firmware image on flash memory. The module implements the required buffering between the network packet size and the device flash page size. When a flash page buffer is full, the module writes the buffer to the next flash page in the (non-active) firmware image slot. After the entire firmware image has been written to flash, the module computes a hash and checks that this hash is identical to the hash announced in the transferred firmware's metadata. The received metadata is cryptographically verified with the help of the trust anchor (the public key stored on the device). If the digital signature is verified, and if other security checks pass (for example, the firmware sequence number is confirmed to be newer), the module also writes the metadata to the flash (otherwise, the metadata is blanked) and launches a reboot. The bootloader then reads the metadata from the two available firmware slots and chooses to boot the newest valid firmware, based on the metadata. Note that, due to blanked metadata, an interruption (e.g. due to power loss) cannot cause the system to boot of an invalid, corrupted or incompletely received image.

SCHEDULING FIRMWARE UPDATES

Using the firmware update module, updates can be (i) either triggered periodically or on demand, (ii) pushed to the device or pulled from the device [63], so as to fit other operational constraints. On the device the real-time, preemptive multi-threading capabilities of RIOT is used, such that the system is not blocked by the computational-intensive task of digital signature verification. In practice, signature validation runs in a separate thread, with low priority, enabling other threads with top priority to execute as needed. However, note that advanced fine tuning is not done for the schedule of firmware updates (e.g. to guarantee the continuity of some service provided by the device, or to optimize network load). Instead, the focus is primarily on the fundamental embedded system characteristics and constraints imposed by standard-compliant firmware update on-board constrained IoT devices.

LIFE CYCLE MANAGEMENT

By changing the trust anchor stored in the next firmware’s update module, authorization to update the firmware can be delegated to another maintainer, who can take over the production and the roll out of authorized updates.

Crypto agility is straightforward because the update module in the new firmware image can implement and use upgraded cryptographic primitives. This flexibility is provided because the cryptographic primitives are implemented fully in the firmware.

Key roll-over is also made possible with the ability to update the trust anchor.

4.5 CONFIGURABILITY OF THE PROTOTYPE

The prototype designed can be configured in multiple ways, as summarized in Table 4.4.

	Firmware Update	IPv6 Support	Standardized Manifest	Device Mgmt
Baseline	×	✓	×	×
Basic-OTA	✓	×	×	×
IPv6-OTA	✓	✓	×	×
SUIT-OTA	✓	✓	✓	×
LwM2M-OTA	✓	✓	✓	✓

Table 4.4: Analyzed Configurations.

The following configurations have been created:

4.5.1 BASELINE

The Baseline configuration covers a typical sensor scenario, and is introduced here only as a reference, to evaluate the relative cost

of over-the-air (over-the-air (OTA)) firmware updates. Therefore, this configuration does not provide firmware update functionality. The Baseline configuration uses 6LoWPAN over IEEE 802.15.4 as a network stack. A CoAP server is installed on the IoT device to respond to requests for sensor data and to actions that trigger an actuator.

4.5.2 BASIC-OTA

This configuration enables over-the-air firmware updates pushed directly from the update server to the IoT device, over the Media Access Control (MAC) layer, without a standard network layer. Therefore, this Basic-OTA configuration requires that the IoT device and the update server can communicate directly over the MAC layer. In other words, they have to be on the same local network or bus. The Basic-OTA configuration uses minimalistic firmware metadata in a proprietary format, namely:

- A sequence number.
- The firmware start address and size.
- A digest of the firmware image.
- A digital signature of the metadata.

4.5.3 IPV6-OTA

This configuration enables the Basic-OTA configuration by using an IPv6-compliant network stack. The IPv6 network layer implementation is provided by the RIOT Generic (GNRC) network stack. CoAP block-wise transfer (block1) is used because UDP limits the size of the firmware image to be transferred to 65,507 bytes and, more importantly, to avoid the inefficiency caused by IP fragmentation.

4.5.4 SUIT-OTA

This configuration implements firmware updates following the IETF SUIT manifest [124]. Compared to IPv6-OTA and Basic-OTA, SUIT-compliant firmware metadata offers more features and additional security guarantees (see section 4.9).

The SUIT manifests used in our prototype contain the following information:

- The firmware version number.
- An 8-byte nonce.
- A monotonic sequence number, for which a Unix time stamp is used.
- A single condition: limiting the validity of the manifest to our device.
- The format of the firmware.
- The size of firmware.

- A storage identifier.
- A single URI to allow the device to download the firmware.
- A SHA256 digest.
- A digital signature on the manifest.

Upon receiving a manifest, the IoT device checks the signature, and, if verified correctly, pulls the firmware from the URI indicated in the SUIT manifest. To pull the firmware image, again CoAP block-wise transfer (CoAP block2 option) are used. It would be possible to attach the firmware to the manifest, but using this two-step approach gives us extra flexibility.

4.5.5 LWM2M-OTA

This configuration adds support for LwM2M v1.0, without the use of the bootstrapping functionality. The device registers to a LwM2M server and provides the necessary API endpoints complying with the LwM2M specification and the core objects, such as the LwM2M Device and the LwM2M Firmware Update objects. The firmware is updated by pushing a SUIT manifest to the Package resource found in the LwM2M Firmware Update object followed by the workflow corresponding to the SUIT-OTA configuration.

In the analyzed configurations above, TLS/DTLS was not used between the IoT device and the update server or device management server for LwM2M. Implementing TLS/DTLS is certainly useful when considering the larger device management functionality in addition to the firmware update. An analysis of IoT device management functionality is, however, outside the scope of this chapter.

4.6 PERFORMANCE EVALUATION

For the evaluation, commercially available hardware based on Arm Cortex M microcontrollers is used. The following hardware from three different vendors is used:

- Atmel SAMR21, which features a Cortex M0+ MCU with 32 kB of RAM and 256 kB of flash.
- STM32F103REY, which features a Cortex M3 MCU with 64 kB of RAM and 512 kB of flash.
- Nordic nRF52840, which features a Cortex M4 with 256 kB of RAM and 1 MB of flash.

The STM32F103REY and the nRF52840 are clocked at 64 MHz, while the SAMR21 runs at 48 MHz. In the following measurements, the code is compiled using GCC 7.2.0 for Arm optimized for code size.

To evaluate cost in this comparative evaluation, the memory required, both flash and RAM, and the CPU performance is measured. These metrics are decisive in terms of hardware costs and in terms of energy costs [59] on these constrained devices.

Component	Bootloader	Baseline	Basic-OTA	IPv6-OTA	SUIT-OTA	LwM2M-OTA
Core	2 760	13 976	10 913	13 241	14 388	14 175
Network	0	26 892	2 732	26 892	27 230	27 208
CoAP	0	1 876	1 910	1 910	2 286	2 676
Crypto	0	308	5 798	5 886	6 472	6 472
COSE & CBOR	0	0	0	0	3 181	3 181
SUIT	0	0	0	0	1 575	1 551
OTA	0	0	2 007	2 007	3 998	3 475
LwM2M	0	0	0	0	0	2 166
Sub-total per image	2 760	43 052	23 360	49 936	59 130	60 904
Total flash footprint	2 760	43 052	49 544	102 696	121 084	124 632

Table 4.5: Flash requirements, in bytes, per component and configuration, on Cortex M0+.

Component	Bootloader	Baseline	Basic-OTA	IPv6-OTA	SUIT-OTA	LwM2M-OTA
Core	800	2 410	1 317	2 410	3 914	3 919
Network	0	11 010	7 224	11 010	11 010	11 026
CoAP	0	1 536	2 560	2 560	1 024	1 024
Crypto	0	28	28	28	60	60
COSE + CBOR	0	0	0	0	512	512
SUIT	0	0	0	0	296	272
OTA	0	0	632	632	2 984	3 000
LwM2M	0	0	0	0	0	1 487
Total	800	14 984	11 760	16 640	19 800	21 300

Table 4.6: RAM requirements for bytes of statically allocated stack, per component and configuration, on Cortex M0+.

4.7 RELATIVE IMPACT OF CRYPTOGRAPHIC LIBRARIES

In constrained embedded devices, the use of cryptography significantly impacts memory and power budgets. To get an idea of the significance of the impact, both the relative memory budget and time spent due to crypto for the Basic-OTA configuration of the prototype is measured, while using the HACL crypto library [177]. First, the time spent on different tasks is shown in Figure 4.2. The bulk of the time and thus energy is spent on signature verification and network transport. The rest of the time is spent on the parsing of the network packets, firmware metadata parsing and validation, this time and energy is negligible (less than 2 %) when compared to the signature verification and network transport. Note that this remains true with other configurations of our prototype as well, using a more elaborate network stack (CoAP) or more elaborate metadata (SUIT). Next, it is observed in Figure 4.3 that cryptographic functions represents 50 % of the memory budget. Going back to Figure 4.2, it seems at first sight that time spent during a firmware update is dominated by network transfer with 60 %, then signature verification with 38 %. However, observe that, since half of the firmware image size is contributed by cryptographic functions, this means 30 % of the time is spent on transferring updated cryptographic

functions inside the firmware over the network, half the total network transfer time. In effect, the conclusion is that handling cryptography dominates, accounting in fact for 68 % of the total time spent on the firmware update process. I conclude that choosing an appropriate cryptographic algorithms and library, offering a good compromise on code size and verification speed, is crucial. In the following section, this topic is discussed in greater detail.

4.8 EVALUATING THE COST OF THE OTA UPDATE FUNCTIONALITY

To evaluate the cost of the firmware update functionality, the RAM and flash memory overhead incurred by this functionality is measured and compared in the prototype for the various configurations defined in [section 4.5](#). The flash memory footprints (total and broken down per component) are shown in [Table 4.5](#), while [Table 4.6](#) shows the RAM requirements calculated for the stack measured on an Atmel SAMR21 (using a Cortex M0+, the most constrained microcontroller used in these experiments). In these two tables the bootloader is listed separately as it is present on the device alongside as shown in [Figure 4.1](#).

The different components of the system are distinguished as follows:

- The *core* component combines the minimal basic operating system functionality, including drivers. The newlib-nano standard C library is also included.
- The *crypto* component includes cryptographic algorithms, such as digest algorithms, the digital signature algorithm, the elliptic curve cryptography and big number library together with the pseudo random number generators.
- The *network* component includes the protocol stack from the radio driver up to the transport layer protocol UDP.
- The modules that enable a firmware update to be received and stored in flash memory are combined in the *OTA* component.
- *CoAP* refers to the [CoAP](#) protocol stack.
- *COSE+CBOR* contains the libraries for [COSE](#) parsing and [CBOR](#) parsing.
- *SUIT* relates to the code parsing a [SUIT](#) manifest.
- Finally, *LwM2M* contains the code for device registration, and functionality required for the [LwM2M](#) protocol to perform firmware updates (particularly the [LwM2M](#) Device and Firmware Update objects).

4.8.1 THE COST OF OTA

The cost of basic [OTA](#) functionality can be measured by comparing the memory requirements of the Baseline configuration with that of the

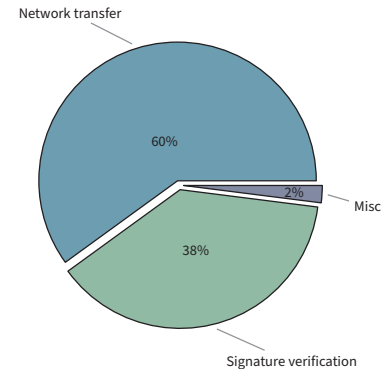


Figure 4.2: Time spent per subtask in a firmware update.

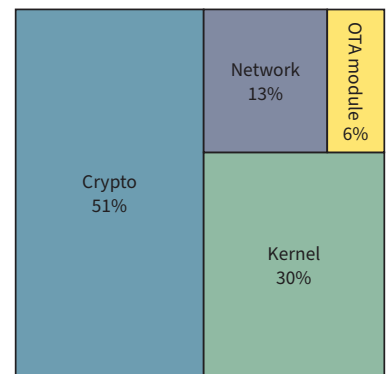


Figure 4.3: Flash memory budget per system component. Basic-OTA configuration, 35kB flash in total.

IPv6-OTA configuration. On a per-image basis, the flash overhead comes from the need for additional modules to perform necessary crypto (5 kB) and to handle OTA (2 kB). However, the prototype needs two image slots with metadata and a bootloader. Therefore, the comparison is with the Baseline flash footprint against twice the flash footprint of IPv6-OTA added with the bootloader footprint (see Table 4.5). In total, the relative overhead in flash memory footprint is 137 %, 59 kB more. Note that this overhead means that the flash memory budget crosses over from below 64 kB to below 128 kB. The largest part of the overhead comes from the doubled image slots. The footprint of the rest (bootloader and metadata) is small: approximately 3 kB of flash for the bootloader and a single flash page for the metadata of each image.

4.8.2 THE COST OF STANDARDS COMPLIANCE FOR OTA

The use of standards-compliant specifications, such as SUIT and LwM2M, increases the memory footprint due to the extra functionality provided. Where the BASIC-OTA scenario used a firmware OTA mechanism optimized for the scenario, using standards-compliant update mechanisms add extra capabilities not leveraged by the scenario. For example, serialization, metadata processing, and object handling all add extra processing and memory overhead. This is expected.

Observe that the relative overhead per image, compared to the Baseline scenario, is small. This is because a lot of features present in the firmware are reused within the network module of each configuration. Furthermore, it is not unlikely that, OTA functionality aside, application code already leverages CBOR, COSE, and other cryptographic functionality. In such cases, the extra memory overhead per image falls to approximately 10 %. This type of software reuse is a clear advantage of using building blocks leveraging existing standards.

Compared to the 124 B of metadata transferred over the network with the Basic-OTA configuration, 226 B of metadata need to be transferred with the SUIT-OTA configuration (counting full COSE data).

Due to the flash memory alignment constraints on the IoT device, this overhead has no effect on the flash memory footprint, because 226 bytes typically fit on a single flash page. For example, 256 bytes fit on a single flash page on the Atmel SAMR21, the most constrained microcontroller used in the measurements.

Extending our measurements to the SUIT manifest case, the code has to be extended with components required by the SUIT specification. A SUIT module and the necessary serialization and cryptographic functions increase the flash size by 10 KB compared to the simple OTA scenario. While the COSE and the CBOR modules are here specifically required for SUIT compliance, in a real-world scenario these modules could also be used for sensor data encoding and application data encryption.

Using LwM2M compatible handlers for this increases the flash size by

another 2 kB because of the need to implement the mandatory **LwM2M** handlers and the registration protocol. These components must be implemented by every device that is **LwM2M**-compliant and should not be considered as overhead purely related to having over-the-air update functionality.

Finally, observe that none of the configurations experimented with exceeds the thresholds of 32 kB of RAM and 128 kB of flash memory. Although our prototype could be further optimized, it fits the nature of constrained **IoT** devices used in the market today.

4.9 SECURITY ASSESSMENT

Typical threats against a firmware update solution are discussed in the **SUIT** information model [125] and can be categorized into the following list:

- Tampered firmware
- Firmware replay attack
- Offline device attack
- Device firmware mismatch
- Firmware installation flash memory location mismatch
- Unexpected precursor firmware image
- Reverse engineering of the firmware
- Device resource exhaustion

Based on these threats, I assess and compare the security of our prototype in the **IPv6-OTA**, **SUIT-OTA**, and **LwM2M-OTA** configurations, which are defined in section 4.5. The summary of our assessment is shown in Table 4.7.

	IPv6-OTA	SUIT-OTA	LwM2M-OTA
Tampered firmware	✓	✓	✓
Firmware replay	✓	✓	✓
Offline device	×	×	✓
Firmware mismatch	×	✓	✓
Wrong memory location	✓	✓	✓
Unexpected precursor	×	✓	✓
Reverse engineering	×	×	×
Resource exhaustion	×	×	✓

Table 4.7: Security Assessment Summary for different configurations. The ✓ shows which threat vectors are protected against by the different configurations.

4.9.1 FIRMWARE TAMPERING

An attacker may try to update the **IoT** device with a modified and intentionally flawed firmware image. All configurations are protected by digital signatures against this attack vector. To counter this threat, the **IPv6-OTA**, **SUIT-OTA**, and **LwM2M-OTA** configurations use digital

signatures to ensure integrity of both the firmware and its metadata. Additionally, the device can verify that an authorized maintainer signed the firmware image via this signature.

4.9.2 FIRMWARE REPLAY

An attacker may try to replay a valid, but old firmware with known vulnerabilities. This threat is mitigated by using a sequence number inside the manifest, where devices reject the update when the sequence number is not higher than any previously seen valid manifests. All three configurations use such a sequence number, which protects them against this attack vector.

4.9.3 OFFLINE DEVICE ATTACK

An attacker may cut communication between the IoT device and the update server for an extended period of time. Then, they may try to update the IoT device with a (known-to-be-flawed) firmware image, which has in the meanwhile been deprecated. IPv6-OTA does not provide any mitigation against this threat.

Following the SUIT specification, a best-before time stamp can be used to expire an update. However, this requires the IoT device to have an approximate knowledge of the current date and time, which may not be available on constrained IoT devices. Therefore, our SUIT-OTA configuration does not mitigate this threat either. Only the LwM2M-OTA configuration may protect against this attack since LwM2M offers an integrated way to provision the device with date and time information.

4.9.4 DEVICE FIRMWARE MISMATCH

An attacker may try replaying a firmware update that is authentic, but for an incompatible device. The IPv6-OTA configuration does not provide mitigation against this threat, the configuration does not have a device-specific identifier in the protocol. The SUIT-OTA and the LwM2M-OTA configurations include device-specific conditions in the manifest. These conditions can be verified before installing a firmware image, thereby preventing the device from installing and invoking an incompatible firmware image.

4.9.5 FLASH MEMORY LOCATION MISMATCH

An attacker may attempt to trick the IoT device into flashing the new firmware to the wrong location in memory. To mitigate this attack, IPv6-OTA, SUIT-OTA, and LwM2M-OTA specify the intended memory location of the firmware update. The simple update mechanism with IPv6-OTA contains a firmware address which states the location where the firmware has to be installed. The SUIT manifest contains an opaque string to specify the location. Both these mechanisms suitably prevent a firmware from being installed in the wrong location. This way the device can verify that an update is installed in the correct

location, preventing mismatches between the firmware location and the expected location of the new firmware.

4.9.6 UNEXPECTED PRECURSOR IMAGE

When using an incremental update scheme, where there is a tight coupling between the installed firmware version and the firmware version from the update, a match between the two versions is essential. An attacker may try to exploit a vulnerability that results from a mismatch between previously installed firmware and the new firmware. While the prototype only uses full firmware images, extending it to incremental updates is possible. While IPv6-OTA does not mitigate this threat, SUIP-OTA and LwM2M-OTA enable specifying the precursor software that must be installed before the update can be applied, enabling modular or incremental updates. The simple firmware update scenario doesn't support a precursor image, it must not be used with incremental updates to prevent this vulnerability from being abused.

4.9.7 FIRMWARE REVERSE ENGINEERING

The firmware image in transmission can be captured by an attacker for vulnerability analysis. Neither the IPv6-OTA configuration nor our SUIP-OTA configuration protect against eavesdropping end-to-end, from the maintainer to the IoT device. Note that the SUIP specification also defines the ability to encrypt the firmware image [164]; however, the prototype here does not make use of this feature. The use of (D)TLS in the SUIP-OTA or LwM2M-OTA configurations can also protect the firmware image against eavesdropping in-flight, while transmitted over the network, but doesn't offer end-to-end security without the extra protection offered by using SUIP.

4.9.8 RESOURCE EXHAUSTION

Receiving, verifying, and storing a new firmware is an operation that typically uses up a significant amount of resources on a constrained IoT device. As discussed in chapter 3, signature verification, both pre- and post-quantum, can take several seconds depending on the library used. By repeatedly attempting fraudulent firmware updates, an attacker may deplete the device's battery or, more generally, make it unavailable for long periods of time. For example, an attacker who manages to transmit valid manifests without a valid signature to an IoT device at regular intervals can drain the battery.

The IPv6-OTA configuration does not mitigate this threat, but the SUIP-OTA configuration lowers the impact by verifying the manifest before downloading the firmware image. However, an attacker could still push invalid manifests at any rate, causing the IoT device to perform signature verifications. Using LwM2M, an additional layer of defense can be added by only processing manifests that are conveyed via the device management infrastructure. In this way, the IoT device trusts the LwM2M server to only forward manifests that pass the following security checks:

- The URL in the manifest points to a firmware update server under the control of the **LwM2M** infrastructure.
- The manifest signature has been verified correctly.
- Other conditions in the manifest (such as the *best-before* time stamp) have been processed successfully.

If the device management server is compromised, the security characteristics of the **LwM2M-OTA** configuration fall back to those of the **SUIT-OTA** configuration.

4.10 DISCUSSION

4.10.1 MAKING THE FIRMWARE UPDATE RELIABLE IS KEY

With the system described, the maintainer is expected to test the new firmware properly before rolling it out. At a minimum, the new firmware must be able to update itself one more time over-the-air. Guarantees beyond this minimum requirement, such as the use of watchdog timers and the ability to use a “factory reset”, fall into the realm of traditional embedded software management and increase the flash memory requirements. Without taking these considerations into account, failures, like those reported with the Taiwanese YouBike service [99] and the Japanese X-ray telescope satellite Hitomi [119], are likely to occur again.

4.10.2 USE DELEGATION CAPABILITIES WITH CARE

As the system allows the maintainer to transfer its authority to another entity, the maintainer is entrusted with the responsibility of not transferring authority to malicious entities. If the maintainer is the owner of the device, trust is not an issue; otherwise, maintenance of **IoT** software is typically of a contractual nature, and the caveats of such trust are well-trodden territory. An improvement of the system could use protected memory and/or a dedicated crypto hardware module to validate authority transfer.

4.10.3 SHIELDING AGAINST RESOURCE EXHAUSTION AND BEST-BEFORE VULNERABILITIES

The extent to which an **IoT** device is protected against resource exhaustion attacks depends on the resources of the firmware update server in the **LwM2M-OTA** configuration. The aspect of dimensioning the server’s resources to counter potential DoS attacks is covered by extensive prior work in the domain. In the end, due to extreme lack in resources, constrained **IoT** devices remain intrinsically vulnerable.

4.10.4 REAL-WORLD REQUIREMENTS MAKE FIRMWARE UPDATES COMPLEX

In this chapter the focus of the efforts on the most basic scenario outlined in [126] and refinements were not considered. Examples of such refinements are: firmware encryption, updating devices with multiple microcontrollers, complications due to policy handling, differential updates, or more efficient distribution using multicast. Encryption, for example, raises the question about key management. In a world where software components are developed, maintained, and updated by different developers, additional challenges arise. While the advantages are known from web development, there are questions about how to trace component versions and their composability with other software libraries, how to sandbox components in constrained IoT devices, how to accomplish faster time to market in regulated industries where software development requirements and testing are much harder than on the internet, and so on.

4.10.5 IOT SOFTWARE UPDATES ARE NOT JUST FOR CRITICAL INFRASTRUCTURE

Interdependence between networks has dramatically increased over the past few decades. Enabling and securing firmware updates is necessary for all IoT devices. Both those that are inside the infrastructure perimeter, for example, industrial sensors, and outside the infrastructure perimeter, such as consumer smart appliances. For instance, a recent study [154] shows how the power grid is indirectly vulnerable to DDoS attacks from hacked consumer appliances in smart homes. Using simulations, the study shows how a botnet controlling a relatively small number of connected water heaters and air conditioners could maliciously disrupt power demand and take down most of a large power grid serving an area as large as Canada, affecting tens of millions of people.

4.11 CONCLUSION

In this chapter, open standards have been surveyed, which provide generic building blocks for secure firmware updates on constrained IoT devices. I build a basic prototype, bundling such standard building blocks and avoiding proprietary components as much as possible. With this the security characteristics of the resulting system have been assessed, and show how it brings state-of-the-art security to IoT devices. The cost of enabling the firmware update solution in our prototype is bearable, in terms of the required memory and computation, with the currently available IoT hardware. I demonstrate that it is possible to implement a generic, standards-compliant firmware update solution on IoT devices without exceeding the typical thresholds of 32 KiB of RAM and 128 KiB of flash memory.

Including a firmware update mechanism in IoT devices is a must-have security measure against future vulnerabilities. The need to secure

constrained devices in the field exacerbates this need. However the mechanism to update firmware through **SUIT** can be leveraged for other payloads beyond firmware files. Compartmentalization of firmware through virtualisation can help reduce the update size when only the affected module requires updates.

The ability granted by the **SUIT** manifest to instruct a device to retrieve and install a payload is not limited to firmware. While the reasons to enable firmware updates are numerous, the ability to update any payload on the device such as individual modules and off-chip code can be leveraged in multiple ways.

One such way is to use **SUIT** to update the application running inside small **VMs**. In this scenario the **SUIT** manifest specifies where the application code of the **VM** must be retrieved from and where it should be installed to. One such **VM** is rBPF presented in [chapter 5](#).

CHAPTER 5

RBPF: A TINY SOFTWARE-ONLY VIRTUAL MACHINE FOR INTERNET OF THINGS FIRMWARE

In the previous chapter, this thesis has presented mechanisms resolving firmware vulnerability via secure over-the-air updates. This allows for resolving existing vulnerabilities when discovered. However, this assumes vulnerabilities are found, and it does not protect against undiscovered and unpatched vulnerabilities.

Small virtual machines hosted on embedded devices can act as sandbox for the device, to isolate and reconfigure part of the application code. These can be configured to be isolated from the main microcontroller by default, preventing them from interfering with the firmware on the device.

While a number of virtual machines specifically for embedded systems already exist, none of these are promising in terms of memory footprint. Usually these virtual machines require multiple tens of kilobytes of flash memory and significant RAM to operate, making them cumbersome to add to existing firmware.

In this chapter, a **VM**, rBPF, is introduced as tiny software-only **VM** to isolate software components inside. With rBPF, two use cases are considered:

- Isolating high-level business logic, updatable on demand remotely over the low-power network. This type of logic is rather long-lived, and has loose (non-real-time) timing requirements.
- Isolating debug/monitoring code snippets 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 content in this chapter is published as “Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF” [1] in PEMWN 2020 – 9th IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks. Furthermore, the rBPF **VM** has been submitted to the RIOT operating system for inclusion in a pull request [12].

5.1 DESIGN GOALS & REQUIREMENTS

For the rBPF VM designed in this work, a number of design goals are set. These goals are to ensure that rBPF is a proper fit for the use case.

5.1.1 MINIMAL MEMORY FOOTPRINT

rBPF must require a minimal memory footprint. Small microcontrollers are already constrained on resources by their design. Still a regular firmware with full capabilities for the design goals of the particular embedded system must be programmed on the device. Given that running rBPF is not the main purpose of the system, it must not take up unreasonable amount of memory on the system, but leave the bulk of the resources to the main application and firmware running.

5.1.2 NO RELIANCE ON HARDWARE-SPECIFIC MECHANISM FOR MEMORY PROTECTION

Multiple types of hardware memory protection systems are available on current generation microcontrollers. However, whether these are available on a given microcontroller depends on the model and the manufacturer. The design of rBPF is such that no assumptions are made on the availability of these kind of hardware-specific mechanisms. Instead software-based memory protection is used where memory protection is based on policies loaded into the VM on execution.

5.1.3 TOLERABLE CODE EXECUTION SLUMP

Any interpreter is bound to be slower than native instruction execution. Even more so when the interpreter must cover the security aspects involved in protecting the host system. The rBPF VM is no exception on this. While a code execution slump is expected and rBPF is not designed for fast execution, the slow down incurred must be tolerable for executing small applications in limited time.

5.1.4 SMALL APPLICATION CODE SIZE

The applications loaded into the VM are likely to be transferred over-the-air to the system running. These applications are updated occasionally and thus must not incur a significant burden on the rest of the network around the system. In turn this means that the application size overhead must not be too large. As such, formats such as executable and linkable format (ELF) contain more information than strictly needed and would incur significant overhead on the transfer. Instead the binaries to be loaded in rBPF must be lean and not incur significant overhead.

5.2 VIRTUAL MACHINE DESIGN

The rBPF VM is a variant of the Linux eBPF VM, designed to execute eBPF instructions as emitted by a compiler. The main difference lies in the bindings provided to the operating system and the events by which execution is triggered. An overview of how the VM is integrated into RIOT is shown in Figure 5.1. The VM runs as a regular thread inside the operating system, restricted by the scheduler to the configured run priority. Within the rBPF VM, the sandboxed application is only guaranteed to have access to a persistent key-value store. Further integration with the operating system is available through bindings, including access to facilities relevant to IoT applications such as sensor values and CoAP packet creation.

5.2.1 EXECUTION HOOKS

The VM execution is geared towards short lived and event triggered applications. Execution is triggered by events in the operating system when an application is added to the respective event hook. These hooks are added to specific conditions inside the operating system such as network packet reception. The application running inside the VM is expected to be short-lived, updating an intermediate result or formatting a response to a request. The VM does not interfere with real-time thread execution on the operating system. However, the VM itself is not suitable for running hard real-time applications, as this is not part of the design requirements.

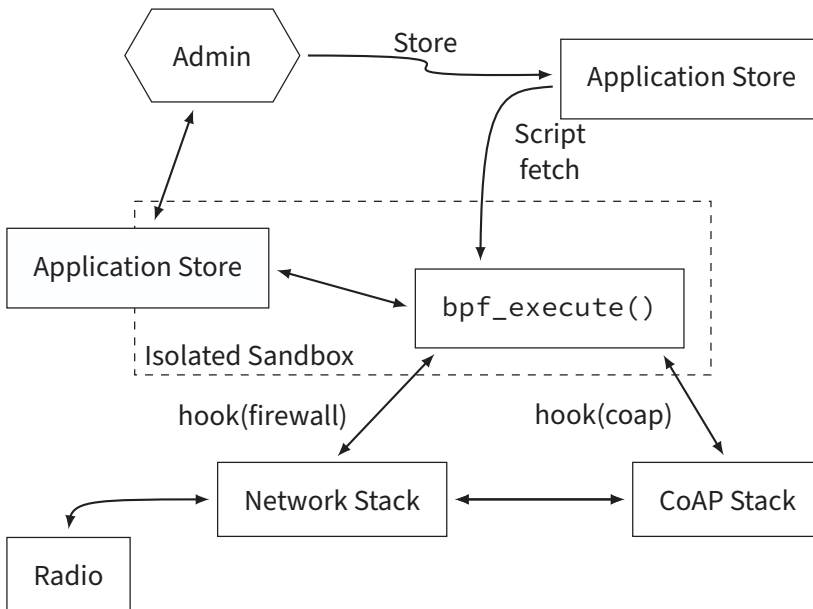


Figure 5.1: Integration & sandboxed execution of rBPF VM in host OS.

As shown in Figure 5.1, multiple sources can trigger the execution of a script. This includes requests received on the CoAP server or packets passing through the network stack. Each event can trigger a different rBPF application from the application store, configured by the administrator. Similar to eBPF the VM supports both an argument passed to the application and a return code from the application back to the calling event. This can be used to communicate vital

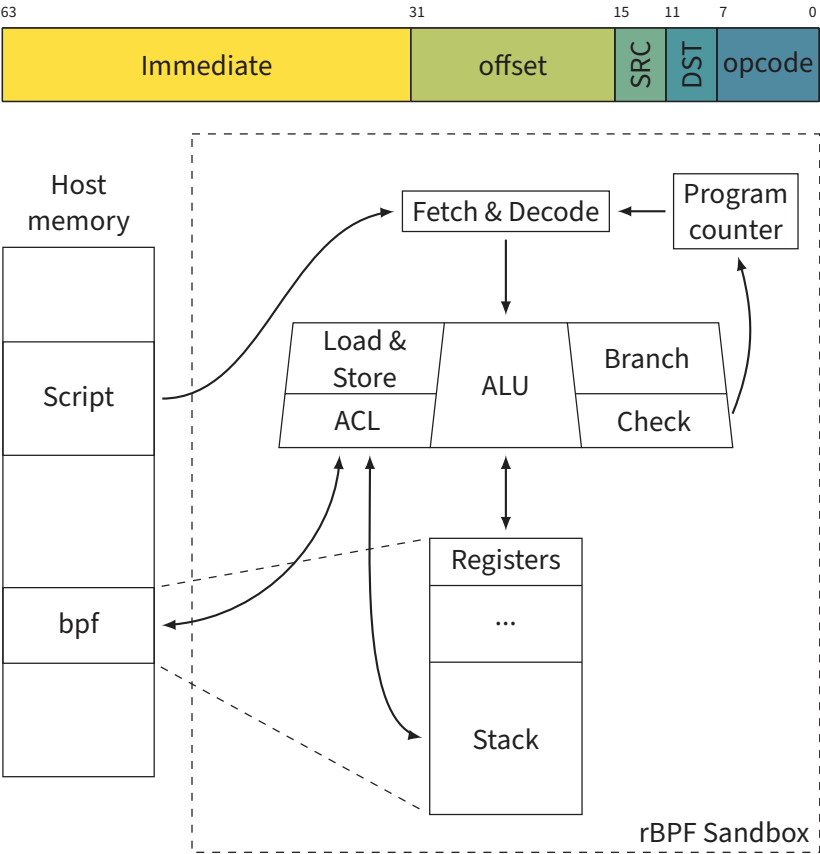


Figure 5.2: eBPF instruction format

Figure 5.3: rBPF execution and memory architecture

execution context with the **VM** and pass a return value back to the initiator. With these capabilities the **VM** application is isolated from the operating system, while retaining enough flexibility to host business logic applications, or simple measure and debug applications.

5.2.2 ARCHITECTURE

The **VM** is based on a simple loop design, iterating over the application instructions as shown in Figure 5.3. The interaction between the instructions, the sandbox guards in place, and the host address space is shown. Both the registers and the application stack reside in the RAM of the host.

To prevent some overhead during the execution of the loaded applications, safety checks are performed before execution where possible. Every instruction is checked for validity of the opcode, the supplied registers, and where applicable the jump offset of branch instructions. This is designed as a single scan pass over the instructions of the loaded application.

INSTRUCTION SET

The eBPF applications used directly by rBPF consist of sequences of 64-bit instructions. Each eBPF instruction consist of an opcode, source and destination registers, an offset and an immediate bit field as shown in Figure 5.2. Not all instructions use all bit fields, and unused fields must be zeroed.

Table 5.1: rBPF arithmetic instructions

Opcode	Pseudocode
0x07	dst += imm
0x0f	dst += src
0x17	dst -= imm
0x1f	dst -= src
0x27	dst x= imm
0x2f	dst x= src
0x37	dst /= imm
0x3f	dst /= src
0x47	dst = imm
0x4f	dst = src
0x57	dst &= imm
0x5f	dst &= src
0x67	dst <=<= imm
0x6f	dst <=<= src
0x77	dst >>= imm (logical)
0x7f	dst >>= src (logical)
0x87	dst = -dst
0x97	dst %= imm
0x9f	dst %= src
0xa7	dst ^= imm
0xaf	dst ^= src
0xb7	dst = imm
0xbf	dst = src
0xc7	dst >>= imm (arithm)
0xcf	dst >>= src (arithm)

For the load and store type instructions, the opcode field is structured along [Figure 5.4](#). The opcode `class` subfield specifies a load or store instruction. The `size` specifies the number of bits operated on, allowing for 8 bit to 64 bit for load and store instructions. Finally the `mode` field specifies the type of load instruction. While different modes of load instructions might be developed at some point, rBPF only implements memory loads and stores, the `mode` field is always set to `0x3`.

The load and store instructions available in rBPF are shown in [Table 5.2](#). Instruction `0x18` follows a slightly different pattern. This instruction uses the size of two instructions to combine the two 32 bit immediate values into a single 64 bit value, which is loaded into the `dst` register. The second instruction used for its immediate has all other fields of the instruction zeroed out.

Jump instructions also have their own sub format in the opcode encoded as shown in [Figure 5.5](#). The compare operation is encoded in the `op` field. The `IMM` field is cleared when the comparison is against the immediate of the instruction, if it is set the comparison is between the `SRC` and the `DST` registers.

5.2.3 MEMORY PROTECTION

Depending on the instruction to be executed, different protection mechanisms are activated. Two main protection mechanisms are in place to isolate the code executed in the `VM`. One protects against illegal load and store instructions, the other prevents code execution outside the loaded application.

First the host address space is isolated from the sandbox by policies loaded in the `VM`, there is no memory translation to the address space as visible from within the `VM`. Access to the address space is guarded via software memory protection built into the `VM`. Every memory access, including stack reads and writes, are subjected to access policy rules. Different address space sections can be configured to allow reads, writes or both by the caller of the `VM`. This offers minimal overhead for memory access while providing the guarantees required for the sandbox.

Second is the protections on the code executed to ensure that the



Figure 5.4: eBPF load and store instruction opcode format



Figure 5.5: eBPF jump and branch instruction opcode format

Table 5.2: rBPF load and store instructions

Opcode	Store Instruction Pseudocode	Opcode	Load Instruction Pseudocode
0x63	$\ast(u32 \ast) (dst + off) = src$	0x61	$dst = \ast(u32 \ast) (src + off)$
0x6b	$\ast(u16 \ast) (dst + off) = src$	0x69	$dst = \ast(u16 \ast) (src + off)$
0x73	$\ast(u8 \ast) (dst + off) = src$	0x71	$dst = \ast(u8 \ast) (src + off)$
0x7b	$\ast(u64 \ast) (dst + off) = src$	0x79	$dst = \ast(u64 \ast) (src + off)$
0x62	$\ast(u32 \ast) (dst + off) = imm$	0x18	$dst = imm$
0x6a	$\ast(u16 \ast) (dst + off) = imm$		
0x72	$\ast(u8 \ast) (dst + off) = imm$		
0x7a	$\ast(u64 \ast) (dst + off) = imm$		

VM does not start to execute code outside the supplied application such as gadget deployed by an attacker. The mechanism works by guarding the branch and jump instructions, ensuring that the jump is not outside the application address space. As the virtual program counter can only be adjusted by jump instructions, the only guard required is to ensure that jump instructions keep the program counter within the application space.

To provide persistent data between these short-lived invocations a key-value store is available. An application can read and write values to both a global and a per-script local storage. Counters or aggregate sensor values can be stored for retrieval in a subsequent application execution.

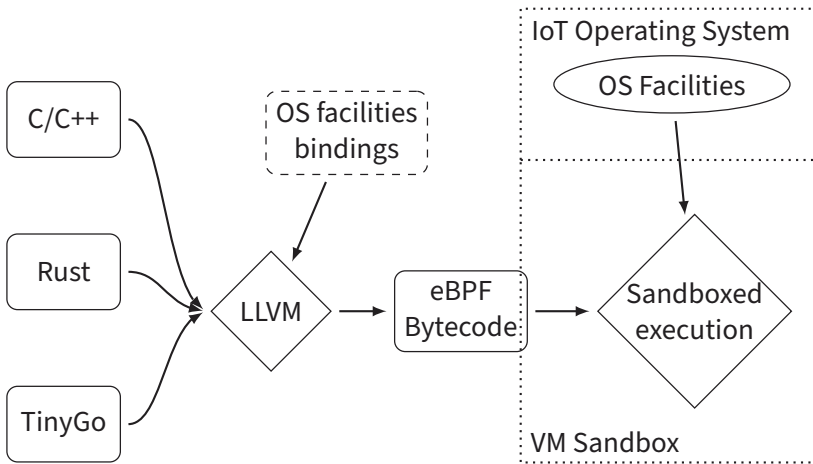


Figure 5.6: rBPF application code development and execution workflow.

5.3 EXPERIMENTAL EVALUATION

The measurements are carried out on popular, commercial off-the-shelf **IoT** hardware: the Nordic nRF52840 Development Kit. This board provides a typical Arm Cortex-M4-based microcontroller with 256 KiB RAM, 1 MiB Flash, and a 2.4 GHz radio transceiver compatible both with IEEE 802.15.4, and Bluetooth Low-Energy. This hardware is also available for reproducibility on open access test beds such as IoT-Lab [17].

On this platform, two types of benchmarks are performed:

1. Measurements of the embedded computing performance provided the VM, to get an idea of basic VM performance.
2. Benchmarks of the **IoT** networking capabilities as provided from within the VM.

As second VM to compare against, a **Wasm**-based virtual machine implementation, WASM3 is used. **Wasm** is used as it is recently developed and ported to microcontrollers and is currently popular for development as described in subsection 2.7.2.

5.3.1 COMPUTING BENCHMARK SETUP

First, the Fletcher32 checksum algorithm [70] is used as basic performance benchmark. The Fletcher32 checksum algorithm requires a mix of mathematical operations memory reads and branches, containing a loop over input data. Benchmark results consist of the impact of the VM on the operating system in the additional memory required to include it. For the VMs themselves, the execution speed and the size of compiled applications loaded into the VM is measured.

5.3.2 NETWORKED BENCHMARK SETUP

Next, a setup involving a simple IoT networked application as case study is constructed. The VM hosts high-level logic, and this loaded application is updatable over the network. The functionality mimics an application used in prior work [28], using small JavaScript run-time containers hosting application code on top of RIOT. The hosted logic has access to both the high-level sensor interface (called SAUL) and the CoAP stack of RIOT. The VM execution is triggered by a CoAP request and the operating system expects a formatted CoAP response payload or an error code from the application loaded in the VM. The goal is to load an application into the VM that, when triggered by a CoAP request, reads a sensor value and constructs a full CoAP payload as response to the requester.

5.3.3 VIRTUAL MACHINE MEMORY REQUIREMENT

Using our experimental setup, an initial set of measurements comparing rBPF and WASM3 are carried out. With each prototype, the performance of VM logic when it hosts the same Fletcher32 checksum is measured. While this example is specific and artificial, it is a good guinea pig to get an idea of what to expect in general. The Fletcher32 checksum algorithm requires a mix of mathematical operations memory reads and branches, containing a loop over input data.

First and foremost as visible in Table 5.3, observe that the Flash memory footprint of the interpreter WASM3 is 15 times bigger than the rBPF interpreter. To get a perspective: relatively to the whole firmware image (assuming simple business logic and a CoAP/UDP/6LoWPAN network stack) adding an rBPF VM represents negligible Flash memory overhead (less than 10 % increase), whereas adding a Wasm VM more than doubles the size of the firmware image.

	ROM size	RAM size
WASM3 Interpreter	64 KiB	85 KiB
rBPF Interpreter	4 364 B	660 B
Host OS Firmware (without VM)	52 760 B	14 856 B

Table 5.3: Memory requirements for WASM3 and rBPF interpreters.

5.3.4 APPLICATION SIZE COMPARISON

To compare the application sizes and thus the data that needs to be transferred over the network for an update, the size of a small native software module is measured. This is compared against the payload data size transferred over the network when the hosted VM logic is updated. With the results as in Table 5.4, it is visible that Wasm script size seem somewhat smaller than rBPF script size (approximately 30 % less in this case). The native C compilation shows the size of the code if the library is compiled into the device firmware itself and is not network updatable without extra measures.

Next, the penalty in terms of execution time for VM logic is compared. The performance of Fletcher32 computation on a sample input string of 361 B, with each VM is compared. The measurements show that execution is longer with the rBPF VM, than with the Wasm VM (2 × longer). Both VMs perform significantly slower than native execution, with WASM3 approximately 35 times slower and rBPF around 70 times slower. However, in terms of instructions, rBPF still enables 1.3M instructions per seconds — a fair performance for a low-power IoT device which generally is not required to process ultra-high data throughput.

Based on these preliminary measurements, it can be concluded that rBPF seems to offer acceptable performance in general, and in particular a very substantial advantage in terms of Flash memory footprint compared to Wasm. Hence, a VM approach based on rBPF seemed promising, and the prototype is fleshed out further, to perform additional experiments with IoT use-cases involving a CoAP network stack.

	code size	time
Native C	74 B	27 μs
WASM3	322 B	980 μs
rBPF	456 B	1 923 μs

Table 5.4: Size and performance of different targets for the Fletcher32 algorithm.

5.3.5 RBPF WITH LOGIC INVOLVING IOT NETWORKING

A use-case described in prior work [28] is reproduced, whereby high-level logic involving CoAP networking is executed by the VM. More precisely, the performance the hosted code shown in Listing 5.1 is evaluated. The application requests a measurement value from the first sensor and stores the value in a CoAP response. The functions called here are provided by the host operating system and exposed to the VM. Implemented are the CoAP bindings as well as the bindings to the high-level sensor (SAUL) interface as depicted in Figure 5.1.

```

1 int coap_resp(bpf_coap_ctx_t *gcoap)
2 {
3     /* Find first sensor */
4     bpf_saul_reg_t *sens = bpf_saul_reg_find_nth(1);
5     phydat_t m; /* measurement value */
6
7     if (!sens ||
8         (bpf_saul_reg_read(sens, &m) < 0)) {
9         return ERROR_COAP_INTERNAL_SERVER;
10    }
11
12    /* Format the CoAP Packet */

```

```

13  bpf_gcoap_resp_init(gcoap, COAP_CODE_CONTENT);
14  bpf_coap_add_format(gcoap, 0);
15  ssize_t pdu_len = bpf_coap_opt_finish(gcoap,
16                                     COAP_OPT_FINISH_PAYLOAD);
17
18  /* Add the sensor as payload */
19  uint8_t *payload = bpf_coap_get_pdu(gcoap);
20  pdu_len += bpf_fmt_s16_dfp(payload, m.val[0],
21                             m.scale);
22  return pdu_len;
23 }

```

Listing 5.1: Example networked sensor read application

5.3.6 APPLICATION FLASH REQUIREMENT

When compiled, the size of the bytecode is 296 B. The overhead of the full script execution, including the execution of the function calls, is 94 μ s. The additional overhead caused by the VM is negligible, when compared to network latencies of several milliseconds.

The size of the full firmware image is 69 KiB, including the rBPF interpreter. While the Flash memory required for the core rBPF interpreter is identical to the previous example (see Table 5.3), there is however an 80 B increase in Flash size due to the additional bindings to the CoAP and sensor interfaces. The RAM requirements are increased by 16 B for an additional memory access region, used to allow access to the CoAP packet.

5.3.7 RUNTIME MEMORY REQUIREMENT

Here, as an additional point of comparison, can refer to similar logic hosted in a small embedded JavaScript run-time container with RIOT bindings, studied and measured in [28] on similar hardware (a Arm Cortex-M microcontroller). These measurements show that similar logic requires 156 KiB for the JavaScript engine, on top of the 59 KiB used by RIOT, and the hosted code (script) size which was around 1 KiB. Note furthermore that these JavaScript containers did not specific memory isolation guarantees, as does rBPF. It can thus be concluded that rBPF offers much better prospects than embedded JavaScript run-time containers too, in terms of memory requirements, hosted logic size and network traffic overhead required to transmit VM updates.

5.4 DISCUSSION

5.4.1 INHERENT LIMITATIONS WITH A VM

By construction, a VM causes execution overhead by interpreting instructions. In turn this increases power consumption with logic executed within the VM instead of native execution. Measuring the full impact of the VM on power consumption is a complex task. However, this impact of increased power consumption is mitigated by two factors. On one hand, depending on the characteristics of the logic executed in

the VM, this overhead may be negligible. For instance, in this setup, The VM is geared towards hosting simple scripts implementing short decision steps rather than lengthy bulk data processing. In such cases, the additional power consumed is not substantial when compared with native execution. On the other hand, smaller script size decreases drastically the energy needed otherwise to transfer software updates, instead of a full firmware update, only the VM application has to be updated.

5.4.2 DECREASING WASM RAM USAGE

One limitation that was hit with WebAssembly is the relatively large RAM requirements: 64 KiB memory pages increment is excessive in the field of low-power microcontrollers. For this reason, based on these measurements, it cannot be concluded yet on how useful Wasm really is for low-power IoT. Also note that the WASM3 interpreter adds an intermediate compile step increasing speed, which also increases the RAM usage by another 10 KiB. Excluding this step in an interpreter can trade a reduction in execution speed performance for reduced memory consumption. An implementation more geared towards embedded applications might be able to reduce the RAM requirements. A next step here could also be to try out other interpreters such as for instance Wasm-micro-runtime [52] and WARDuino [81].

5.4.3 IMPROVING RBPF EXECUTION TIME OVERHEAD

If execution time overhead really becomes an issue, then going back to design a VM from scratch, not restricted to a software-only solutions, and use hardware MPU or even an MMU as base for memory protection. A more advanced step is to translate the executed instructions ahead-of-time into native instruction on the embedded device. Adding such an intermediate transpilation technique to RBPF (similar to what is used by WASM3) and translate the raw eBPF instructions to a format more suitable for direct consumption on the system can significantly reduce the execution time overhead. Either of these enhancements however will increase the memory requirements on the host by the added complexity.

5.4.4 DECREASING RBPF SCRIPT SIZE OVERHEAD

The RBPF VM implementation is designed as a secure sandbox for running untrusted code on small embedded devices while adhering to the already defined eBPF ISA. It can be seen from the application script sizes that the current implementation are relative big compared to applications compiled to WebAssembly bytecode. As the eBPF instructions are fixed in size and can contain a lot of unused bit fields depending on the exact instruction, compressing them with well known algorithms can reduce this downside. Initial measurements show that Heatshrink [27], an LZSS-based [159] compression library suitable for small embedded systems, can reduce the application size

by 60 % depending on the application surpassing similar WebAssembly applications.

5.4.5 EXTENDING RBPF SANDBOXING GUARANTEES

The current use case of rBPF lies in short term execution of business logic and debug applications. However the current VM design does not limit the actual execution time of the application, a virtualised application can keep the system busy without limitations, possibly draining the battery of the IoT device. A potential next step could be to limit the CPU time a single invocation of the virtual machine can occupy, further limiting the potential harm untrusted code can inflict on the device.

5.5 CONCLUSION

In this chapter, I present the design of a minimal VM, implemented and studied experimentally against a second VM implementation, both targeting low-power, microcontroller-based IoT devices. rBPF is a register-based VM hosed in RIOT, and an interpreter based on Linux's extended Berkeley Packet Filters. I compare the performance, experimentally on commercial IoT hardware, to an approach hosting high-level logic in an embedded WebAssembly virtual machine. With the benchmarks I show that, compared to WebAssembly and to prior work on small run-time containers for interpreted logic, hosting rBPF VMs requires an overhead of $\approx 10\%$ of flash usage for a typical IoT application. When compared to the $\approx 200\%$ extra flash usage required by Wasm implementations, rBPF is much more attractive.

As presented here, rBPF can be used to isolate small applications inside the VM. It is a promising approach to host and isolate small software modules, with acceptable execution time overhead and without any reliance on specific hardware. It shows minimal memory overhead of $\approx 10\%$ for a typical IoT application.

The future direction for rBPF is to use it as core VM for a rich multi-tenant environment around the sandbox. The minimal footprint of rBPF with a trusted environment must be extended to an environment in which different stake holders can adjust the functionality of a pre-programmed device in a reliable manner without having to update the whole firmware.

CHAPTER 6

SANDBOXED FUNCTION EXECUTION FOR DEVOPS-STYLE RECONFIGURATION OF CONSTRAINED DEVICES

The capability to run applications inside a **VM** and virtualise these applications can be leveraged further. The applications can be bundled into small software components and isolated from the operating system. The operating system can be enhanced to offer rich facilities to the applications inside the **VM**. Extrapolating this, the result is a Functions-as-a-Service-like environment specifically geared towards microcontrollers.

Such a **FaaS**-like environment provides an environment where small event-triggered functions can be run, providing the device operator with debug or business logic on the device. This logic can be update as a single module without having to update the full firmware on the device. Furthermore, because of the isolated nature of the **VM**, the function runs in a fault-tolerant environment. It is not possible for the function to directly influence the memory of the operating system, only through bindings exposed by the operating system.

In this chapter, rBPF is further extended into Femto-Container to provide a rich and secure **FaaS**-like environment for microcontrollers. The design of Femto-Container, together with benchmarks and comparison against other environments is presented.

The work is published before as “Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers” [2], presented in Middleware 2022 – 23rd ACM/IFIP International Conference Middleware.

6.1 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 this threat model, both malicious tenants which can deploy malicious

code and malicious clients which can maliciously interact with deployed code are considered.

6.1.1 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 does not 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.

6.1.2 MALICIOUS CLIENT

The malicious client does not 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.

6.1.3 ATTACK VECTORS

A number of attack vectors are considered to be in scope for the sandbox used in Femto-Container 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 constrained 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.

Another attack vector that can be considered is resource exhaustion of resources on the host system itself. These can be attacked from a sandbox environment itself or via a different surface. While these attacks can cause harm to the system, for example to cause excessive battery drain, the responsibility to protect against these attacks is with the operating system and not for the sandbox environment, and therefor out of scope of this work.

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

6.2 EMBEDDED RUNTIME ARCHITECTURE DESIGN

In this section, Femto-Containers is introduced, 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 deployment and execution of small logic modules. These modules, or functions, are hosted on top of a middleware offering isolation and abstraction with respect to the underlying OS and hardware. By combining isolation and hardware/OS abstraction, the crucial properties of FaaS runtimes are retained: code mobility and security. Differently from typical FaaS runtimes, however, Femto-Containers must be able to interact with specific hardware (such as sensor/actuators), and must drastically reduce the scope and the cost of virtualisation to operate within the constraints IoT hardware. The Femto-Container architecture therefore relies on ultra-light weight virtualisation, as well as on a set of assumptions and features regarding an underlying RTOS, defined below.

6.2.1 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 [30] or FreeRTOS [20] and others [83]. 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.

6.2.2 NO ASSUMPTIONS ON MICROCONTROLLER HARDWARE

To retain generality, the aim is 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 [135].

6.2.3 USE OF ULTRA-LIGHTWEIGHT VIRTUALISATION

The virtual machine (VM) 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 VM must have a low memory footprint, both in Flash and in RAM, per instantiated VM application. This allows to run multiple VMs in parallel on the device. Note that, as the aim is to virtualise less functionalities, the VM can in fact implement a reduced virtual hardware feature set. For instance, virtualised peripherals such as an interrupt controller are not required, and remove the possibility of virtualising a full OS.

6.2.4 USE OF SIMPLE CONTAINERIZATION

A slim environment around the VM exposes RTOS facilities to the VM. The container sandboxing a VM allows this VM to be independent of the underlying operating system, and provide 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 third party module reprogramming.

6.2.5 ISOLATION & SANDBOXING THROUGH VIRTUALISATION

The OS and Femto-Containers must be mutually protected from malicious code. This implies in particular that code running in the VM must not be able to access memory regions outside of what is allowed. Here again, simple contracts can be used to define and limit memory and peripheral access of the code running in the Femto-Container.

6.2.6 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, the design mandates 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.

6.2.7 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 [7] (CBOR, COSE) to secure updates end-to-end over network paths including low-power wireless segments. Leveraging SUIT for these update payloads provides

authentication, integrity checks and roll-back 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 Universally Unique Identifier (UUID) as 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.

6.3 ULTRA-LIGHTWEIGHT VM MICRO-BENCHMARK

In this section, the performance of an initial proof of concept using RIOT [30] to host Femto-Container runtimes is compared. Different ultra-lightweight virtualisation techniques are compared: Python (MicroPython runtime), WebAssembly (WAMR runtime), eBPF (rBPF runtime) and JavaScript (RIOTjs runtime).

Experiments with Femto-Containers are run using each virtualisation candidate 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.

In these benchmarks reported on below, each Femto-Container is minimally implemented, and loaded with a VM hosting logic performing a Fletcher32 checksum on a 360 B input string. The assumption is that this computing load roughly mimics intensive sensor data (pre-) processing on-board.

Our benchmarks results are shown in Table 6.1 and Table 6.2.

6.3.1 CONSIDERING SIZE

While the size of applications are roughly comparable across virtualisation techniques (see Table 6.2) 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.

Runtime	ROM size	RAM size
WASM3	64 KiB	85 KiB
eBPF	4.4 KiB	0.6 KiB
RIOTjs	121 KiB	18 KiB
µPython	101 KiB	8.2 KiB
Bare Host OS	52.5 KiB	16.3 KiB

Table 6.1: Memory requirements for Femto-Container runtimes.

Runtime	code size	startup time	run time
Native C	74 B	–	27 μ s
WASM3	322 B	17 096 μ s	980 μ s
rBPF	456 B	1 μ s	2 133 μ s
RIOTjs	593 B	5 589 μ s	14 726 μ s
MicroPython	497 B	21 907 μ s	16 325 μ s

For comparison, the biggest ROM budget measured requires 27 times more memory than the smallest budget. Similarly, RAM requirements vary a lot. Note that it was not possible to 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. As noted in [chapter 5](#) 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 relaxed. However the RAM budget would still be well above an order of magnitude more than the lowest RAM budget as measured with rBPF.

Last but not least, to 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 considered, 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 6.1](#). For such targets, according to our measurements, adding a **VM** can either incur a tremendous increase of 200 % more ROM with MicroPython, or a negligible impact with 8 % more ROM with rBPF as visualized in [Figure 6.1](#) and [Figure 6.2](#).

6.3.2 CONSIDERING SPEED

To no surprise, beyond size overhead, virtualisation also has a cost in terms of execution speed. But here again, performance varies wildly depending on the virtualisation 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 ≈ 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 virtualisation technique, this startup time varies almost 1000 fold — from a few microseconds compared to a few milliseconds.

Table 6.2: Size and performance of Fletcher32 logic hosted in different Femto-Container runtimes.

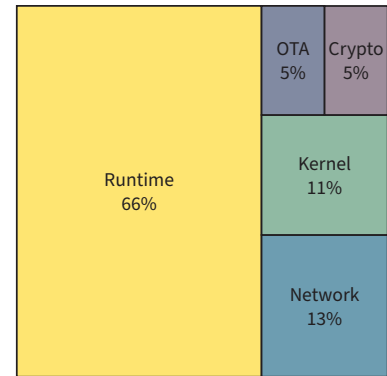


Figure 6.1: Flash memory distribution of RIOT with MicroPython Femto-Container, 154 KiB total.

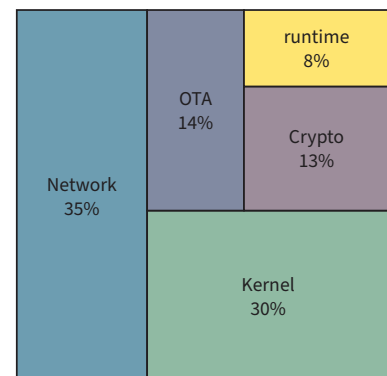


Figure 6.2: Flash memory distribution of RIOT with rBPF Femto-Container, 57 KiB total.

6.3.3 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 some potential advantages in terms of memory (pooling) efficiency, however it also has some potential drawbacks in terms of security with mutual isolation of the **VMs'** memory.

Furthermore, security guarantees call for a formally verified implementation of the hosting engine. A typical approximation is: less Lines of Code (**LoC**) means much less effort to produce a verified implementation. For instance, the rBPF implementation is ≈ 1.5 k**LoC**, while the WASM3 implementation is ≈ 10 k**LoC**. The other implementations considered in our pre-selection, RIOTjs and MicroPython, encompass significantly more **LoC**.

6.3.4 CHOICE OF VIRTUALISATION

Our benchmarks indicate that in terms of memory overhead, startup time and **LoC**, Femto-Containers using rBPF virtualisation is the most attractive, by far. Note that execution time with Femto-Containers using WebAssembly is 2x faster than Femto-Containers using rBPF. However, it is expected that a 2x factor in execution time will have no significant impact in practice, for the use cases targeted: small lightweight workloads. Since the priority is on memory footprint, the aim is ≈ 10 percent memory overhead for functionality containerization, rBPF is chosen to flesh out the concept further.

6.4 FEMTO-CONTAINER RUNTIME IMPLEMENTATION

The Femto-Container **VM** design is based on rBPF which is designed from the eBPF instruction set architecture. The instruction set itself is minimal and optimized for fast parsing with compact code. As proof of concept, the Femto-Container architecture is implemented with containers hosted in the operating system RIOT and virtualisation using an instruction set compatible with the eBPF instruction set. This implementation is open source (published in [14]). The main characteristics are shown in detail below.

6.4.1 USE OF RIOT MULTI-THREADING

Each Femto-Container application instance running on the operating system is scheduled as a regular thread in RIOT. The native OS thread scheduling mechanism can thus simply execute concurrently and share resources amongst multiple Femto-Containers and other tasks, spread over different threads. An overview of how Femto-Containers integrates in the operating system is shown in **Figure 6.3**. A Femto-Container 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-Container instance.

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

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 operating system. 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.

6.4.2 ULTRA-LIGHTWEIGHT VIRTUALISATION USING EBPF INSTRUCTION SET

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

REGISTER-BASED VM

The VM 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 a 32 bit immediate value. Position-independent code is achieved by using the reference in r10 and the offset field in the instructions.

JUMP TABLE & INTERPRETER

The interpreter parses instructions and executes them operating on the registers and stack. The machine itself is implemented as a computed jump table, with the instruction opcodes as keys. During execution, the hosting engine iterates over the instruction opcodes in the application,

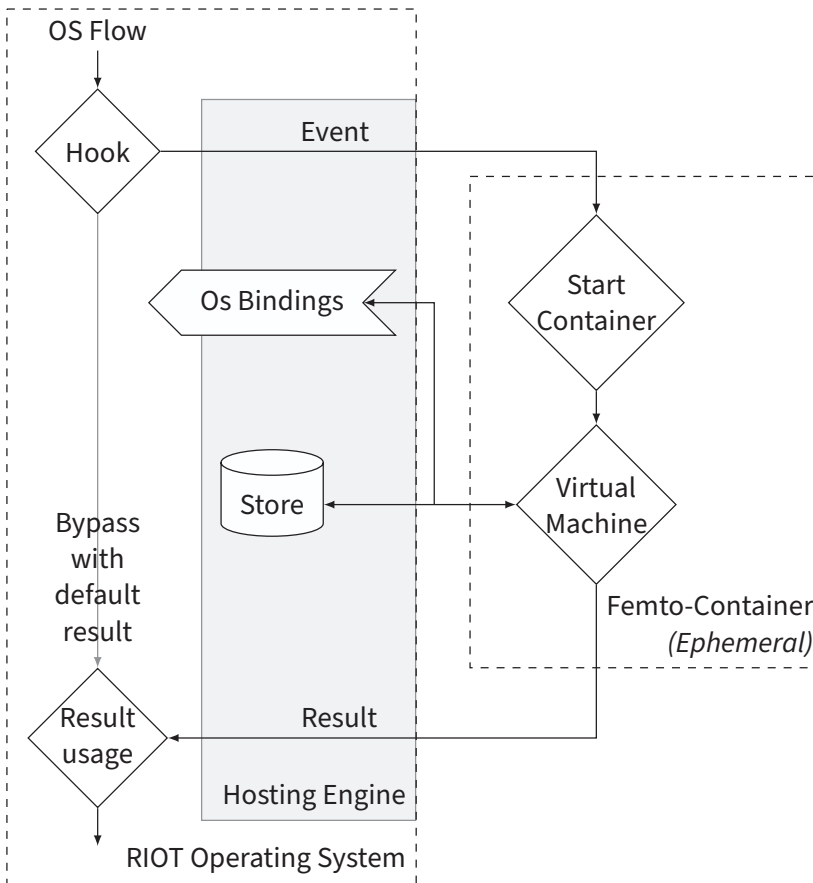


Figure 6.3: Femto-Container RTOS integration.

and jumps directly to the instruction-specific code. This design keeps the interpreter itself small and fast.

AHEAD-OF-TIME VS JUST-IN-TIME

One approach to speed up embedded execution time is to perform a translation into device-native code. One way to offload the device is to use more Ahead-of-Time (AOT) compilation and interpretation, and less Just-in-Time (JIT) processing on-device. However, using AOT pre-compiled code can both complicate run-time security checks on-board the IoT device, and reduce the portability of the code deployed on the device. For these reasons, in this section, primarily JIT is considered.

6.4.3 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 VM instance only has access to its VM-specific registers and its stack.

MEMORY ACCESS CHECKS AT RUNTIME

The allowlist 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

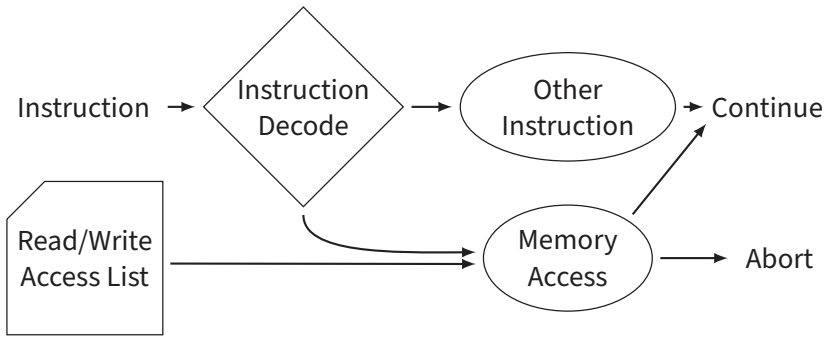


Figure 6.4: Interaction between memory instructions and the access lists.

access. For example, a firewall-type trigger can grant read-only access to the network packet, allowing the VM to inspect the packet, but not to modify it. As the memory instructions allow for calculated addresses based on register values, memory accesses are checked at runtime against the resulting address, as show in Figure 6.4. Illegal access aborts execution.

PRE-FLIGHT INSTRUCTION CHECKS

A Femto-Container verifies the application before it is 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 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.

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: $N_i \times N_b$.

6.4.4 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.

```
1 sched_ctx_t context = {
```

```
2     .previous = active_thread,  
3     .next = next_thread,  
4 };  
5  
6 int64_t result;  
7  
8 fc_hook_execute(BPF_HOOK_SCHED, &context,  
9                sizeof(context), &result);
```

Listing 6.1: Example hook implementation.

An example of a hook integrated in the firmware is shown in [Listing 6.1](#). The firmware has to set up the context structure for the Femto-Containers after which it can call the hosting engine to execute the containers associated with the hook.

6.5 USE-CASE PROTOTYPING WITH FEMTO-CONTAINERS

In this section, the programming model exposed by Femto-Containers is described. Furthermore Femto-Containers is used to prototype the implementation of several use cases involving one or more applications, hosted concurrently on a microcontroller, matching targets identified initially. Where multiple function are involved, these are hosted concurrently on a single microcontroller.

6.5.1 PROGRAMMING MODEL

In the prototype implementation shown below, C is used to code logic hosted in Femto-Containers. However, any other language compiled with LLVM could be used instead such as C++, Rust and TinyGo.

Inherent limitations due to the eBPF instruction set, combined with the absence of virtualised hardware, restrict what logic can be deployed in Femto-Containers currently. Femto-Containers are designed to host logic that is rather script-like, short-lived, and not computation-intensive. On the one hand, such characteristics increase security-by-design. On the other hand they reduce flexibility. For instance, asynchronous operation is not supported: there is no option to interrupt the control flow inside a Femto-Container from outside the **VM**. Another limitation is 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. Allowing the application to request more stack from the RTOS, for example via the contracts, could solve part of 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.

6.5.2 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 6.2](#). A small C structure 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.

```

1 #include <stdint.h>
2 #include "bpf/bpfapi/helpers.h"
3
4 #define THREAD_START_KEY    0x0
5
6 typedef struct {
7     uint64_t previous; /* previous thread */
8     uint64_t next;     /* next thread */
9 } sched_ctx_t;
10
11 int pid_log(sched_ctx_t *ctx)
12 {
13     /* Zero pid means no next thread */
14     if (ctx->next != 0) {
15         uint32_t counter;
16         uint32_t thread_key = THREAD_START_KEY +
17                             ctx->next;
18         bpf_fetch_global(thread_key,
19                         &counter);
20         counter++;
21         bpf_store_global(thread_key,
22                         counter);
23     }
24     return 0;
25 }

```

Listing 6.2: Thread counter code.

6.5.3 NETWORKED SENSOR CODE EXAMPLE

For the second prototype two Femto-Containers are added 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 6.5](#). 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 [Listing 6.3](#).

```

1 #include <stdint.h>
2 #include <stdbool.h>
3 #include "bpf/bpfapi/helpers.h"
4
5 #define SHARED_KEY 0x50
6 #define AVERAGING_LEN 10
7 #define PERIOD_US (1000 * 1000)
8
9 static uint32_t _average(uint32_t *values)
10 {
11     uint64_t sum = 0;
12     for (size_t i = 0;
13         i < AVERAGING_LEN;

```

```
14     i++) {
15         sum += values[i];
16     }
17     return sum / AVERAGING_LEN;
18 }
19
20 int measurement(void *conf)
21 {
22     uint32_t last_wakeup = bpf_ztimer_now();
23     uint32_t counter = 0;
24     size_t pos = 0;
25     bool initial = true;
26
27     uint32_t values[AVERAGING_LEN];
28
29     while (1) {
30         /* Read sensor value from sensor */
31         bpf_saul_reg_t *sensor;
32         phydat_t measurement;
33
34         /* Periodic blocking sleep */
35         bpf_ztimer_periodic_wakeup(&last_wakeup,
36                                   PERIOD_US);
37
38         /* Find first sensor */
39         sensor = bpf_saul_reg_find_nth(1);
40
41         /* Abort if the sensor is
42            not available */
43         if (!sensor ||
44             (bpf_saul_reg_read(sensor,
45                               &measurement) < 0))
46             continue;
47
48     }
49
50     uint32_t value = measurement.val[0];
51
52     if (initial) {
53         /* Fill array with the
54            initial measurement */
55         for (size_t i = 0;
56              i < AVERAGING_LEN;
57              i++) {
58             values[i] = value;
59         }
60         initial = false;
61     }
62     else {
63         values[pos] = value;
64         pos++;
65         if (pos >= AVERAGING_LEN) {
66             pos = 0;
67         }
68     }
69
70     uint32_t average =
71         _average(values);
```

```

72         bpf_store_global(SHARED_KEY,
73                         average);
74     }
75
76     /* Unreachable */
77     return 0;
78 }
79

```

Listing 6.3: Femto-Container Sensor readout process.

The second container's logic is triggered upon receiving a network packet (CoAP request), and returns the stored sensor value back to the requester. The code for this logic is shown in [Listing 6.4](#).

```

1  #include <stdint.h>
2  #include "bpf/bpfapi/helpers.h"
3
4  #define SHARED_KEY 0x50
5  #define COAP_OPT_FINISH_PAYLOAD (0x0001)
6
7  typedef struct {
8      uint32_t hdr_p;      /* ptr to raw packet */
9      uint32_t token_p;    /* ptr to token */
10     uint32_t payload_p;   /* ptr to payload */
11     uint16_t payload_len; /* length of payload */
12     uint16_t options_len; /* length of options */
13 } bpf_coap_pkt_t;
14
15 int coap_resp(bpf_coap_ctx_t *gcoap)
16 {
17     bpf_coap_pkt_t *pkt = gcoap->pkt;
18     /* Track executions */
19     uint32_t counter;
20     bpf_fetch_global(SHARED_KEY, &counter);
21
22     char stringified[20];
23     size_t str_len = bpf_fmt_u32_dec(stringified,
24                                     counter);
25
26     /* Format the packet with a 205 code */
27     bpf_gcoap_resp_init(gcoap, (2 << 5) | 5);
28     /* Add Text type response header */
29     bpf_coap_add_format(gcoap, 0);
30     ssize_t pdu_len = bpf_coap_opt_finish(gcoap,
31                                         COAP_OPT_FINISH_PAYLOAD);
32
33     uint8_t *payload =
34         (uint8_t*)(intptr_t)(pkt->payload_p);
35
36     if (pkt->payload_len >= str_len) {
37         bpf_memcpy(payload, stringified,
38                   str_len);
39         return pdu_len + str_len;
40     }
41
42     return -1;
43 }

```

Listing 6.4: Femto-Container CoAP Endpoint.

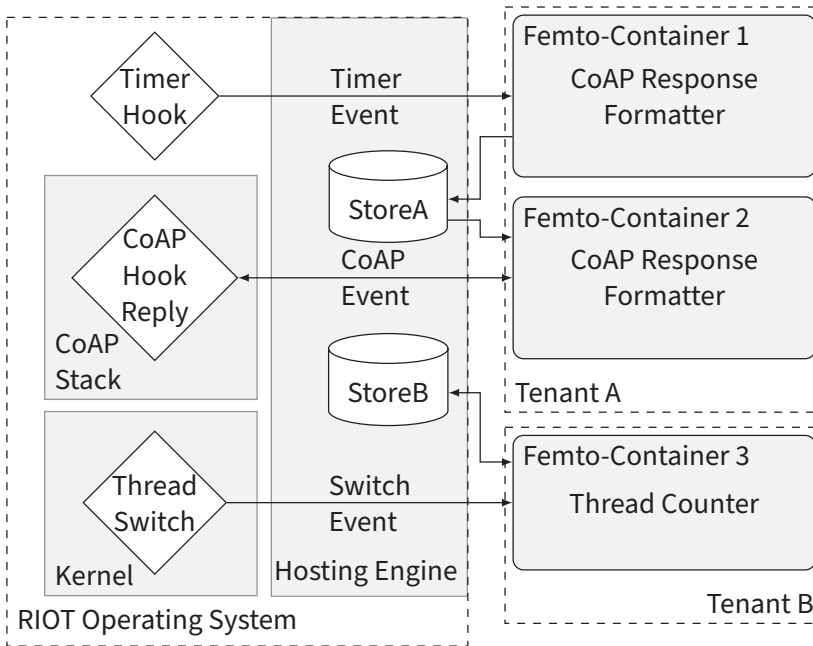


Figure 6.5: Event and value flow when hosting multiple containers for different tenants.

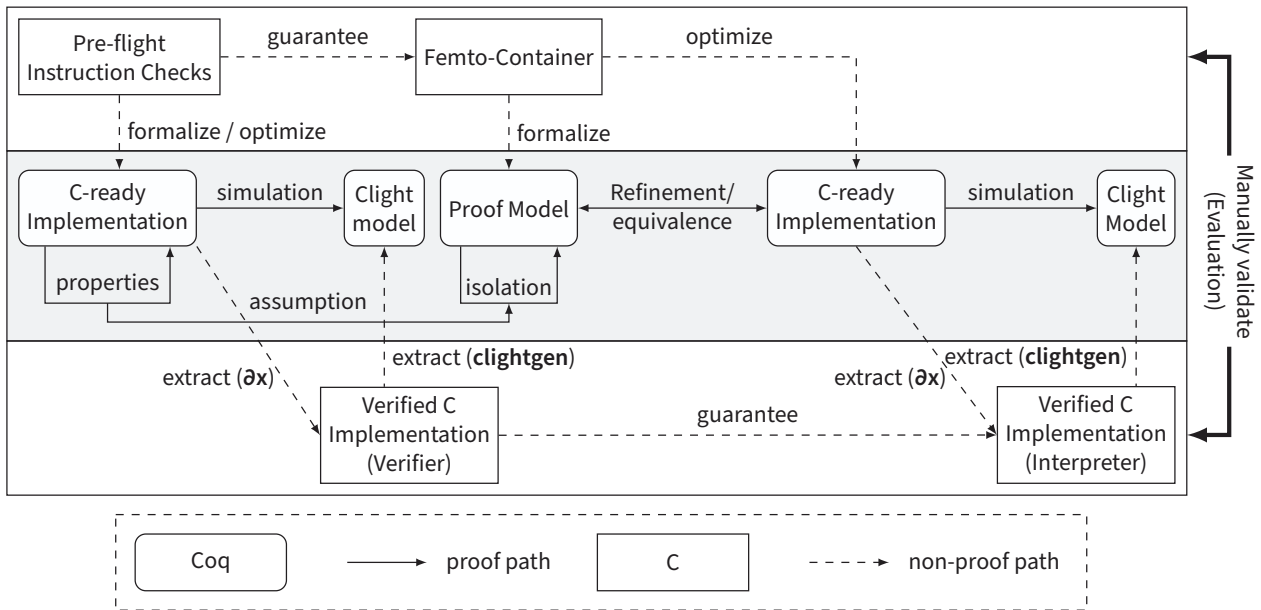


Figure 6.6: Certified Femto-Container (CertFC) Formal verification workflow.

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 possible (between sensor value reading/processing on the one hand, and on the other hand the communication between the device and a remote requester).

6.6 FORMAL VERIFICATION

The critical components of Femto-Containers in terms of cyber-security 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), a formally verified implementation of these components could be produced. This runtime, called Certified Femto-Container (**CertFC**), which uses the formally verified interpreter and checker.

6.6.1 TARGETED REQUIREMENTS FORMALIZATION.

The security guarantees provided Femto-Containers with are essentially memory and fault isolation. More precisely, the aim was to prove it impossible for **CertFC** to access a memory location out of its application'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 **VM**.

6.6.2 FORMAL VERIFICATION APPROACH.

The Coq proof assistant was used to mechanically and exhaustively verify these requirements by employing the multi-step design workflow depicted in **Figure 6.6**:

1. First, a proof model and a C-ready implementation that formalize and optimize the native, vanilla, C implementations of the rBPF verifier and **VM** in RIOT was provided. 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 [106].
 - (b) the pre-flight runtime checks of the verifier.
 - (c) the defensive runtime checks of the **VM** itself for numerical and memory operations.
3. The verified C implementation is automatically extracted from the C-ready Coq model using the *dx tool* [100]. Based on a set of formalized translation rule from Coq to C, *dx* 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 [23]. Second, proving that Clight model to simulate the C-ready model using translation validation [136].

The pre-flight checks of the verifier are directly formalized by a C-ready implementation in Coq because of its simplicity. By contrast, the interpreter is first formalized by a proof model in Coq that defines the formal syntax and semantics of the rBPF ISA. The Coq specification is then refined into an optimized (yet semantically equivalent) C-ready implementation in Coq, for the purpose of extracting C code using ∂x . This refinement/optimization principle allows to 1) prove the native optimizations correct, 2) improve the performance of the extracted code and, 3) facilitate the extracted code review and validation with system designers. Pre-flight checks, subsection 6.4.3, define rudimentary guarantees for applications to run on the VM. The formalization of these guarantees also defines necessary pre-conditions to the verification of fault-isolation, i.e. the guarantee `verifier_inv` of the verifier C-ready model is used as assumption by the proof model of the interpreter. Combined with the registers invariant `register_inv` and the memory invariant `memory_inv`, it yields the proof of software-fault isolation of **CertFC** in the Coq proof assistant, that is, the isolation of all transitions to a crash state using runtime safety checks, hence the impossibility of an undefined behavior. The Coq theorem `inv_ensure_no_undef` states that our model satisfies a software fault isolation property where `st` is a **CertFC** state and `fuel` is used to enforce finite execution.

Table 6.3 provides statistics on the complete specifications and proofs of **CertFC**. The proof model of the interpreter consists of 2.4k lines of Coq code. The C-ready implementation model of the verifier & interpreter is approx. 4.7k long. The proof of the VMs properties (e.g. isolation) exceeds 5.4k and the refinement/equivalence theorem is completed by a 0.6k proof. The final step includes 23.5k of translation validation proofs between the Coq specification and the extracted Clight model. The last part of Figure 6.6 is the manual validation between the native C code and the verified implementation,

6.7 PERFORMANCE EVALUATION

In this section the performance of Femto Containers is evaluated and compared 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.

6.8 HOSTING ENGINE ANALYSIS

The Femto-Container implementation is benchmarked on a number of aspects. First, the footprint of the hosting engine on the embedded device is compared. This shows the impact of adding Femto-Containers to the applications. Second, the execution time of a number of individual instructions is compared. This shows the difference in computational overhead between the different implementations.

Table 6.3: Code statistics of **CertFC**

	LoCs
Proof Model	2 445
C-ready Implementation	4 744
Properties	5 432
Equivalence	635
Simulation	23 564
(Total)	36 820

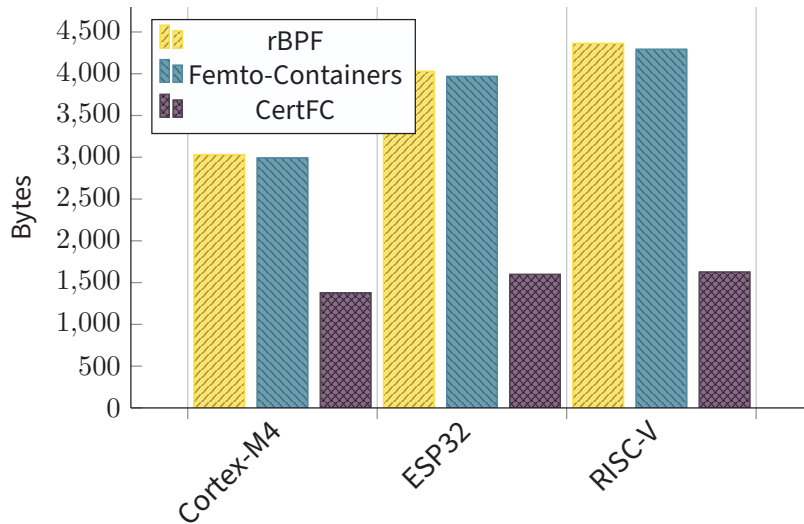


Figure 6.7: Flash requirement for the different implementations and platforms

To compare the impact of adding the Femto-Containers to an existing firmware, the memory footprint of the different implementations is measured and compared. In general, each Femto-Container needs memory to:

- Store the application bytecode.
- Handle the **VM** state and stack.

The impact on the required flash on the firmware is shown in [Figure 6.7](#) and [Table 6.4](#). 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 **VM** in the context memory structure and not on the thread stack.

The different implementations of Femto-Containers are compared in [Figure 6.8](#) against a set of eBPF instructions, showing that the rBPF extensions incur minimal overhead on the **VM** 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.

	ROM size	RAM size
Femto-Containers	2 992 B	624 B
rBPF	3 032 B	620 B
CertFC	1 378 B	672 B

Table 6.4: Memory footprint of a Femto-Container hosting minimal logic on Arm Cortex-M4.

6.9 EXPERIMENTS WITH A SINGLE CONTAINER

In this section, the execution times of a number of Femto-Container applications are shown. This shows the applicability of Femto-Containers in the different scenarios. The execution times are shown in [Figure 6.11](#).

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

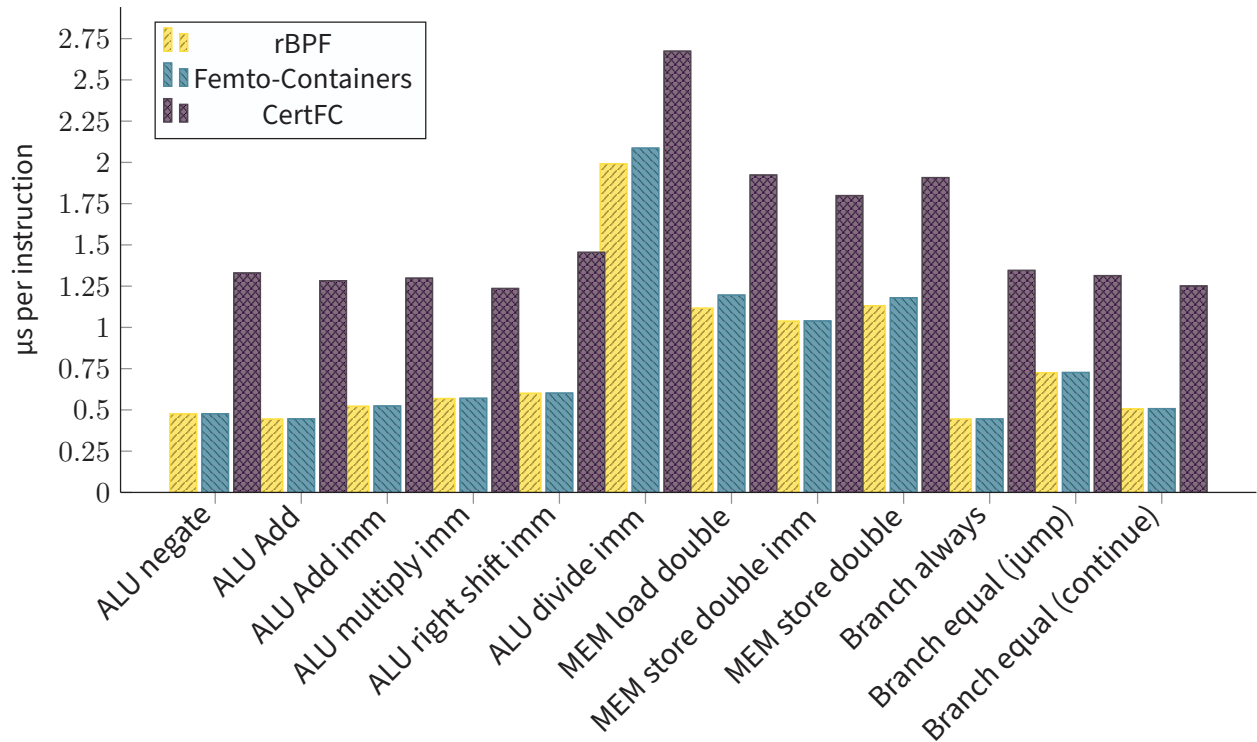


Figure 6.8: Time per instructions on the Cortex-M4 platform

Femto-Containers the duration of this application is long.

The second example shown is the thread counter example previously shown in Listing 6.2. 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 from Listing 6.4. 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 does not impact the performance of the overall system.

6.9.1 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 VM stack. The other requirement is that the microcontroller must have a large enough storage for the all the application images.

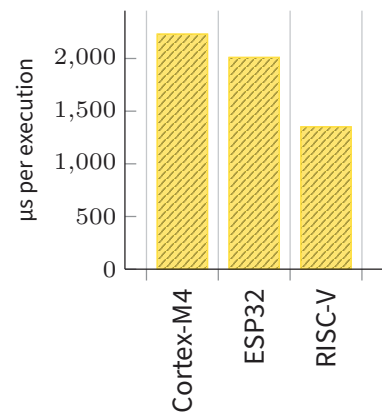


Figure 6.9: Fletcher32 checksumming algorithm application.

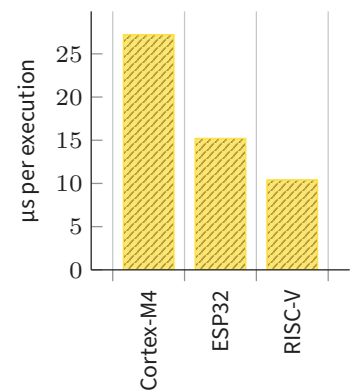


Figure 6.10: Thread log example application.

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.

The results are shown in Table 6.5. For example, the CoAP handler container, as described by Figure 6.5, requires additional read/write permissions to two memory regions to handle the CoAP packet, which increases the RAM overhead by 16 B per region.

Next, the memory required to concurrently host multiple containers from multiple tenants on the same microcontroller is measured, using the examples described in section 6.5. As shown previously in Table 6.4, the minimal default memory footprint used by a Femto-Container amounts to 624 B, which is for storing the VM stack, housekeeping memory structures 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 housekeeping for different tenants was 340 B. Hence, the required RAM memory measured so as to run the example with 3 containers and 2 tenants is 3.2 KiB. Beyond these examples, when increasing the number of containers hosting larger applications (e.g. $\approx 2\,000$ 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.

	Bytecode	Container RAM	Total RAM
Thread Counter	104 B	664 B	768 B
Sensor Reader	496 B	664 B	1 160 B
CoAP Handler	264 B	696 B	960 B

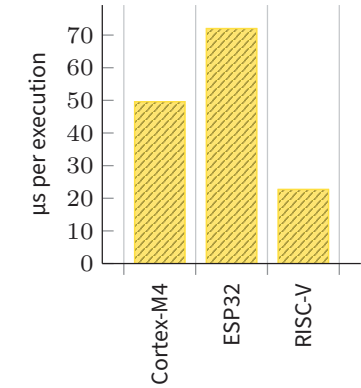


Figure 6.11: CoAP response formatter application.

Table 6.5: RAM required to host multiple concurrent Femto-Containers applications.

	RIOT	Hook	Application	Hook Overhead
Cortex-M4	427	645	3499	218
ESP32	1607	1773	2325	166
RISC-V	573	784	1508	211

Table 6.6: Thread switch performance in clock ticks

6.10 OVERHEAD ADDED BY HOOKS

One key question is how performance is affected by pre-provisioning launchpads (hooks) in the RTOS firmware. In Table 6.7 the overhead caused by adding a hook to the RTOS workflow is measured. This overhead amounts to ≈ 100 clock ticks on all the hardware 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 percent of the number of cycles needed to execute

the logic hosted in a Femto-Container. From this observation, it can be concluded 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.

6.11 DISCUSSION

6.11.1 VIRTUALISATION VS POWER-EFFICIENCY

Inherently, virtualisation 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 negligible, *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.

6.11.2 CONTROLLING TENANT PRIVILEGES

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 pre-provisioned 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 privileges that can be granted, the container specifies the set of privileges 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 privileges possible per hook. In case 2 tenants have different privileges, a second hook must be made available. Additional mechanisms would be required to overcome this limitation and/or to enable dynamic privilege levels.

6.11.3 INSTALL TIME VS EXECUTION TIME

As mentioned before, one limitation due to virtualisation is the inherent slump in execution speed, compared to native code execution. 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.

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

Table 6.7: Hook overhead in clock ticks for the thread switch example

6.11.4 TENANT-LOCAL STORAGE OF VALUES

Currently Femto-Containers distinguish between a container-local value store and a fully device-global value store. This becomes a limitation when a single tenant needs to share values between VMs, but needs to keep them private from other tenants. Resolving this issue requires another level of separation in the value store. A tenant-specific store that tracks values shared between VMs would be sufficient. Another option is to specify to which value in the store each VM has access to. This would allow sharing values with a very specific set of VMs, while retaining mutual isolation and granting the exact minimal required permissions.

6.11.5 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 6.3, looping on a 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.

6.11.6 FIXED- VS VARIABLE-LENGTH INSTRUCTIONS

Originally, eBPF scripts are optimized for fast execution on 64-bit platforms. Compared to other VMs 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.

6.12 CONCLUSION

In this chapter, I introduce Femto-Containers, a new designed middleware runtime architecture, which enables **FaaS** capabilities embedded on heterogeneous low-power **IoT** hardware. Using Femto-Containers, authorized, third party maintainers of **IoT** software can deploy and manage via the network mutually isolated software modules embedded on a microcontroller-based device. I provide 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. A formally verified variant of the **VM** engine is provided with 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. I then demonstrated experimentally the performance of the Femto-Container runtime on the most common 32-bit microcontroller architectures: Arm Cortex-M, RISC-V, ESP32. I 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.

Femto-Containers is suitable for multiple scenarios and environments in which constrained microcontroller-based devices are used. For example, Femto-Containers can be leveraged in a nanosatellite environment in which measurement and experiment logic is hosted inside the **VM**. Together with the SUIT manifest, Femto-Containers provides a rich **FaaS**-like environment which can be updated on-demand. This allows for hosting mission code on the satellite, which can be modified over-the-air, when it must evolve during the life-time of the satellite. In **chapter 7**, this scenario is presented as use case with SUIT as update mechanism to deploy hosted logic on CubeSat payloads.

CHAPTER 7

CASE STUDY: SECURE SOFTWARE RECONFIGURATION ON NANOSATELLITE

The previous chapters described the different building blocks to design a secure reconfigurable system. One use case for these building blocks lies within the CubeSat payloads. CubeSats provide tiny rack slots, 0.25 U each, for low-power payloads hosted on the CubeSat.

As the lifetime of these CubeSats is 5 to 10 years, there is a need for updates and reconfigurability of the firmware on these satellite payloads. In this chapter the ThingSat payload for CubeSats is described, a real-world use case where building blocks of previous chapters is used. Based on ThingSat, the Cubedate framework for updating firmware and mission control modules is described. The LEO environment in which these CubeSats reside, together with the limited power and network links, provides a constrained environment requiring careful design of these components.

This work is based on “Cubedate: Securing Software Updates in Orbit for Low-Power Payloads Hosted on CubeSats”[6] as presented at the 12th IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks.

7.1 THINGSAT

7.1.1 SYSTEM ARCHITECTURE DESIGN

Figure 7.1 describes the ThingSat deployment components. It gives an overview of a typical CubeSat ecosystem, whereby the interaction with this payload traverses untrusted elements.

LOW-POWER SPACE SEGMENT

The Low-power Space Segment comprises the On-Board Computer (OBC) and hosted payloads, whom, interconnected via a Controller Area Network (CAN) bus, which share resources on the CubeSat.

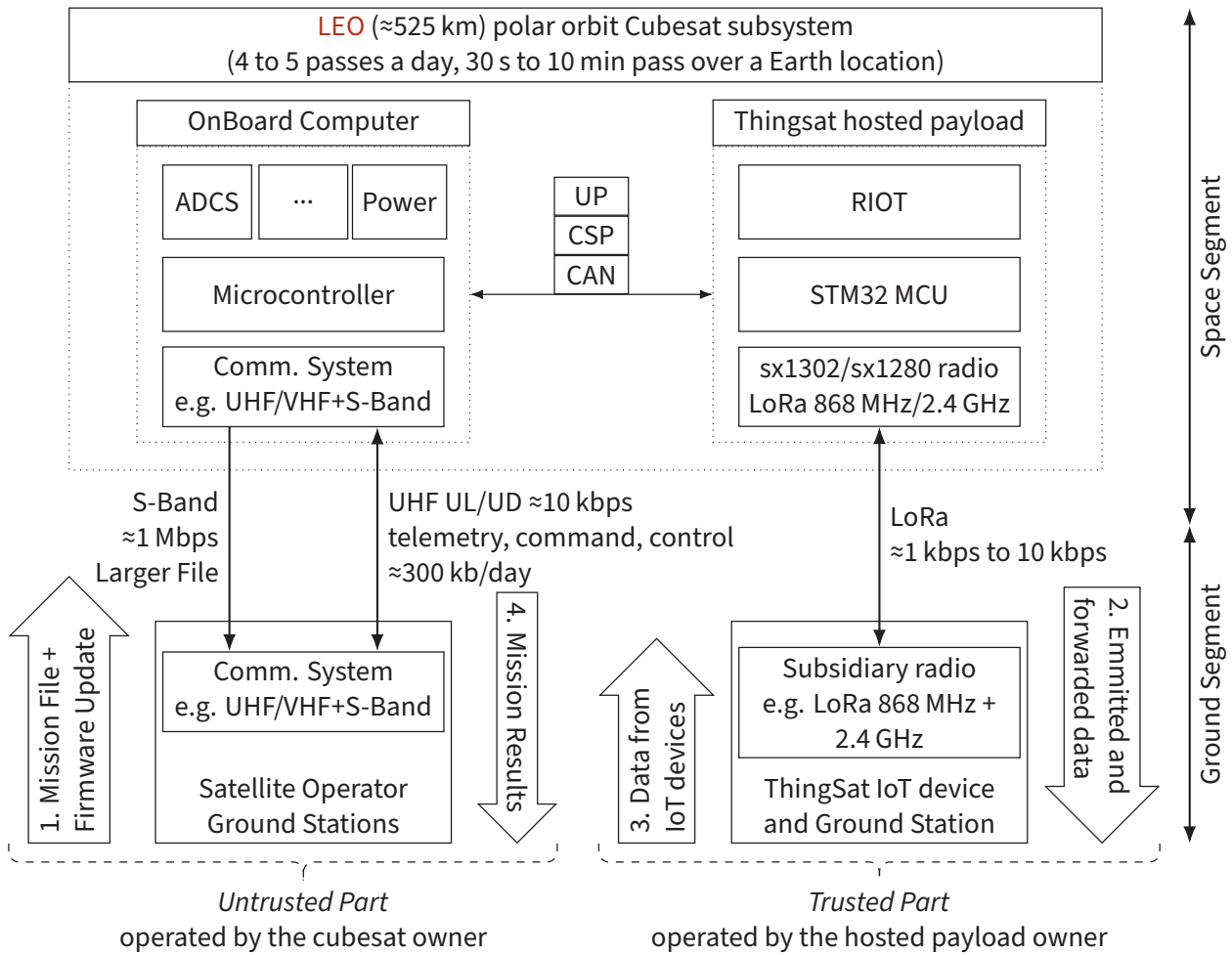


Figure 7.1: ThingSat hosted payload: deployed components and architecture.

The **OBC** provided by the satellite operator consists of a microcontroller with all its subsystems to operate the CubeSat: Attitude Determination and Control System (**ACDS**), communication subsystems (UHF/VHF/S-band for uplink/downlink and antennas) and power subsystem (Battery Management, Energy Harvesting with Solar Panels, Auxiliary Power Supply).

The ThingSat payload designed for this use case is build around an STM32F405RG microcontroller featuring an ARM Cortex-M4 core and open source firmware based on RIOT [30]. It embeds both a **SX1302 transceiver** for communications on the 868MHz band and a **SX1280 transceiver** for communications on the 2.4GHz band. Furthermore a corresponding dual-band patch antenna is designed. When active and using the 868MHz band, the ThingSat payload consumes at 3.3 V:

- 90 mA in standby,
- 110 mA during a frame reception (RX) and
- 300 mA during a frame transmission (TX) at the 27dBm maximum power.

By operating in the milliwatt range, our payload achieves low-power

consumption what could not have been achieved with a raspberry pi or without the de-facto low-power LoRa technology.

GROUND SEGMENTS

There are two ground segment elements that communicate with the ThingSat payload, the SatRevolution ground stations and custom ThingSat LoRa ground stations.

The SatRevolution Ground Stations are provided by the CubeSat operator, but not necessarily owned by the CubeSat operator, to communicate via UHF/VHF with the **OBC**, and indirectly with the payload. This can be done directly through a Command & Control Center, which acts as a broker between payload maintainers and hosted payload. This communication path provides indirect access to the payload.

The ThingSat LoRa Ground Stations provide a low cost and simple ground deployed and maintained which can communicate via LoRa directly with the ThingSat payload. These stations are based on an ESP32 microcontroller, and a 2.4GHz SX1280-LoRa transceiver, also running an open source firmware based on RIOT.

7.1.2 COMMUNICATION CHARACTERISTICS OVERVIEW

ThingSat payload communicates either directly via Low Power WAN (**LPWAN**), or indirectly via the UHF/VHF link provided by the CubeSat's OBC. Multiple avenues of communication are available for the ThingSat payload. It communicates either directly via **LPWAN**, or indirectly via the UHF/VHF link provided by the CubeSat's **OBC**.

INDIRECT COMMUNICATION CHARACTERISTICS VIA UHF/VHF

CubeSat-GS communications are typically done on amateur frequency bands (UHF/VHF) with typically low data rates ranging from 9.6 kbit/s to 100 kbit/s. A polar **LEO** satellite will typically pass over a given ground station 2 to 4 times/day, each pass having a communication window of 5 min to 10 min. For ThingSat, the CubeSat Operator provides only 2 ground stations, which are both in Europe, communicating with the CubeSat via a 10 kbit/s UHF/VHF link. Thus, the daily throughput is roughly 1 500 kB, corresponding to 2GS x 2 passes/day x 5-min pass duration x 10 kbit/s. However, this throughput must be shared between communications to/from the **OBC**, for telecommand/telemetry/update, and to- and from hosted payloads. Therefore in practice, the total communication budget available for ThingSat via the UHF/VHF link is around 300KB/day.

DIRECT COMMUNICATION SETUP VIA **LPWAN**

ThingSat can communicate directly with LoRa. In principle, although it is not used as such so far, this communication link could also be used

to transport software updates. The ThingSat payload may act as either:

1. A Sat-IoT end-device (ED) that will send LoRa frames to terrestrial LoRaWAN gateways or a ThingSat ground stations.
2. An in-orbit LoRa sniffer.
3. A store-carry-and-forward LoRa gateway.

Patterns 1 and 2 allow to benchmark simple ground-space LoRa links by computing statistics over multiple sent/received frames. Pattern 3 is a more complex scenario: the satellite stores packets received from the Sat-IoT end-device carries them and delivers them once LoRa ground stations are inside the footprint of the satellite.

7.1.3 INTERMITTENT COMMUNICATION AND POWER SUPPLY

One issue inherent about the setup is that the ThingSat payload is not constantly powered on. Typically, at any point in time, only one single hosted payload is powered on. For a 3U, 1U is dedicated to the OBC and the remaining 2U is available for hosted payloads, 8 payloads slots of 0.25U in the case of ThingSat. Therefore, on average ThingSat is powered only 1/8th (12.5 %) of the time, further reduced by other factors such as mission specificities, regulations, battery level and others.

7.1.4 HOSTED PAYLOAD UPDATE REQUIREMENTS

Data exchanges between the Payload Maintainer and ThingSat (Step 5 on Figure 7.2) consist of downlinks: used by ThingSat to send mission results (radio metadata, frame stats, collected LoRa frames) and diagnosis data (debug info on failed missions/updates) and uplinks: used for software updates of two categories:

1. Firmware updates: to fix bugs, add/improve functionality, typically ≈ 200 kB per firmware, 1 firmware/month.
2. Mission updates: to configure scenarios, typically ≈ 700 B per mission scenario, 1 scenario/day.

7.2 SOFTWARE UPDATE IMPLEMENTATION

7.2.1 SECURITY REQUIREMENTS

The minimal security guarantees aimed for with Cubedate are authenticity and integrity of software updates delivered over the network, during the lifetime of the satellite mission, approximately 5 to 10 years. Cubedate must allow for crypto agility, i.e. update the crypto primitives used to secure update to the satellite while in operation. This need can be dictated either by cryptography's evolving state-of-the-art (implementation/algorithm vulnerabilities are discovered) or by

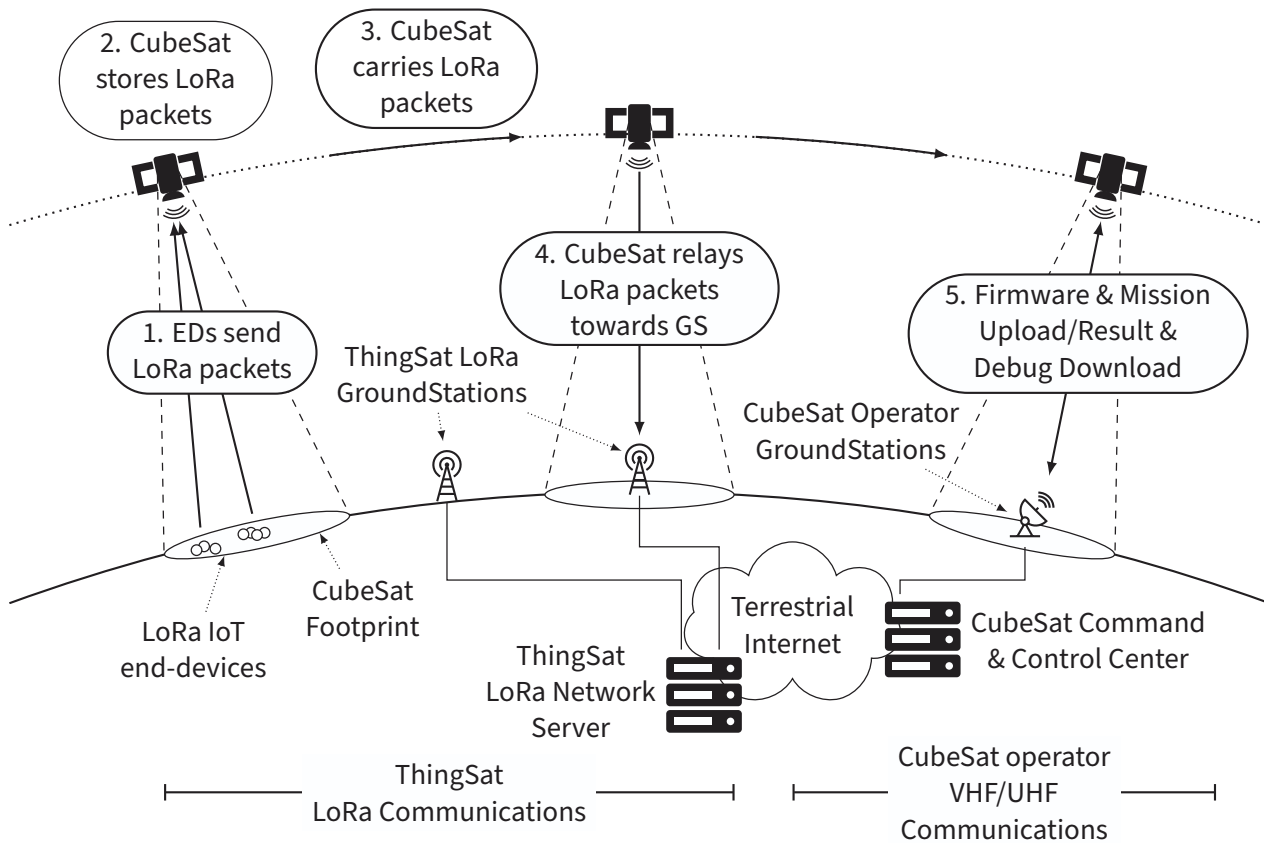


Figure 7.2: ThingSat in-orbit communication patterns.

the need to transfer the trust anchor to a new entity, for when the authorized maintainer has changed. Additional guarantees beyond authenticity/integrity should also be possible with Cubedate, such as confidentiality, software update replay attacks, or software update mismatch attacks.

7.2.2 TRUST ANCHOR

Our model is based on a single trust anchor: the secret key from the single authorized maintainer for the CubeSat hosted payload. There is no mitigation if this trust anchor used is compromised. It relies on the maintainers' ability to keep their private keys secure. Extensions using a possible hierarchical public key infrastructure are possible but out of scope for this work.

7.2.3 CUBEDATE SOFTWARE LIFE-CYCLE PHASES

The basic process used for securing authenticity and integrity of software updates is decomposed in six phases shown in Figure 7.3. During a preliminary, pre-flight phase (*Phase 0*) the authorized maintainer for the CubeSat-hosted payload produces and flashes the payload with commissioning material: a bootloader, the initial firmware, and authorized crypto material, including a public key, and a cryptographic

hash function. Once the hosted payload is commissioned it can be sent to the CubeSat operator of installation in the CubeSat.

Once the CubeSat is in orbit, the hosted payload maintainer can trigger iterations through cycles of Phases 1 to 5, whereby the authorized maintainer can build a new software update (*Phase 1*), hash the update and sign the hash (*Phase 2*) then push a network transfer (PUT) towards the hosted payload via the ground station and the OBC (*Phase 3.1*). The next time it wakes up, the hosted payload can then ping and fetch (GET) the update from the OBC (*Phase 3.2*), proceed to verify the signature and the hash (*Phase 4*), and upon successful verification, install/boot the new software (*Phase 5*), otherwise the update is dropped.

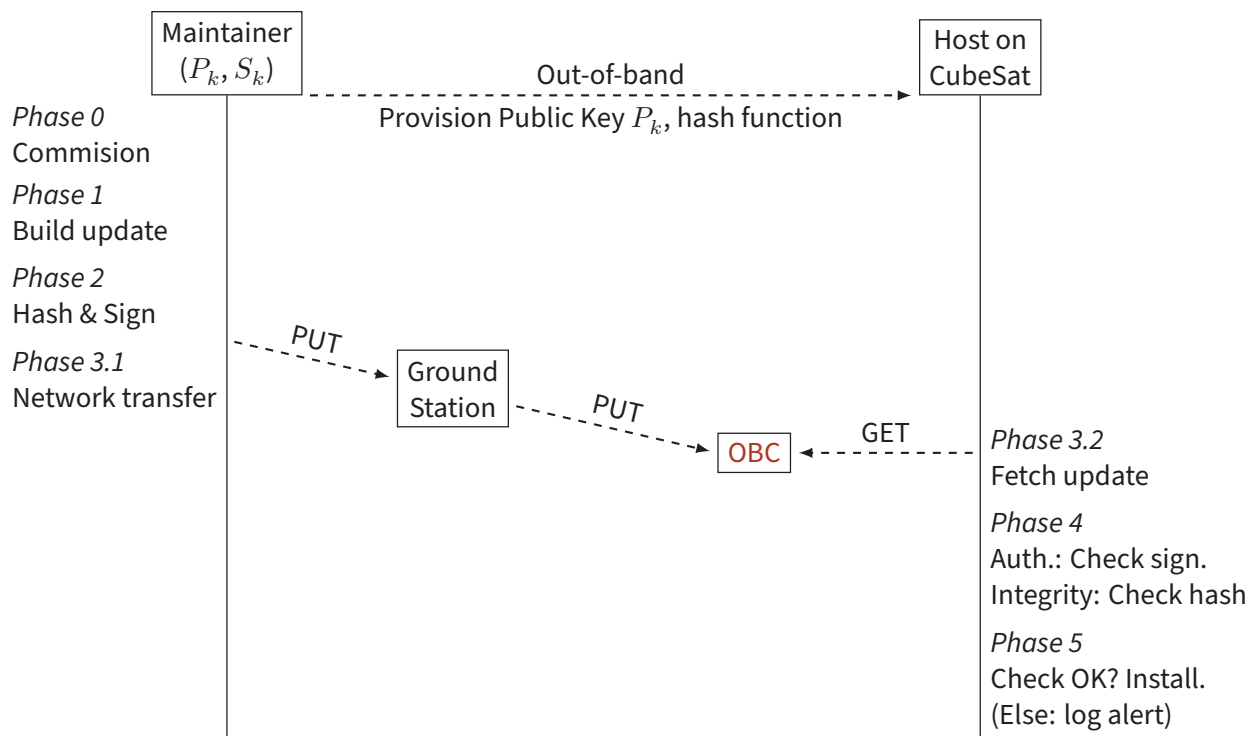


Figure 7.3: CubeSat hosted payload secure software update process.

7.2.4 SUPPORTING NETWORK TRANSPORT HETEROGENEITY

This aspect concerns *Phase 3.1* and *3.2* in Figure 7.3. Security guarantees on software updates must remain valid end-to-end. Depending on the use-case, “end-to-end” spans differently, as depicted for example in Figure 7.4. In the most complex case tackled in this work, end-to-end means all the way from the hosted payload software maintainer to the payload hosted in orbit on the CubeSat. Software updates may be transported over one or more network links of varying nature such as either developer-to-ground station link (Internet) or ground station-to-CubeSat links (UHF/VHF, LoRa...) or intra-CubeSat buses (CAN, I2C, RS-232...).

Intermittent power supply, combined with orbiting and radio range limitations impacts the reliance of the network connectivity to/from

the hosted payload: establishing a delay-tolerant path and in-network data caching might be required. To cope with this wide variety of network paths and links, including ultra-constrained low-power elements, different approaches can be envisioned at the network layer, the transport layer and the application layer. Approaches span from proprietary solutions to standards such as the low-power IPv6 protocol stack (6LoWPAN, UDP, CoAP) or experimental stacks such as information-centric networking which benefits from in-network caching even with small caches on microcontrollers [84].

Nevertheless, in order to retain generality, Cubedate does not specify any particular approach at the network, transport and application layers to enable the delivery of software updates across the network. Cubedate only aims to guarantee end-to-end security properties for the software update binaries that are delivered, somehow, over the network.

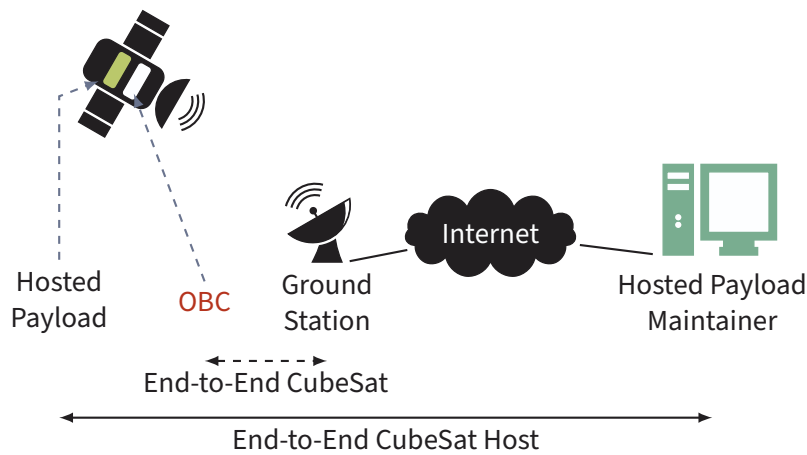


Figure 7.4: CubeSat hosted payload software update security end-to-end.

7.2.5 SUPPORTING UPDATED SOFTWARE HETEROGENEITY

This aspect concerns both *Phase 1* and *Phase 5* in Figure 7.3. As seen in subsection 7.2.3, software updates may be of various nature and size. Cubedate aims to support the same mechanism, workflow and guarantee to update the CubeSat (1) firmware updates, (2) mission scenario files and (3) runtime configuration files.

For this reason, choose not to rely on specialized approaches such as DFU (Device Firmware Update [33]) which assumes that the software is firmware and that the device is connected directly via some local bus connection (e.g. USB).

Instead, the aim is to combine the use of generic and standard metadata characterizing software updates and state-of-the-art cryptographic primitives applicable on most low-power microcontrollers and a large variety of low-power networks, as described below.

7.2.6 LOW-POWER END-TO-END SECURITY USING SUIT

Cubedate leverages the SUIT manifest [3], an updated format as presented in [chapter 4](#). The Cubedate software update binary itself can be either encapsulated in the SUIT manifest, or transferred separately based on the URI provided in the manifest. For instance, the metadata includes a sequence number (preventing unwanted rollbacks), the expected device type (preventing software mismatch), the SHA-256 digest of the software update binary and of the manifest, and the Ed25519 digital signature of the manifest (the metadata). As such, using Cubedate, software updates for payload hosted on CubeSats mitigate attacks including:

TAMPERED SOFTWARE UPDATE ATTACKS

An attacker may try to update the IoT device with a modified and intentionally flawed software image. To counter this threat, Cubedate uses digital signatures on a hash of the image binary and the metadata to ensure integrity of both the firmware and its metadata.

UNAUTHORIZED SOFTWARE UPDATE ATTACKS

An unauthorized party may attempt to update the IoT device with modified image. Using digital signatures and public key cryptography, Cubedate ensures that only the authorized maintainer (holding the authorized private key) will be able to update the device.

SUPPORTING CRYPTO AGILITY

The first level of crypto agility enabled by Cubedate uses flexibility provided by the SUIT standard specification: while keeping the same metadata and workflow, diverse crypto primitives backends can be used. For instance, to upgrade from pre- to post-quantum security, digital signature performed with Ed25519 (elliptic curve crypto), can be swapped for hash-based signatures, such as HSS-LMS as described in [chapter 3](#).

The second level of crypto agility enabled by Cubedate leverages a dedicated embedded runtime architecture: on the one hand, the software update manager (implementing SUIT-related operations) is placed in the firmware image itself. On the other hand, cryptographic operations are performed in software only.

Thus, changing the trust anchor stored is as simple as swapping a public key in the next firmware's update manager. Authorization to update the firmware can thus be easily delegated to another maintainer, who can take over the production and the roll out of authorized updates. Furthermore, the update manager in the next firmware image could implement and use upgraded cryptographic primitives.

GUARANTEES BEYOND AUTHENTICITY/INTEGRITY

Cubedate may also guarantee confidentiality by optionally encrypting software updates transmitted over the network. It is performed using the encrypt/decrypt mechanism provided by the SUIE specifications [164], using a symmetric cryptographic key commissioned in the update manager by the authorized maintainer. Confidentiality can mitigate additional cyberattacks leveraging analysis of CubeSat firmware/software binaries.

Going beyond authenticity, integrity and confidentiality guarantees for software updates delivered over the network, using Cubedate also mitigates other attacks including the reply attack and software update mismatch attack vectors covered by SUIE as described in [chapter 4](#)

7.3 PERFORMANCE EVALUATION

In the following, code measurements were generated compiling with ARM GCC 10.2.1, optimized for code size. As code base, RIOT release 2022.01 and SUIE configured with Ed25519 digital signatures provided by the C25519 crypto library is used, as this library has a particular small memory footprint [3].

7.3.1 MEMORY FOOTPRINT OVERHEAD

To evaluate the RAM and Flash footprint of our Cubedate implementation, it is applied to the ThingSat use-case, compiled for the hosted payload hardware described in [subsection 7.1.1](#).

In [Table 7.1](#) and [Table 7.2](#), the RAM and Flash memory requirements for a ThingSat firmware with/without Cubedate-compliant updates are compared. It is observed that Cubedate requires a memory budget of ≈ 4 KiB of RAM and ≈ 19 KiB of Flash, which represents roughly a 10 % increase in the total RAM and Flash memory budget for ThingSat.

- **ThingSat** refers to the ThingSat payload application with no software updates support
- **Cubedate** implementation of the Cubedate architecture on the ThingSat payload
- **CAN** stack as well as low level interface
- Crypto includes all cryptographic algorithms such as digest algorithm, digital signature, Elliptic curve with bignum code, as well as pseudo-random number generator
- CoAP protocol library (CoAP endpoint handler stack excluded)
- CSP (Cube Sat Protocol) network stack to communicate with the **OBC**.
- LoRa GW includes the sx1302 driver as well as the LoRa gateway code

Table 7.1: Cubedate implementation: memory flash footprint of the ThingSat firmware with and without the Cubedate component.

	ThingSat	+Cubedate
CAN	8 762	8 762
Crypto	7 386	13 760
CoAP	2 192	1 632
CSP	11 771	12 653
SUIE	0	7 425
LoRa GW	45 688	45 688
Firmware	108 881	114 088
Total	184 680	204 008

Table 7.2: Cubedate implementation: memory RAM footprint of the ThingSat firmware with and without the Cubedate component.

	ThingSat	+Cubedate
CAN	10 774	10 774
Crypto	633	64
CoAP	1 024	1 024
CSP	8 541	8 541
SUIE	0	3 200
LoRa GW	22 300	22 300
Firmware	7 008	8 225
Total	50 280	54 128

Table 7.3: Cubedate implementation: SUIE metadata size.

Component	Size
Sequence Number	4
Manifest Digest	32
Data Digest	32
Identifiers	32
Data-URI	64
Authentication	64
Other	96
Total	324

- SUIT encompasses all components enabling retrieval and installation of suit data (fw or other), this include e.g. the CoAP endpoint stack.
- Firmware: application specific code related to the CubeSat Payload excluding the LoRa gateway

7.3.2 NETWORK TRANSFER OVERHEAD

On the UHF/VHF link at 1kb/s, the additional network transfer time induced by the Cubedate firmware size overhead (19KB) is roughly 15 seconds. This overhead is reasonable, but non-negligible considering that a connection to the CubeSat is segmented in time windows of ≈ 300 seconds.

Next, in [Table 7.3](#) a more detailed look at the metadata, the SUIT manifest, used to secure Cubedate software updates for ThingSat is given. As is visible, the metadata including all [CBOR/COSE](#) formatting, digests (SHA-256 hashes) and authentication data (Ed25519 signature) amounts to ≈ 330 B. The metadata overhead thus incurs negligible overhead, an increase of +0.15 %, in case of a ThingSat firmware update (of size 200KB on average, recall [subsection 7.2.3](#)). However, for smaller software update such as updating a mission scenario (average size 700B) the metadata overhead is significant (almost +50%). Nevertheless, on the UHF/VHF link at 1kb/s, this overhead remains negligible in terms of additional network transfer time.

7.4 DISCUSSION

7.4.1 PORTABILITY

Bolted on top of RIOT, our Cubedate implementation works out-of-the-box (or is trivially portable) on a very wide variety of low-power hardware built on Cortex-M microcontrollers: the bulk of the 200+ boards supported by RIOT. However, additional work would be needed to support other hardware based on different 32-bit microcontroller architectures (e.g. RISC-V).

7.4.2 NETWORK STACK SIMPLIFICATION & STANDARDIZATION

The use of CSP was mandated by the CubeSat operator. The purpose of CSP was to provide an ultra-low footprint equivalent of the IP protocol stack for CubeSats with legacy (8-bit) microcontrollers. However, on modern (32-bit) microcontrollers such as those used in ThingSat, this approach is can be discussed. More widely spread standard alternatives to CSP seems possible for a similar “price”. For example, RIOT’s default low-power IPv6 (6LoWPAN) stack used with static routing has a footprint in RAM/Flash memory that is comparable to libCSP memory footprint. The 6LoWPAN stack could run directly on the CAN bus or on LoRa (see 6LoCAN [[168](#)] and SCHC [[120](#)]).

7.4.3 ALTERNATIVE CRYPTOGRAPHIC PRIMITIVES

In the experimental evaluation, Ed25519 (elliptic curve cryptography) digital signature providing 128-bit pre-quantum security is used. One can consider either alternative primitives, while remaining compliant with Cubedate and the SUIT standard. One compromise to significantly decrease network transfer and memory footprint with Cubedate is to use the symmetric Hash-based Message Authentication Code (HMAC) instead of digital signatures for authentication. Another option would be to upgrade security to 128-bit post-quantum security by using hash-based signature instead of Ed25519.

7.5 CONCLUSION

As the space race intensifies, rises the need for state-of-the-art security to protect software updates on multi-tenant CubeSat in orbit. In this chapter, I present a corresponding case-study: ThingSat, a low-power payload, hosted on a CubeSat operated by a separate entity, currently orbiting. Based on the ThingSat payload, a framework achieving strong security guarantees and low overhead, for continuous deployment of software over the air on multi-tenant CubeSat is designed called Cubedate. Cubedate provides a full update environment for both full firmware updates as well as the transfer of mission script file for VM execution. The open source implementation of Cubedate provided and evaluated for ThingSat was built to be reusable on a wide variety of low-power CubeSat hardware.

With the implementation here I show how SUIT and the other components presented in this work can be used to maintain and adapt the logic running as CubeSat payload. Mission logic can be adapted on the fly via the SUIT updates and can potentially be isolated inside a Femto-Container, allowing third party contributors to run mission code on the ThingSat platform.

CHAPTER 8

CONCLUSION

Software reconfiguration, while ubiquitous on desktop and server systems, is fledgling on IoT devices. The challenges associated with the constrained nature of the devices and network links hamper the design of solutions. This thesis presents a number of solutions to address these challenges. This thesis presented a number of novel solutions enabling efficient software reconfiguration on resource-constrained devices, at several levels of the embedded software stack. Compared to prior work, these contributions offer a more future-proof and open-source implementation for the smallest of devices in the field.

8.1 SUMMARY

As communication between constrained devices and management systems needs to be secured against malicious attacks, cryptographic primitives are required as base for any communication. For this purpose, [chapter 3](#) provided a comparative evaluation state-of-the-art pre-quantum digital signatures with promising post-quantum digital signature algorithms. The benchmarks use real world hardware spanning a set of different 32 bit architectures. Demonstrated is the additional performance cost involved with the current post-quantum signature algorithms. While not all new post-quantum algorithms are able to run on all hardware, sufficient algorithms are available to have candidates available within the constraints around embedded devices.

Digital signature algorithms can be applied to secure firmware updates against tampering and a variety of other attack vectors. For this purpose in [chapter 4](#), the design and evaluation of an open firmware update manifest standard with respect to current firmware called SUIT is addressed. While secure updates on constrained devices are not trivial and must protect against numerous attack vectors, as well as ensure a design suitable for constrained devices, the SUIT manifest specification manages to achieve these goals. The design of the manifest, leveraging both [CBOR](#) and [COSE](#), achieves a small size with to the firmware. This ensures a low overhead on both the network links to the devices and the constrained devices themselves. Demonstrated is that resource overhead of the implementation is sufficiently low that standards-compliant firmware updates can be provided on microcontrollers with memory as low as 32 kB of RAM and 128 kB of flash. This provides a fundamental and open building block

towards securing the IoT ecosystem against future vulnerabilities. The above properties, combined with the open source implementation I published and integrated in the RIOT operating system, provide a fundamental building block towards securing the IoT ecosystem against future vulnerabilities.

The SUIT specification itself is concerned primarily with the secure delivery of firmware updates. However not all updates require a full firmware update to patch required functionality. Therefore, in [chapter 5](#) presented, a new VM optimized for constrained devices is proposed, based on the popular eBPF virtual machine (VM) in the Linux kernel. rBPF provides a tiny register-based VM, which when employed for some code, incurs negligible memory overhead compared to native execution of this code in the host OS. While extra overhead is added by execution of software modules inside rBPF, the code executed is isolated within the VM and can not influence the memory outside the VM without specific permissions. Furthermore rBPF does not rely on any hardware security mechanisms for the memory isolation, it is portable across a wide range of architectures. Experimental comparative evaluation against a WebAssembly runtimes on microcontrollers shows that rBPF is a promising approach to isolate small software modules on constrained devices. While rBPF is shown to be slower than the fastest WebAssembly runtimes on microcontrollers, the overhead when adding rBPF to existing applications is only around 10 %, significantly less than other VMs.

Building on top of the rBPF virtual machine (VM), [chapter 6](#) then presented a new middleware runtime architecture design specifically for constrained devices: Femto-Containers. Femto-Containers provides a FaaS-like capabilities environment for constrained embedded devices, allowing modular and multi-tenant execution of small isolated software modules. Several implementations have been published as open source, including an implementation of a formally verified hosting engine providing fault-isolation guarantees (in collaboration with formal verification experts). With this, Femto-Container provides an isolated runtime without specific hardware requirements for security. Experimental benchmarks are provided on common microcontroller architectures to demonstrate the FaaS-like capabilities of Femto-Container and the negligible flash and RAM requirements added by Femto-Container.

As demonstrating case-study, the Cubedate framework with ThingSat for updating a low-power payload hosted on CubeSats is provided in [chapter 7](#). Last but not least, [chapter 7](#) presents Cubedate, a novel framework for strong security guarantees and low network transfer overhead for continuous deployments of software modules in the challenging environment of CubeSats on Low-Earth Orbit (LEO). To achieve this, Cubedate applies the results of [chapters 4 to 6](#) to this use-case. By combining secure software updates with SUIT, rBPF virtualisation and Femto-Containers, Cubedate enables convenient updates of not only firmwares but also individual satellite mission files hosted in Femto-Containers. This demonstrates and validates concretely that, when used together, the components of this thesis

improve the state-of-the-art by providing a set of secure mechanisms for managing, isolating and executing firmware modules on tiny constrained devices.

Together the components of this thesis improve the state-of-the-art by providing a set of secure mechanisms for managing, isolating and executing firmware modules on tiny constrained devices.

8.2 PERSPECTIVES

Concerning post-quantum digital signatures benchmarked, while most options are possible to deploy on the constrained devices used, none of the options match the performance of the pre-quantum options. While these algorithms perform sufficiently to be standardized by NIST, when considering them in a constrained scenario their attractiveness is lacking. Each post-quantum digital signature algorithm requires some concession, either on the signature size or the required memory. Further exploration for different algorithms more suitable for embedded devices can save considerable memory or network transfer overhead. For example, that the NIST competition is still on-going, collaborative work with cryptographers to develop signature algorithms with signature verification suitable for constrained devices, at an increase in resource consumption on the signature generation side, would be beneficial here.

SUIT manifest as implemented is sufficient for transferring payloads in a secure way to their target device. The work on the SUIT manifest is not frozen however, with extensions and new options in the process of standardization by the IETF. One particular attractive option is the full firmware encryption, extending the security guarantees already provided with confidentiality. A further advantage is the shift to symmetric cryptography, mitigating the need for the costly signature verification on the manifest. Furthermore, the SUIT specification in its current form provides a huge amount of flexibility. A more constrained variant could encompass a larger device base, while still providing the security level and main use case of firmware updates.

The rBPF VM with Femto-Container on top provide a rich and secure FaaS environment. Multiple avenues of improvements are available on these technologies, as discussed in [chapter 5](#) and [chapter 6](#). Direct improvements to the rBPF VM are possible to increase the performance and decrease the application size. Femto-Container itself is portable across many different operating systems and device architectures. However, it heavily relies on the integration into the host operating system, where providing bindings and facilities of the operating system to Femto-Container is often manual work by developers. Investigating how this integration can be simplified to decrease the burden on the developers can in turn improve the user experience for developers working on Femto-Container applications. One particular avenue for exploration would be the automatic generation of bindings to the operating system and the translation of data from the Femto-Container VM to host operating system and vice versa.

Altogether, the challenge of long-term maintenance of constrained IoT devices in the field persists. While this thesis addresses a number of challenges and provides solutions, more work is needed to provide a solution for these devices. A holistic solution at large scale is required to address these challenges in an encompassing way [47]. This requires a solution for constrained device management where software updates and reconfiguration is provided as core aspect. More work is needed to provide a holistic solution, at large scale. For instance, the lack of convenience of available software update back-ends is a bottleneck. Separate recent work such as [47] also hints at this lack, and points towards solutions that are similar to what I proposed in my thesis: combining software virtualization over HAL and a general-purpose OS. All in all, a paradigm shift is still to happen, whereby the update and reconfiguration software components of constrained devices is no longer an afterthought or an extra feature, but instead a core feature, always-required and always-available on such devices or fleets thereof.

APPENDIX A

BIBLIOGRAPHY

- [1] Koen Zandberg and Emmanuel Baccelli. “Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF”. In: *9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks, PEMWN 2020, Berlin, Germany, December 1-3, 2020*. IEEE, 2020, pp. 1–6. DOI: [10.23919/PEMWN50727.2020.9293081](https://doi.org/10.23919/PEMWN50727.2020.9293081). URL: <https://doi.org/10.23919/PEMWN50727.2020.9293081>.
- [2] Koen Zandberg, Emmanuel Baccelli, Shenghao Yuan, Frédéric Besson, and Jean-Pierre Talpin. “Femtocontainers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers”. In: *Middleware ’22: 23rd International Middleware Conference, Quebec, QC, Canada, November 7 - 11, 2022*. Ed. by Paolo Bellavista, Kaiwen Zhang, Abdelouahed Gherbi, Saurabh Bagchi, Marta Patiño, Giuseppe Di Modica, and Julien Gascon-Samson. ACM, 2022, pp. 161–173. DOI: [10.1145/3528535.3565242](https://doi.org/10.1145/3528535.3565242). URL: <https://doi.org/10.1145/3528535.3565242>.
- [3] Koen Zandberg, Kaspar Schleiser, Francisco Acosta Padilla, Hannes Tschofenig, and Emmanuel Baccelli. “Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check”. In: *IEEE Access* 7 (2019), pp. 71907–71920. DOI: [10.1109/ACCESS.2019.2919760](https://doi.org/10.1109/ACCESS.2019.2919760). URL: <https://doi.org/10.1109/ACCESS.2019.2919760>.
- [4] Gustavo Banegas, Koen Zandberg, Emmanuel Baccelli, Adrian Herrmann, and Benjamin Smith. “Quantum-Resistant Software Update Security on Low-Power Networked Embedded Devices”. In: *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. Lecture Notes in Computer Science. Springer, 2022, pp. 872–891. DOI: [10.1007/978-3-031-09234-3_43](https://doi.org/10.1007/978-3-031-09234-3_43). URL: https://doi.org/10.1007/978-3-031-09234-3_43.
- [5] Zhaolan Huang, Koen Zandberg, Kaspar Schleiser, and Emmanuel Baccelli. “RIOT-ML: toolkit for over-the-air secure updates and performance evaluation of TinyML models”. In: *Annals of Telecommunications* (2024), pp. 1–15.
- [6] François-Xavier Molina, Emmanuel Baccelli, Koen Zandberg, Didier Donsez, and Olivier Alphand. “Cubedate: Securing Software Updates in Orbit for Low-Power Payloads Hosted on CubeSats”. In: *12th IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks, PEMWN 2023, Berlin, Germany, September 27-29, 2023*. IEEE, 2023, pp. 1–6. DOI: [10.23919/PEMWN58813.2023.10304910](https://doi.org/10.23919/PEMWN58813.2023.10304910). URL: <https://doi.org/10.23919/PEMWN58813.2023.10304910>.
- [7] Brendan Moran, Hannes Tschofenig, Henk Birkholz, Koen Zandberg, and Øyvind Rønningstad. *A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest*. Internet-Draft draft-ietf-suit-manifest-25. Work in Progress. Internet Engineering Task Force, Feb. 2024. 101 pp. URL: <https://datatracker.ietf.org/doc/draft-ietf-suit-manifest/25/>.
- [8] Shenghao Yuan, Frédéric Besson, Jean-Pierre Talpin, Samuel Hym, Koen Zandberg, and Emmanuel Baccelli. “End-to-End Mechanized Proof of an eBPF Virtual Machine for Microcontrollers”. In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13372. Lecture Notes in

- Computer Science. Springer, 2022, pp. 293–316. DOI: [10.1007/978-3-031-13188-2_15](https://doi.org/10.1007/978-3-031-13188-2_15). URL: https://doi.org/10.1007/978-3-031-13188-2_15.
- [9] Schleiser, Kaspar and Zandberg, Koen and Abadie, Alexandre and Molina, François-Xavier. *sys/suit: initial support for SUIT firmware updates*. 2019. URL: <https://github.com/RIOT-OS/RIOT/pull/11818>.
 - [10] Zandberg, Koen. *libcose: Constrained node COSE library*. 2022. URL: <https://github.com/bergzand/libcose>.
 - [11] Zandberg, Koen. *NanoCBOR: CBOR library aimed at heavily constrained devices*. 2024. URL: <https://github.com/bergzand/NanoCBOR>.
 - [12] Zandberg, Koen. *rBPF: Initial include of small virtual machine*. 2021. URL: <https://github.com/RIOT-OS/RIOT/pull/19372>.
 - [13] Zandberg, Koen. *SUIT: Introduction of a payload storage API for SUIT manifest payloads*. 2020. URL: <https://github.com/RIOT-OS/RIOT/pull/15110>.
 - [14] Zandberg, Koen and Baccelli, Emmanuel. *Femto-Containers: Femto-Containers RIOT Implementation & Hands-on Tutorials*. 2022. URL: https://github.com/future-proof-iot/Femto-Container_tutorials.
 - [15] Francisco Javier Acosta Padilla et al. “The Future of IoT Software Must be Updated”. In: *IAB Workshop on Internet of Things Software Update (IoTSU)*. 2016.
 - [16] Adafruit Industries. *CircuitPython - The easiest way to program microcontrollers*. 2024. URL: <https://circuitpython.org/>.
 - [17] Cedric Adjih et al. “FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed”. In: *Proc. of IEEE WF-IoT*. Dec. 2015.
 - [18] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. “Firecracker: Lightweight virtualization for serverless applications”. In: *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 2020, pp. 419–434.
 - [19] ZigBee Alliance et al. *Zigbee Specification*. 2006.
 - [20] Amazon Web Services. *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*. 2024. URL: <https://www.freertos.org/>.
 - [21] Christian Amsüss, John Preuß Mattsson, and Göran Selander. *Constrained Application Protocol (CoAP): Echo, Request-Tag, and Token Processing*. RFC 9175. Feb. 2022. DOI: [10.17487/RFC9175](https://doi.org/10.17487/RFC9175). URL: <https://www.rfc-editor.org/info/rfc9175>.
 - [22] Christian Amsüss, Zach Shelby, Michael Koster, Carsten Bormann, and Peter Van der Stok. *Constrained RESTful Environments (CoRE) Resource Directory*. RFC 9176. Apr. 2022. DOI: [10.17487/RFC9176](https://doi.org/10.17487/RFC9176). URL: <https://www.rfc-editor.org/info/rfc9176>.
 - [23] Andrew W Appel. *Program logics for certified compilers*. Cambridge University Press, 2014.
 - [24] ARM. *Arm Cortex-M Processor Comparison Table*. 2023. URL: <https://developer.arm.com/documentation/102787/0300/>.
 - [25] Faisal Aslam, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan Rührup, and Zastash A Uzmi. “Optimized java binary and virtual machine for tiny motes”. In: *Distributed Computing in Sensor Systems: 6th IEEE International Conference, DCOSS 2010, Santa Barbara, CA, USA, June 21-23, 2010. Proceedings 6*. Springer. 2010, pp. 15–30.
 - [26] N Asokan, Thomas Nyman, Norrathep Rattanaivanon, Ahmad-Reza Sadeghi, and Gene Tsudik. “ASSURED: Architecture for secure software update of realistic embedded devices”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2290–2300.
 - [27] Atomic Object. *heatshrink: data compression library for embedded/real-time systems*. Dec. 2015. URL: <https://github.com/atomicobject/heatshrink>.
 - [28] Emmanuel Baccelli, Joerg Doerr, Shinji Kikuchi, Francisco Acosta Padilla, Kaspar Schleiser, and Ian Thomas. “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. 2018, pp. 504–507.
 - [29] Emmanuel Baccelli, Joerg Doerr, Shinji Kikuchi, Francisco Acosta Padilla, Kaspar Schleiser, and Ian Thomas. “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. 2018, pp. 504–507.
- [30] Emmanuel Baccelli, Cenk Gündogan, Oliver Hahm, Peter Kietzmann, Martine Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. “RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT”. In: *IEEE Internet Things J.* 5.6 (2018), pp. 4428–4440. doi: [10.1109/JIOT.2018.2815038](https://doi.org/10.1109/JIOT.2018.2815038). URL: <https://doi.org/10.1109/JIOT.2018.2815038>.
 - [31] Gustavo Banegas and Daniel J. Bernstein. “Low-Communication Parallel Quantum Multi-Target Preimage Search”. In: *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*. Ed. by Carlisle Adams and Jan Camenisch. Vol. 10719. Lecture Notes in Computer Science. Springer, 2017, pp. 325–335. doi: [10.1007/978-3-319-72565-9_16](https://doi.org/10.1007/978-3-319-72565-9_16). URL: https://doi.org/10.1007/978-3-319-72565-9_16.
 - [32] Dmitry Bankov, Evgeny Khorov, and Andrey Lyakhov. “On the Limits of LoRaWAN Channel Access”. In: *2016 International Conference on Engineering and Telecommunication (EnT)*. 2016, pp. 10–14. doi: [10.1109/EnT.2016.011](https://doi.org/10.1109/EnT.2016.011).
 - [33] J Beningo. *Update firmware in the field using a microcontrollers dfu mode*. 2018.
 - [34] Daniel J Bernstein et al. “ChaCha, a variant of Salsa20”. In: *Workshop record of SASC*. Vol. 8. 1. Citeseer. 2008, pp. 3–5.
 - [35] Daniel J Bernstein. “Curve25519: new Diffie-Hellman speed records”. In: *International Workshop on Public Key Cryptography*. Springer. 2006, pp. 207–228.
 - [36] Daniel J Bernstein. “The Poly1305-AES message-authentication code”. In: *International workshop on fast software encryption*. Springer. 2005, pp. 32–49.
 - [37] Daniel J Bernstein and Tanja Lange. “Post-quantum cryptography”. In: *Nature* 549.7671 (2017), pp. 188–194.
 - [38] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. “Keccak”. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2013, pp. 313–314.
 - [39] Ward Beullens. “Improved cryptanalysis of UOV and rainbow”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2021, pp. 348–373.
 - [40] Martin Björklund. *The YANG 1.1 Data Modeling Language*. RFC 7950. Aug. 2016. doi: [10.17487/RFC7950](https://doi.org/10.17487/RFC7950). URL: <https://www.rfc-editor.org/info/rfc7950>.
 - [41] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks*. RFC 7228. May 2014. doi: [10.17487/RFC7228](https://doi.org/10.17487/RFC7228). URL: <https://www.rfc-editor.org/info/rfc7228>.
 - [42] Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. Oct. 2013. doi: [10.17487/RFC7049](https://doi.org/10.17487/RFC7049). URL: <https://www.rfc-editor.org/info/rfc7049>.
 - [43] Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 8949. Dec. 2020. doi: [10.17487/RFC8949](https://doi.org/10.17487/RFC8949). URL: <https://www.rfc-editor.org/info/rfc8949>.
 - [44] Carsten Bormann, Simon Lemay, Hannes Tschofenig, Klaus Hartke, Bill Silverajan, and Brian Raymor. *CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets*. RFC 8323. Feb. 2018. doi: [10.17487/RFC8323](https://doi.org/10.17487/RFC8323). URL: <https://rfc-editor.org/rfc/rfc8323.txt>.
 - [45] Carsten Bormann and Zach Shelby. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. RFC 7959. Aug. 2016. doi: [10.17487/RFC7959](https://doi.org/10.17487/RFC7959). URL: <https://www.rfc-editor.org/info/rfc7959>.
 - [46] Guillaume Bouffard and Léo Gaspard. “Hardening a Java Card Virtual Machine Implementation with the MPU”. In: *Symposium sur la sécurité des technologies de l’information et des communications (SSTIC)*. 2018.
 - [47] Conner Bradley and David Barrera. “Escaping Vendor Mortality: A New Paradigm for Extending IoT Device Longevity”. In: *Proceedings of the 2023 New Security Paradigms Workshop, NSPW 2023, Segovia, Spain, September 18-21, 2023*. ACM, 2023, pp. 1–16. doi: [10.1145/3633500.3633501](https://doi.org/10.1145/3633500.3633501). URL: <https://doi.org/10.1145/3633500.3633501>.

- [48] Broadband Forum. *TR-069, CPE WAN Management Protocol Version 1.4*. Mar. 2018. URL: <https://www.broadband-forum.org/technical/download/TR-069.pdf>.
- [49] Broadband Forum. *User Services Platform*. URL: <https://usp.technology/>.
- [50] Niels Brouwers, Peter Corke, and Koen Langendoen. "Darjeeling, a Java compatible virtual machine for microcontrollers". In: *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*. 2008, pp. 18–23.
- [51] Stephen Brown and Cormac J Sreenan. "Software updating in wireless sensor networks: A survey and lacunae". In: *Journal of Sensor and Actuator Networks* 2.4 (2013), pp. 717–760.
- [52] Bytecode Alliance. *WebAssembly Micro Runtime (WAMR)*. Oct. 2020. URL: <https://github.com/bytecodealliance/wasm-micro-runtime>.
- [53] Cadence. *Xtensa LX Processor Platform*. 2024. URL: https://www.cadence.com/en_US/home/tools/silicon-solutions/compute-ip/tensilica-xtensa-controllers-and-extensible-processors/xtensa-lx-processor-platform.html.
- [54] Cesanta Software. *mJS - a new approach to embedded scripting*. Jan. 24, 2017. URL: <https://mongoose-os.com/blog/mjs-a-new-approach-to-embedded-scripting/>.
- [55] Cesanta Software. *Mongoose OS - reduce IoT firmware development time up to 90%*. 2024. URL: <https://mongoose-os.com/>.
- [56] André Chailloux, María Naya-Plasencia, and André Schrottenloher. "An Efficient Quantum Collision Search Algorithm and Implications on Symmetric Cryptography". In: *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10625. Lecture Notes in Computer Science. Springer, 2017, pp. 211–240. DOI: [10.1007/978-3-319-70697-9_8](https://doi.org/10.1007/978-3-319-70697-9_8). URL: https://doi.org/10.1007/978-3-319-70697-9_8.
- [57] Rym Chéour, Sabrine Khriji, Olfa Kanoun, et al. "Microcontrollers for IoT: optimizations, computing paradigms, and future directions". In: *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*. IEEE. 2020, pp. 1–7.
- [58] Inc. Cisco Systems. *cisco/hash-sigs: A full-featured implementation of of the LMS and HSS Hash Based Signature Schemes from draft-mcgrew-hash-sigs-07*. 2024. URL: <https://github.com/cisco/hash-sigs/>.
- [59] Robert Davis, Nick Merriam, and Nigel Tracey. "How embedded applications using an RTOS can stay within on-chip memory limits". In: *12th EuroMicro Conference on Real-Time Systems*. Citeseer. 2000, pp. 71–77.
- [60] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. "SQISign: compact post-quantum signatures from quaternions and isogenies". In: *Advances in Cryptology-ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part I* 26. Springer. 2020, pp. 64–93.
- [61] Fabrizio De Santis, Andreas Schauer, and Georg Sigl. "ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 692–697.
- [62] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. "Analysis of SHA-512/224 and SHA-512/256". In: *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*. Ed. by Tetsu Iwata and Jung Hee Cheon. Vol. 9453. Lecture Notes in Computer Science. Springer, 2015, pp. 612–630. DOI: [10.1007/978-3-662-48800-3_25](https://doi.org/10.1007/978-3-662-48800-3_25). URL: https://doi.org/10.1007/978-3-662-48800-3_25.
- [63] Krishna Doddapaneni et al. "Secure FoTA object for IoT". In: *IEEE LCN Workshops*. 2017.
- [64] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. "Run-time dynamic linking for reprogramming wireless sensor networks". In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. 2006, pp. 15–28.

- [65] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. “Contiki-a lightweight and flexible operating system for tiny networked sensors”. In: *29th annual IEEE international conference on local computer networks*. IEEE. 2004, pp. 455–462.
- [66] Morris J Dworkin. “SHA-3 standard: Permutation-based hash and extendable-output functions”. In: (2015).
- [67] Joshua Ellul and Kirk Martinez. “Run-time compilation of bytecode in sensor networks”. In: *2010 Fourth International Conference on Sensor Technologies and Applications*. IEEE. 2010, pp. 133–138.
- [68] European Commission. *Regulation Of The European Parliament And Of The Council on horizontal cybersecurity requirements for products with digital elements and amending Regulation (EU) 2019/1020*. 2019. URL: <https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:52022PC0454>.
- [69] Matt Fleming. “A Thorough Introduction to eBPF”. In: *Linux Weekly News* (2017).
- [70] J. Fletcher. “An Arithmetic Checksum for Serial Transmissions”. In: *IEEE Transactions on Communications* 30.1 (1982), pp. 247–252.
- [71] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei Zhang, et al. “Falcon: Fast-Fourier lattice-based compact signatures over NTRU”. In: *Submission to the NIST’s post-quantum cryptography standardization process* 36.5 (2018), pp. 1–75.
- [72] Dustin Frisch, Sven Reißmann, and Christian Pape. “An Over the Air Update Mechanism for ESP8266 Microcontrollers”. In: (Oct. 2017).
- [73] Mario Frustaci, Pasquale Pace, Gianluca Aloï, and Giancarlo Fortino. “Evaluating critical security issues of the IoT world: Present and future challenges”. In: *IEEE Internet of things journal* 5.4 (2017), pp. 2483–2495.
- [74] Gartner, Inc. *Gartner Says 8.4 Billion Connected “Things” Will Be in Use in 2017, Up 31 Percent From 2016*. Feb. 7, 2017. URL: <https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016>.
- [75] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. “Ultra lightweight JavaScript engine for internet of things”. In: *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. 2015, pp. 19–20.
- [76] Gareth George, Fatih Bakir, Rich Wolski, and Chandra Krintz. “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. 2020, pp. 220–231.
- [77] George Robotics Limited. *MicroPython - Python for microcontrollers*. 2023. URL: <https://micropython.org/>.
- [78] Lov K. Grover. “A Fast Quantum Mechanical Algorithm for Database Search”. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*. Ed. by Gary L. Miller. ACM, 1996, pp. 212–219. DOI: [10.1145/237814.237866](https://doi.org/10.1145/237814.237866). URL: <https://doi.org/10.1145/237814.237866>.
- [79] Kai Grunert. “Overview of JavaScript engines for resource-constrained microcontrollers”. In: *2020 5th International Conference on Smart and Sustainable Technologies (SpliTech)*. IEEE. 2020, pp. 1–7.
- [80] Fiona Guerin, Teemu Kärkkäinen, and Jörg Ott. “Towards a Programmable World: Lua-based Dynamic Local Orchestration of Networked Microcontrollers”. In: *Proceedings of the 14th Workshop on Challenged Networks*. 2019, pp. 13–18.
- [81] Robbert Gurdeep Singh and Christophe Scholliers. “WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 2019, pp. 27–36.
- [82] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. “Bringing the web up to speed with WebAssembly”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 185–200.

- [83] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. “Operating systems for low-end devices in the internet of things: a survey”. In: *IEEE Internet of Things Journal* 3.5 (2015), pp. 720–734.
- [84] Oliver Hahm, Emmanuel Baccelli, Thomas C. Schmidt, Matthias Wählisch, Cédric Adjih, and Laurent Massoulié. “Low-power internet of things with NDN & cooperative caching”. In: *Proceedings of the 4th ACM Conference on Information-Centric Networking, ICN 2017, Berlin, Germany, September 26-28, 2017*. Ed. by Thomas C. Schmidt and Jan Seedorf. ACM, 2017, pp. 98–108. DOI: [10.1145/3125719.3125732](https://doi.org/10.1145/3125719.3125732). URL: <https://doi.org/10.1145/3125719.3125732>.
- [85] Tony Hansen and Donald E. Eastlake 3rd. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*. RFC 6234. May 2011. DOI: [10.17487/RFC6234](https://doi.org/10.17487/RFC6234). URL: <https://www.rfc-editor.org/info/rfc6234>.
- [86] Klaus Hartke. *Observing Resources in the Constrained Application Protocol (CoAP)*. RFC 7641. Sept. 2015. DOI: [10.17487/RFC7641](https://doi.org/10.17487/RFC7641). URL: <https://www.rfc-editor.org/info/rfc7641>.
- [87] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. “RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022.
- [88] Peter Hoddie and Lizzie Prader. *IoT Development for ESP32 and ESP8266 with JavaScript: A Practical Guide to XS and the Moddable SDK*. Springer, 2020.
- [89] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H Silverman, and William Whyte. “NTRUSIGN: Digital signatures using the NTRU lattice”. In: *Cryptographers’ track at the RSA conference*. Springer. 2003, pp. 122–140.
- [90] Jonathan W Hui and David Culler. “The dynamic behavior of a data dissemination protocol for network programming at scale”. In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. 2004, pp. 81–94.
- [91] “IEEE Standard for Low-Rate Wireless Networks”. In: *IEEE Std 802.15.4-2020* (2020), pp. 1–800. DOI: [10.1109/IEEESTD.2020.9144691](https://doi.org/10.1109/IEEESTD.2020.9144691).
- [92] IETF. *Trusted Execution Environment Provisioning (TEE) Working Group*. URL: <https://datatracker.ietf.org/wg/teep/about/>.
- [93] Intel. *TinyCrypt Cryptographic Library*. May 2024. URL: <https://github.com/intel/tinycrypt>.
- [94] RISC-V International. *RISC-V: The Open Standard RISC Instruction Set Architecture*. 2024. URL: <https://riscv.org/>.
- [95] IoT Analytics GmbH. *State of IoT 2024: Number of connected IoT devices growing 13% to 18.8 billion globally*. Sept. 3, 2024. URL: <https://iot-analytics.com/number-connected-iot-devices/>.
- [96] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. “Zero-knowledge from secure multiparty computation”. In: *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 2007, pp. 21–30.
- [97] ITU-T. “ITU-T Rec. Y.2060 (06/2012) Overview of the Internet of things”. In: (June 2012), pp. 1–22.
- [98] Jaein Jeong and David Culler. “Incremental network programming for wireless sensors”. In: *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004*. IEEE. 2004, pp. 25–33.
- [99] Liu Jian-band et al. “YouBike service down in Taiwan”. In: *Focus Taiwan* (Aug. 2016). URL: <http://focustaiwan.tw/news/asoc/201608310010.aspx>.
- [100] Narjes Jomaa, Paolo Torrini, David Nowak, Gilles Grimaud, and Samuel Hym. “Proof-oriented design of a separation kernel with minimal trusted computing base”. In: *18th International Workshop on Automated Verification of Critical Systems (AVOCS 2018)*. 2018.
- [101] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. “Improving Software Quality in Cryptography Standardization Projects”. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*. Los Alamitos, CA, USA: IEEE Computer Society, 2022, pp. 19–30. DOI: [10.1109/EuroSPW55150.2022.00010](https://doi.org/10.1109/EuroSPW55150.2022.00010). URL: <https://eprint.iacr.org/2022/337>.

- [102] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. *PQClean/PQClean: Clean, portable, tested implementations of post-quantum cryptography*. 2024. URL: <https://github.com/PQClean/PQClean>.
- [103] Yoonseok Ko, Tamara Rezk, and Manuel Serrano. “Securejs compiler: Portable memory isolation in javascript”. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 2021, pp. 1265–1274.
- [104] Trishank Karthik Kuppusamy, Lois Anne DeLong, and Justin Cappos. “Uptane: Security and customizability of software updates for vehicles”. In: *IEEE vehicular technology magazine* 13.1 (2018), pp. 66–73.
- [105] Alexandru Lavric, Adrian I Petrariu, and Valentin Popa. “Sigfox communication protocol: The new era of iot?”. In: *2019 international conference on sensing and instrumentation in IoT Era (ISSI)*. IEEE. 2019, pp. 1–4.
- [106] Xavier Leroy. “Formal verification of a realistic compiler”. en. In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782, 1557-7317. DOI: [10.1145/1538788.1538814](https://doi.org/10.1145/1538788.1538814). URL: <https://dl.acm.org/doi/10.1145/1538788.1538814> (visited on 12/16/2021).
- [107] Philip Levis and David Culler. “Maté: A tiny virtual machine for sensor networks”. In: *ACM Sigplan Notices* 10.605397.605407 (2002), pp. 85–95.
- [108] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. “Multiprogramming a 64kb computer safely and efficiently”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 234–251.
- [109] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. “Multiprogramming a 64kB Computer Safely and Efficiently”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Association for Computing Machinery, 2017, pp. 234–251.
- [110] Linaro Limited. *Mbed TLS*. 2024. URL: <https://www.trustedfirmware.org/projects/mbed-tls/>.
- [111] Patrick Lin. *10 Takeaways From Cal Poly’s Space Cyberattacks Report*. June 25, 2024. URL: <https://interactive.satellitetoday.com/via/july-2024/global-consolidation-is-changing-dynamics-of-the-sector/>.
- [112] lora-alliance. *TS001-1.0.4 LoRaWAN® L2 1.0.4 Specification*. June 2024. URL: <https://resources.lora-alliance.org/technical-specifications/ts001-1-0-4-lorawan-l2-1-0-4-specification>.
- [113] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. “Crystals-dilithium”. In: *Algorithm Specifications and Supporting Documentation* (2020).
- [114] David Malan. “Crypto for tiny objects”. In: *Harvard University, Cambridge, Massachusetts, USA, Tech. Rep* (2004).
- [115] Peter Marwedel. *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems, Second Edition*. Embedded Systems. Springer, 2011. ISBN: 978-94-007-0256-1. DOI: [10.1007/978-94-007-0257-8](https://doi.org/10.1007/978-94-007-0257-8). URL: <https://doi.org/10.1007/978-94-007-0257-8>.
- [116] Steven McCanne and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture.” In: *USENIX winter*. Vol. 46. 1993, pp. 259–270.
- [117] Robert J McEliece. “A public-key cryptosystem based on algebraic”. In: *Coding Thv* 4244 (1978), pp. 114–116.
- [118] David McGrew, Michael Curcio, and Scott Fluhrer. *Leighton-Micali Hash-Based Signatures*. RFC 8554. Apr. 2019. DOI: [10.17487/RFC8554](https://doi.org/10.17487/RFC8554). URL: <https://www.rfc-editor.org/info/rfc8554>.
- [119] Rud Merriam. “Software Update Destroys \$286 Million Japanese Satellite”. In: *Hackaday* (May 2016).
- [120] Ana Minaburo, Laurent Toutain, Carles Gomez, Dominique Barthel, and Juan-Carlos Zúñiga. *SCHC: Generic Framework for Static Context Header Compression and Fragmentation*. RFC 8724. Apr. 2020. DOI: [10.17487/RFC8724](https://doi.org/10.17487/RFC8724). URL: <https://www.rfc-editor.org/info/rfc8724>.
- [121] Moddable Tech Inc. *Moddable*. 2024. URL: <https://moddable.com/>.

- [122] Gabriel Montenegro, Jonathan Hui, David Culler, and Nandakishore Kushalnagar. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. Sept. 2007. DOI: [10.17487/RFC4944](https://doi.org/10.17487/RFC4944). URL: <https://www.rfc-editor.org/info/rfc4944>.
- [123] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. “Consolidate IoT edge computing with lightweight virtualization”. In: *IEEE network* 32.1 (2018), pp. 102–111.
- [124] Brendan Moran and Hannes Tschofenig. *A CBOR-based Firmware Manifest Serialisation Format*. Internet-Draft draft-moran-suit-manifest-03. Work in Progress. Internet Engineering Task Force, Oct. 2018. 42 pp. URL: <https://datatracker.ietf.org/doc/draft-moran-suit-manifest/03/>.
- [125] Brendan Moran, Hannes Tschofenig, and Henk Birkholz. *A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices*. RFC 9124. Jan. 2022. DOI: [10.17487/RFC9124](https://doi.org/10.17487/RFC9124). URL: <https://www.rfc-editor.org/info/rfc9124>.
- [126] Brendan Moran, Hannes Tschofenig, David Brown, and Milosch Meriac. *A Firmware Update Architecture for Internet of Things*. RFC 9019. Apr. 2021. DOI: [10.17487/RFC9019](https://doi.org/10.17487/RFC9019). URL: <https://www.rfc-editor.org/info/rfc9019>.
- [127] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. “CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds”. In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 1271–1287. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin>.
- [128] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. June 2018. DOI: [10.17487/RFC8439](https://doi.org/10.17487/RFC8439). URL: <https://www.rfc-editor.org/info/rfc8439>.
- [129] NIST. *Hash functions*. 2023. URL: <https://csrc.nist.gov/projects/hash-functions>.
- [130] Marek Novak and Petr Skryja. “Efficient partial firmware update for IoT devices with lua scripting interface”. In: *2019 29th International Conference Radioelektronika (RADIOELEKTRONIKA)*. IEEE, 2019, pp. 1–4.
- [131] George Oikonomou, Simon Duquennoy, Atis Elsts, Joakim Eriksson, Yasuyuki Tanaka, and Nicolas Tsiftes. “The Contiki-NG open source operating system for next generation IoT devices”. In: *SoftwareX* 18 (2022), p. 101089.
- [132] OMA. “LwM2M Technical Specification, Approved Version 1.0.2”. In: (Feb. 2018). URL: http://www.openmobilealliance.org/release/LightweightM2M/V1_0_2-20180209-A/.
- [133] OMA SpecWorks. “Lightweight Machine to Machine Technical Specification: Core, Approved Version 1.1”. In: (July 2018). URL: http://www.openmobilealliance.org/release/LightweightM2M/V1_1-20180710-A/.
- [134] OMA SpecWorks. “Lightweight Machine to Machine Technical Specification: Transport Bindings, Approved Version 1.1”. In: (July 2018). URL: http://www.openmobilealliance.org/release/LightweightM2M/V1_1-20180710-A/.
- [135] Sandro Pinto and Nuno Santos. “Demystifying Arm TrustZone: A Comprehensive Survey”. In: *ACM Comput. Surv.* 51.6 (2019), 130:1–130:36. DOI: [10.1145/3291047](https://doi.org/10.1145/3291047). URL: <https://doi.org/10.1145/3291047>.
- [136] Amir Pnueli, Michael Siegel, and Eli Singerman. “Translation validation”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS’98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS’98 Lisbon, Portugal, March 28–April 4, 1998 Proceedings 4*. Springer, 1998, pp. 151–166.
- [137] Niels Reijers and Chi-Sheng Shih. “CapeVM: A safe and fast virtual machine for resource-constrained Internet-of-Things devices”. In: *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. 2018, pp. 250–263.
- [138] Vincent Rijmen and Joan Daemen. “Advanced encryption standard”. In: *Proceedings of federal information processing standards publications, national institute of standards and technology* 19 (2001), p. 22.

- [139] Rodrigo Roman, Cristina Alcaraz, and Javier Lopez. “A survey of cryptographic primitives and implementations for hardware-constrained sensor network nodes”. In: *Mobile Networks and Applications* 12 (2007), pp. 231–244.
- [140] Peter Ruckebusch, Eli De Poorter, Carolina Fortuna, and Ingrid Moerman. “Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules”. In: *Ad Hoc Networks* 36 (2016), pp. 127–151.
- [141] Spyridon Samonas and David Coss. “The CIA strikes back: Redefining confidentiality, integrity and availability in security.” In: *Journal of Information System Security* 10.3 (2014).
- [142] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Anna Shubina. “The Halting Problems of Network Stack Insecurity”. In: *login Usenix Mag.* 36.6 (2011). URL: <https://www.usenix.org/publications/login/december-2011-volume-36-number-6/halting-problems-network-stack-insecurity>.
- [143] Jim Schaad. *CBOR Object Signing and Encryption (COSE)*. RFC 8152. July 2017. DOI: [10.17487/RFC8152](https://doi.org/10.17487/RFC8152). URL: <https://www.rfc-editor.org/info/rfc8152>.
- [144] J Schlien and D Raddino. “Narrowband internet of things whitepaper”. In: *White Paper, Rohde & Schwarz* (2016), pp. 1–42.
- [145] Göran Selander, John Preuß Mattsson, Francesca Palombini, and Ludwig Seitz. *Object Security for Constrained RESTful Environments (OSCORE)*. RFC 8613. July 2019. DOI: [10.17487/RFC8613](https://doi.org/10.17487/RFC8613). URL: <https://www.rfc-editor.org/info/rfc8613>.
- [146] Nik Shaylor, Douglas N Simon, and William R Bush. “A java virtual machine architecture for very small devices”. In: *ACM SIGPLAN Notices* 38.7 (2003), pp. 34–41.
- [147] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. DOI: [10.17487/RFC7252](https://doi.org/10.17487/RFC7252). URL: <https://www.rfc-editor.org/info/rfc7252>.
- [148] Zhengguo Sheng, Shusen Yang, Yifan Yu, Athanasios V Vasilakos, Julie A McCann, and Kin K Leung. “A survey on the ietf protocol suite for the internet of things: Standards, challenges, and opportunities”. In: *Wireless Communications, IEEE* 20.6 (2013), pp. 91–98.
- [149] Kyung-Ah Shim. “A survey on post-quantum public-key signature schemes for secure vehicular communications”. In: *IEEE Transactions on Intelligent Transportation Systems* 23.9 (2021), pp. 14025–14042.
- [150] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. In: *SIAM Rev.* 41.2 (1999), pp. 303–332. DOI: [10.1137/S0036144598347011](https://doi.org/10.1137/S0036144598347011). URL: <https://doi.org/10.1137/S0036144598347011>.
- [151] Volodymyr Shymanskyi. *WASM3: A high Performance WebAssembly Interpreter Written in C*. Oct. 2020. URL: <https://github.com/wasm3/wasm3>.
- [152] Sigfox. *Sigfox Device Radio Specification*. 2019. URL: <https://build.sigfox.com/sigfox-device-radio-specifications>.
- [153] Miguel Silva, David Cerdeira, Sandro Pinto, and Tiago Gomes. “Operating systems for Internet of Things low-end devices: Analysis and benchmarking”. In: *IEEE Internet of Things Journal* 6.6 (2019), pp. 10375–10383.
- [154] Saleh Soltan, Prateek Mittal, and H Vincent Poor. “BlackIoT: IoT Botnet of high wattage devices can disrupt the power grid”. In: *Proc. USENIX Security*. Vol. 18. 2018.
- [155] Secure Hash Standard. “Secure hash standard”. In: *FIPS PUB* (1995), pp. 180–1.
- [156] National Institute of Standards and Technology. “Digital Signature Standard”. In: *Federal Information Processing Standards FIPS 186-4*. NIST. July 2013.
- [157] Peter Van der Stok, Carsten Bormann, and Anuj Sehgal. *PATCH and FETCH Methods for the Constrained Application Protocol (CoAP)*. RFC 8132. Apr. 2017. DOI: [10.17487/RFC8132](https://doi.org/10.17487/RFC8132). URL: <https://www.rfc-editor.org/info/rfc8132>.
- [158] Milosh Stolikj, Pieter JL Cuijpers, and Johan J Lukkien. “Efficient reprogramming of wireless sensor networks using incremental updates”. In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*. IEEE. 2013, pp. 584–589.

- [159] James A. Storer and Thomas G. Szymanski. “Data Compression via Textual Substitution”. In: 29.4 (Oct. 1982), pp. 928–951. doi: [10.1145/322344.322346](https://doi.org/10.1145/322344.322346).
- [160] The Apache Software Foundation. *Apache NuttX is a mature, real-time embedded operating system (RTOS)*. 2024. URL: <https://nuttx.apache.org/>.
- [161] *The MCUboot Bootloader*. URL: <https://github.com/runtimeco/mcuboot>.
- [162] *The Update Framework*. URL: <https://github.com/theupdateframework/tuf>.
- [163] Hannes Tschofenig and Thomas Fossati. *Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things*. RFC 7925. July 2016. doi: [10.17487/RFC7925](https://doi.org/10.17487/RFC7925). URL: <https://www.rfc-editor.org/info/rfc7925>.
- [164] Hannes Tschofenig, Russ Housley, Brendan Moran, David Brown, and Ken Takayama. *Encrypted Payloads in SUIT Manifests*. Internet-Draft draft-ietf-suit-firmware-encryption-20. Work in Progress. Internet Engineering Task Force, July 2024. 55 pp. URL: <https://datatracker.ietf.org/doc/draft-ietf-suit-firmware-encryption/20/>.
- [165] Sami Vaarala. *Duktape*. 2024. URL: <https://duktape.org/>.
- [166] Michel Veillette, Peter Van der Stok, Alexander Pelov, Andy Bierman, and Carsten Bormann. *CoAP Management Interface (CORECONF)*. Internet-Draft draft-ietf-core-comi-17. Work in Progress. Internet Engineering Task Force, Mar. 2024. 48 pp. URL: <https://datatracker.ietf.org/doc/draft-ietf-core-comi/17/>.
- [167] W3C. *WASI: libc Implementation for WebAssembly*. May 2024. URL: <https://github.com/WebAssembly/wasi-libc>.
- [168] Alexander Wachter. *IPv6 over Controller Area Network*. Internet-Draft draft-wachter-6lo-can-01. Work in Progress. Internet Engineering Task Force, Feb. 2020. 18 pp. URL: <https://datatracker.ietf.org/doc/draft-wachter-6lo-can/01/>.
- [169] Jos Wetzels. “Internet of Pwnable Things: Challenges in Embedded Binary Security”. In: *login Usenix Mag.* 42.2 (2017). URL: <https://www.usenix.org/publications/login/summer2017/wetzels>.
- [170] Gordon F Williams. *Making Things Smart: Easy Embedded JavaScript Programming for Making Everyday Objects into Intelligent Machines*. Maker Media, Inc., 2017.
- [171] Martin Woolley. “Bluetooth core specification v5. 1”. In: *Bluetooth*. 2019.
- [172] Mehmet Erkan Yüksel. “Power consumption analysis of a Wi-Fi-based IoT device”. In: *Electrica* 20.1 (2020), pp. 62–71.
- [173] Zephyr Project. *Zephyr Project – A proven RTOS ecosystem, by developers, for developers*. 2024. URL: <https://www.zephyrproject.org/>.
- [174] Wei Zhou, Zhouqi Jiang, and Le Guan. “Understanding MPU Usage in Microcontroller-based Systems in the Wild”. In: *Proceedings 2023 Workshop on Binary Analysis Research*. San Diego, CA, USA: Internet Society. 2023.
- [175] Xiaorui Zhu, Xianping Tao, Tao Gu, and Jian Lu. “ReLog: A systematic approach for supporting efficient reprogramming in wireless sensor networks”. In: *Journal of Parallel and Distributed Computing* 102 (2017), pp. 132–148.
- [176] Torsten Zimmermann, Jens Hiller, Helge Reelfs, Pascal Hein, and Klaus Wehrle. “SPLIT: Smart Protocol Loading for the IoT.” In: *EWSN*. 2018, pp. 49–54.
- [177] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. “HACL*: A verified modern cryptographic library”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 1789–1806.

APPENDIX B

LIST OF ACRONYMS

ACDS	Attitude Determination and Control System
ANSI	American National Standards Institute
AOT	Ahead-of-Time
BPF	Berkeley Packet Filter
CAN	Controller Area Network
CBOR	Concise Binary Object Representation
CertFC	Certified Femto-Container
CoAP	Constrained Application Protocol
CORECONF	CoAP Management Interface
COSE	CBOR Object Signing and Encryption
DFU	Device Firmware Upgrade
ELF	executable and linkable format
FaaS	Functions-as-a-Service
HMAC	Hash-based Message Authentication Code
HSS	Hierarchical Signature System
IETF	Internet Engineering Task Force
IoT	Internet of Things
JIT	Just-in-Time
JOSE	Javascript Object Signing and Encryption
JSON	JavaScript Object Notation
LEO	Low-Earth Orbit
LMS	Leighton-Micali Signature
LoC	Lines of Code
LPWAN	Low Power WAN
LwM2M	Lightweight Machine-to-Machine
MAC	Message Authentication Code