# CerberOS: A Resource-Secure OS for Sharing IoT Devices

Sven Akkermans, Wilfried Daniels, Gowri Sankar R., Bruno Crispo and Danny Hughes
imec-DistriNet, KU Leuven
3001 Leuven, Belgium
sven.akkermans@cs.kuleuven.be

## Abstract

To continue to grow, the Internet of Things (IoT) requires scalable and secure system software solutions for resource-constrained devices. To maximize return on investment of these devices, IoT platforms should support multiple third-party applications and adaptation of software over time. However, realizing the vision of shared IoT platforms demands not only strong guarantees on the confidentiality and integrity of application data, but also guarantees on the use of critical resources such as computation, sensors and energy. We refer to this vision as *resource security*. Prior research on Operating Systems (OS) for tiny IoT devices has focused on miniaturizing core functionality such as scheduling and communication and does not consider resource security. To address this problem, we introduce CerberOS, a resource-secure OS for sharing IoT devices. CerberOS enables multiple applications on constrained IoT devices while, for the first time, guaranteeing data confidentiality, integrity and secure resource management. Our approach is based upon the twin pillars of virtualization, which isolates applications, and contracts, which control application resource usage. Evaluation shows that CerberOS supports the secure coexistence of up to seven applications on a representative IoT device with a memory usage of 40KB ROM and 5KB RAM while preserving multi-year battery lifetimes.

## Categories and Subject Descriptors

I.6 [**Operating Systems Security**]: Virtualization and security

## General Terms

DESIGN, SECURITY

*Keywords*

Multiple applications, virtualization, resource security, Internet of Things, operating systems

International Conference on Embedded Wireless
Systems and Networks (EWSN) 2017
20–22 February, Uppsala, Sweden
© 2017 Copyright is held by the authors.
Permission is granted for indexing in the ACM Digital Library
ISBN: 978-0-9949886-1-4

## 1 Introduction

The Internet of Things (IoT) is growing rapidly with large-scale networks of constrained devices being deployed in our homes and across all sectors of industry. Despite advances in hardware and software technologies, IoT deployments are costly to create and operate due to the manpower required to deploy, configure and manage thousands of devices. This motivates the maximization of Return on Investment (RoI) by sharing IoT platforms with multiple parties.

In this paper, we explore how these IoT platforms can be securely shared by multiple applications (apps), without increasing hardware costs or significantly decreasing battery life. Secure multi-app hosting is as-of yet poorly supported by current IoT OS research [9, 12, 17]. Memory protection often is not supported or requires additional hardware. Software on IoT devices is not prevented from blocking or exhausting essential resources. As IoT infrastructure providers cannot share their platforms with third parties while retaining full manageability and control of their resources, multi-app IoT devices are not common. This is a significant roadblock in achieving commercial large-scale IoT deployments.

To overcome this roadblock, we present *CerberOS*, a resource-secure operating system for tiny multi-app IoT nodes. The core idea of CerberOS is instruction-level monitoring and fine-grained resource management for all apps running on the device. This is achieved by an interpreting virtualization layer, implemented through a virtual machine, that isolates apps and separates the system into user and kernel space. *Resource security* is paramount in our system. Key resources, such as memory and peripherals, are managed so that apps can run safely on a node without being disrupted by other apps. Our research aims to leverage the benefits of existing, efficient solutions while going beyond prior work to enable resource-secure multi-app IoT devices. Specifically, CerberOS protects the device and its apps from any malicious or malfunctioning app.

CerberOS is implemented on a representative IoT device based on the ATmega1284P. It is designed to be modular and can work stand-alone but also allows the use of discrete assets from other OSs, such as timers, scheduling and networking. CerberOS works on nodes with as little as 40KB ROM and 5KB RAM and is shown to support seven coexisting applications. Our experimental evaluation proves that the performance overhead and energy impact of CerberOS is feasible for real-world IoT scenarios.

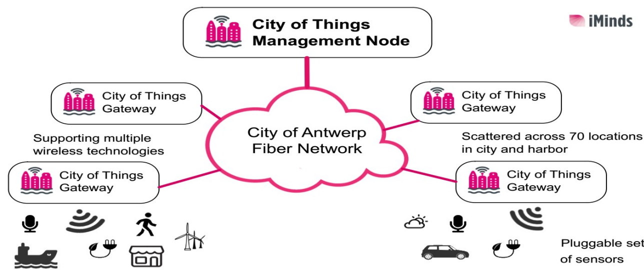


**Figure 1. A smart city deployment with multiple stakeholders for the city of Antwerp (Belgium) [27].**

The remainder of this paper is structured as follows: Section 2 gives the case, requirements and threat model for multi-app IoT devices, Section 3 discusses related work, Section 4 describes the design and architecture of CerberOS, Section 5 presents the evaluation and Section 6 concludes.

# 2 Background

## 2.1 The Case for Sharing IoT Devices

The IoT envisions a future where billions of Internet-connected devices are deployed in our environment to support novel cyber-physical applications. Contemporary IoT networks are rapidly growing in scale from smart buildings to smart cities. Research deployments such as City of Things [27], shown in Figure 1, and SmartSantander [24] already incorporate tens of thousands of IoT devices. However, commercial deployments of similar scale have been slow to appear. One reason for this disparity is the unclear RoI for large-scale IoT networks, which demand significant up-front investment in infrastructure as well as technical staff to deploy, manage and maintain the system.

Supporting multiple apps enables IoT infrastructure providers to increase their RoI. Multi-app nodes allow an IoT deployment to satisfy multiple stakeholders and therefore minimizes hardware costs and associated staff costs arising from the deployment, management and maintenance of devices. IoT infrastructure providers can specialize in deploying IoT infrastructure or platforms as a service and lease out resources on underutilized devices to third parties. IoT software developers can focus on their core competencies while still having access to scalable and elastic hardware solutions. The efficacy of this approach has been extensively demonstrated in mainstream cloud computing and the attractiveness of such a model has led key industry players to coin the term 'fog computing' [4]. Realizing multi-app IoT platforms requires security mechanisms that ensure: (i) only authorized parties deploy apps, (ii) apps execute as determined by contracts, (iii) app code and data remain confidential, and (iv) the integrity of app code and data is assured.

Large-scale IoT networks primarily consist of IETF Class-1 devices which have around 10KB RAM and 100KB ROM and are considered as having the minimal resources necessary to communicate securely with the Internet [5]. The stringent resource constraints of these IoT devices drives the development of custom lightweight OSs such as Contiki, TinyOS and RIOT [9, 17, 12]. These OSs do not provide memory protection or fine-grained resource management which are essential for securely sharing devices between untrusted third parties.

In addition, many IoT platforms are battery powered, employing energy saving techniques to achieve lifetimes of several years on a single battery charge [28]. Energy harvesting approaches are increasingly common but likewise impose a strict limit on the power consumption of IoT devices. These resource limitations must be considered when sharing IoT platforms. In contrast to the cloud, in fog computing every processor cycle, byte of memory and joule is precious and must be carefully managed. The infrastructure provider therefore requires support for monitoring and limiting the resources that apps use. The app developer likewise requires assurances that the resources they require will be available.

The ideal multi-app IoT OS executes efficiently on Class-1 embedded devices and secures the execution and resource usage of coexisting third-party apps. Management support is necessary to minimize the effort needed to develop, deploy and manage multiple apps and preserve scalability. In the following section, we analyze these requirements in detail.

## 2.2 Requirements Analysis

To realize a secure multi-app IoT OS, we identify three families of requirements: (i) core OS demands, (ii) resource-security guarantees, and (iii) classic security support.

The *core OS demands* for sharing embedded devices are:

- **Preemptive multi-threading:** The OS must provide mechanisms to safely and fairly execute coexisting apps. Cooperative scheduling, used by Contiki and TinyOS [9, 17], is insufficient since any app may block other apps or the OS by refusing to yield. It is essential that the OS can preempt processes to maintain control.
- **Remote management support:** For large-scale deployments such as smart cities, dispatching employees to service thousands of individual devices is clearly not cost-effective. All management activities must be performed remotely. This includes: deployment and removal of apps, adaptation of platforms, and monitoring of software state and node health.
- **Lightweight and efficient:** An IoT OS should execute efficiently on Class-1 devices with 10KB RAM and 100KB ROM as they make up the majority of today's embedded IoT systems. Furthermore, the OS should minimize energy consumption through duty-cycling techniques to retain multi-year battery lifetimes.

The limited resources of IoT devices must be carefully managed to ensure the resources of the infrastructure provider are spent according to contracts and *resource security* is maintained. Therefore, resources should not be blocked for an unreasonable time span (e.g., processor, network, peripherals) or exhausted (e.g., battery and memory):

- **Contractual limitation of resource usage:** Contracts specify exactly the resources an app requires and in which quantities, if applicable. A device should decide to either accept or reject the app based on this contract. This requirement protects the investment of the infrastructure provider and guarantees that apps have the resources necessary to execute if they are accepted.

- **Fine-grained accounting of resource usage:** Accounting is essential to enforce contracts and to take appropriate action when resource usage of an app reaches the contractually agreed limits. Therefore, the OS should monitor the resource usage of all apps.

A strong base of *classic security support* is necessary to prevent third parties from circumventing resource security mechanisms or breaking confidentiality, integrity or availability of other apps:

- **Memory isolation:** The program logic and data of apps must be isolated to protect their confidentiality and integrity from other malicious or malfunctioning apps. Current IoT OSs, such as Contiki, TinyOS and RIOT [9, 17, 12], do not meet this requirement as they provide unrestricted memory access to apps.
- **Network security:** Secure communications are essential to maintain the confidentiality and integrity of transmitted data. CerberOS achieves this by using IETF and NIST standard IoT network security, i.e., AES-128 running in CCM mode [29]. Each node has a unique session key, which must be transferred to the network manager to join the network. Shared join keys are not used. Our research focuses on building a secure OS for the IoT and does not contribute to network security.
- **Access control:** Only authorized parties may deploy software, perform management operations or access application APIs. This prevents unauthorized use of the IoT infrastructure or applications. Our approach naturally builds on standard multi-party access control techniques, such as HTTPS.

### 2.3 Threat Model

The threat model details the type of attack a multi-app OS should defend against. CerberOS uses standard network security so confidentiality, integrity and authenticity of communication is assumed and the administrator is trusted. In addition, we assume the OS is not compromised and is part of the trusted base of the platform.

The threat is a third-party app developer who has been authorized to deploy apps on the IoT infrastructure. The attack vector of the attacker is his ability to submit any software to remotely upload to the devices, including malicious or malfunctioning apps. However, he does not have physical access and cannot reprogram the node's firmware to replace or change the OS. We consider three types of software attacks by the deployed app: (i) disrupting operations on the node itself, thereby committing a denial-of-service attack on all apps of the device, (ii) interfering with a specific app, and (iii) modifying or stealing the data of an app, for instance, by influencing sensor readings or reading stored app data. Respectively, an attacker could upload an app to crash the node, to block a resource from being used by others and to read and modify data from a running app. A secure multi-app IoT OS has to defend against these attacks while still realizing the requirements detailed in Section 2.2.

## 3 Related Work

This section provides an overview of IoT OSs, software module security, interpreters for constrained devices and research that considers node resource management.

### 3.1 Overview IoT Operating Systems

Table 1 shows an overview of common IoT OSs and how they relate to the requirements outlined in Section 2.2. We consider Contiki, TinyOS, RIOT and FreeRTOS [9, 17, 12, 2] due to their popularity and widespread use. These OSs do not focus on security or resource security. None have full memory isolation; only RIOT provides some protection against bugs in single components due to its modular microkernel design. Only Contiki supports remote deployment and dynamic loading of software modules. Only FreeRTOS and RIOT provide preemptive multi-threading. Thus, existing OSs succeed only partly in meeting the requirements.

A new OS of note is Zephyr [18], a tiny open-source real-time IoT OS, which was announced by the Linux Foundation in February 2016. It focuses on a modular kernel design and divides the kernel into a core nanokernel and an optional microkernel which provides multitasking abilities and services such as memory and mailboxes. Zephyr aims to be highly configurable, to support a large number of architectures and to enable scalability, security and connectivity. While it shows similarities to our work (e.g., focus on portability and security), it supports only a single multitasking application with no explicit user space and no dynamic loading and manages device resources through synchronization mechanisms making it unable to withstand malicious tasks.

### 3.2 Software Module Security

Initial IoT nodes were single-purpose where isolation of software modules was not considered, with manufacturers and researchers focusing on meeting core functionalities within limited node resources. Recent years show increased interest in isolating multiple software modules with diverse approaches due to the heterogeneity of the IoT. The main difference with our work is that none of the discussed research considers software modules and their resource usage beyond binary access control and memory protection.

Melete [31] is a system built upon the Maté VM that supports coexisting apps for constrained nodes. Melete handles three challenges for multi-app use cases: (i) reliable code storage and execution, (ii) dynamic app deployment at the network level, and (iii) reliable code distribution. Similar to CerberOS, they identify the importance of multiple apps in the IoT. However, Melete focuses mainly on deployment at the network level, does not solve other app needs (e.g., resource security) and is less generic, being reliant on the specialized Maté VM.

Microsoft developed the Singularity OS [13] to investigate three architectural features which are similar to our work: software-isolated processes, contract-based channels and manifest-based programs. Processes are isolated through a combination of static verification and runtime checks, instead of memory management hardware. Communication between processes flows through a contract-governed channel and requires programs to specify a manifest describing requirements and capabilities similar to our approach. Singularity is based on a custom type-safe language, Sing#, and is designed for general computers. As such, they explore similar ideas but for other less-constrained platforms.

**Table 1. Comparison of popular IoT OSs in terms of multi-app device requirements.**

| OS | Remote Management | Memory Isolation | Resource Security | Preemptive Multi-Threading |
|---|---|---|---|---|
| Contiki | Over-the-air, dynamic loading | No | No | No |
| TinyOS | Over-the-wire, single image | No | No | No |
| RIOT | Over-the-wire, single image | Partly | No | Yes |
| FreeRTOS | Over-the-wire, single image | No | No | Yes |

TrustLite [14] is a security architecture for hardware-enforced isolation of software modules on embedded systems. Custom hardware enforces safe memory access control and loading and execution of untrusted software. It has a small trusted base, allowing even the OS to be untrusted, and can be instantiated with several levels of security. TrustLite provides protection and communication between multiple apps. In contrast to CerberOS, it does so without requiring trust in the OS through execution-aware memory protection. However, it is not generic and requires a specific hardware and software approach.

## 3.3 Interpreters for Constrained Devices

Interpreters have been adapted to work on constrained devices. Most research focuses on highlighting different approaches (e.g., high-level language VMs and thin hypervisors), on enabling specific features through interpretation (e.g., multitasking) or on proving the feasibility in terms of performance, energy or memory cost of interpretation. To the best of our knowledge, no research provides all the features necessary to support resource security.

Squawk [26] is a small Java virtual machine, written mostly in Java, that runs without an OS on embedded devices. Similar to CerberOS, Squawk implements app isolation mechanisms and highlights the benefits of Java on embedded devices. However, Squawk requires a specific board, the Sun SPOT, which is more powerful than Class-1 devices, and does not support the incorporation of other OSs.

Java Card [7] is a Java implementation that allows Java apps (applets) to securely run on smart cards. These typically have 1KB of RAM and 16KB of ROM which imposes severe restrictions on the language, e.g., no support for threads or garbage collection. Java Card focuses on security and portability of applets and introduces data encapsulation, cryptography and an applet firewall. Accordingly, it has a similar focus but is dedicated to a more niche purpose for very constrained devices and requires another system for support, the card reader. In contrast, CerberOS is a full IoT OS with no specialized hardware or software requirements.

## 3.4 Resource Management on IoT nodes

Device resource management is an important topic in the IoT but has not been widely explored for multi-app nodes.

VirtualSense [15] is a low-power wireless sensor based on the Darjeeling VM and Contiki [6, 9]. It focuses on energy as a resource with dynamic power management and advanced power-performance design. VirtualSense does not consider multiple apps or resource security, although it does support multitasking [16]. In addition, this work is not generic, being a custom solution for a specific OS and hardware. VirtualSense shows the feasibility of high-level language VMs on constrained devices to support interesting features, like power management, and is complimentary to our own work.

Pixie OS [19] is a sensor node OS for data-intensive applications. It implements policies for resource management, based on a data flow programming model using resource tickets, a representation of resource availability, and reservations. Pixie OS allows for a range of policies by giving the system control over resource management. Resource brokers reduce complexity by mediating between low-level physical resources and high-level application demands. CerberOS is similar, using resource buckets to manage access and an interpreter as a resource broker. Pixie OS differs in its data flow programming model which gives the OS control over application behavior. They focus on awareness of and adaptation to resource availability but do not consider resource security and only allow a single app running on the node. Consequently, it is an advanced approach to individual app resource management and could be interesting to adopt into our multi-app OS research.

Nano-RK [11] is a reservation-based real-time OS for wireless sensor networks. It supports preemptive multitasking and energy budgets. Tasks can specify their resource demands and the OS provides controlled access to CPU cycles and network packets. Nano-RK does not focus on security but is more concerned with timeliness and energy use. As such, it is an interesting point-of-reference for further developing our own resource management scheme.

# 4 Design

## 4.1 Overview Architecture

Our approach is based around three main concepts: (i) a *code interpreter* for instruction-level monitoring and managing running apps, (ii) *resource contracts* which strictly define app resource usage, and (iii) per-app *buckets* that provide fine-grained tracking of resource usage. As all app code execution is interpreted, the interpreter can perform instruction-level resource monitoring and control during interpretation using the contracts and buckets. These elements form a secure layer between the apps and the underlying system. In essence, an untrusted but monitored user space is decoupled from a trusted kernel space by a virtualization layer based on the interpreter. With this design, CerberOS meets the requirements and threat model from Sections 2.2 and 2.3. Specifically, the interpreter provides preemptive multi-threading and memory isolation and contracts and buckets solve the resource management requirements. The evaluation in Section 5 shows that we satisfy the resource efficiency requirement. Remote management is supported through over-the-air (OTA) code loading and standard network communications to retrieve node data. The remaining requirements, network security and access control, are provided by existing technologies.

CerberOS implements a lightweight Java Virtual Machine (JVM), a high-level language VM, to serve as the interpreter. Our primary motivations for this choice were:

- **Proven track record.** Embedded JVMs have a strong track record in networked embedded systems. Key examples include: Squawk, Darjeeling and Java Card [26, 6, 7]. Prior work shows that it is feasible to realize JVMs within the constraints of embedded systems.
- **Virtualization for security on shared platforms.** Virtualization approaches are a common way in mainstream, non-embedded systems to improve security between programs running on the same hardware [23].
- **Benefits of high-level language programming.** Developing IoT apps in a common high-level language provides improved developer productivity, smaller file sizes and portable, platform-independent apps. Other benefits include: type safety, exception handling, garbage collection, protection against common programming errors and a large developer community.

CerberOS incorporates and extends *uJ*[1], a micro JVM (µJVM) for microcontrollers. We adapted uJ extensively for our purposes. Primary changes include: preventing apps from executing plain C code, support for multiple Java apps with scheduling and runtime management, mandatory specification of resource contracts and a resource management scheme through buckets.

Figure 2 shows the basic elements of CerberOS. CerberOS is the OS running on the device and consists of a user and kernel space. The user space contains the untrusted apps, each with a contract. The kernel space contains the interpreter, i.e., the virtualization layer or the µJVM, and board support package (BSP). The interpreter divides the insecure user and secure kernel space, provides communications between apps and monitors and controls resources through buckets. The BSP is a layer of indirection between the boards native hardware and software and the rest of CerberOS and provides access to the software and hardware of the board. This facilitates adaptation of CerberOS to different boards and platforms by simply changing the BSP. Apps are loaded on a device over the network through an IoT gateway after authentication. This gateway accepts, compiles and uploads *.java* files to the device. These elements are further explained in the following sections.

## 4.2 Basics of the µJVM

From the viewpoint of a secure multi-app OS, we consider the level of Java compliance of the µJVM, how app files are produced for and used by the µJVM and how they are securely loaded on the node. Since the interpretation of Java bytecode is not the focus of the paper, we do not discuss other VM details in-depth. For more details, we refer to the uJ website[1] and related VMs [1, 6].

### 4.2.1 *Java compliance*

The µJVM is modular and can selectively support Java features to optimize the VM for size and speed. CerberOS

[1] An open-source project. Available on: `http://dmitry.gr/index.php?r=05.Projects`.

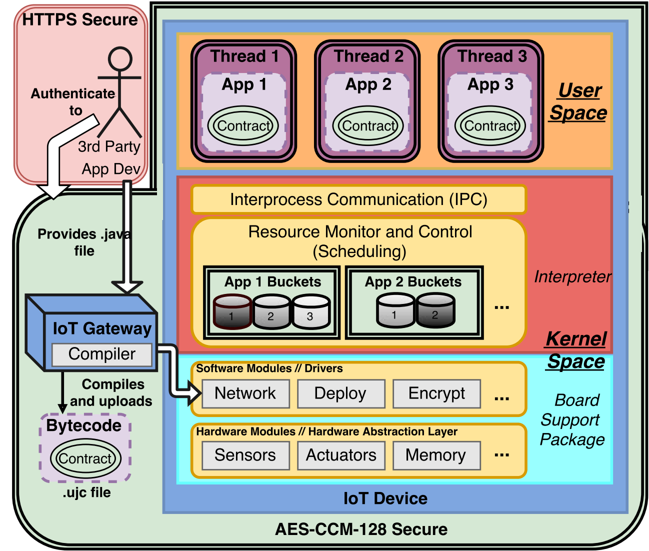


**Figure 2. Overview design of CerberOS.**

supports the Java ME framework [22]. Full-featured CerberOS is compliant with the CLDC 1.1 specification and supports features like exceptions, inheritance and floating point calculations. CerberOS can sacrifice support for these features, and thereby CLDC-compliance, for a smaller footprint. Note that the requirements in Section 2.2 are preserved regardless of the enabled Java features. The memory impact of the level of Java compliance is analyzed in Section 5.1.

### 4.2.2 *Application file format*

The µJVM executes bytecode stored in *.ujc* files. Apps are created by developers as *.java* files. They are then compiled on the IoT gateway to bytecode in a *.class* file which is further processed to optimized bytecode in *.ujc* files. *.ujc* files are smaller in size and faster in execution speed due to several optimizations such as discarding unused constants and creating a table of contents for method and field lookups. The gateway uploads the *.ujc* files to the device. App files in Java are inherently smaller than C or native code apps since a high-level language instruction often maps to multiple low-level language instructions, thereby requiring fewer instructions taking up less space for the same functionality. Combined with the further optimization to *.ujc* files, this can lead to significantly smaller file sizes. App file sizes are further evaluated in Section 5.3.

### 4.2.3 *Secure code loading*

CerberOS provides over-the-air (OTA) app loading which allows for flexible and scalable remote app management and adaptation of IoT platforms. OTA loading is done by initiating a transfer through a predetermined message and then storing the files in flash and loading them when the transfer is complete. Only authenticated parties are able to upload apps and file transfers confirm to the classic information security (confidentiality, integrity and availability) principles through basic cryptography practices. Specifically, the third-party app developer authenticates himself to the IoT gateway using HTTPS and provides the *.java* file which is deployed securely using the AES-CCM-128 security suite [29]. The

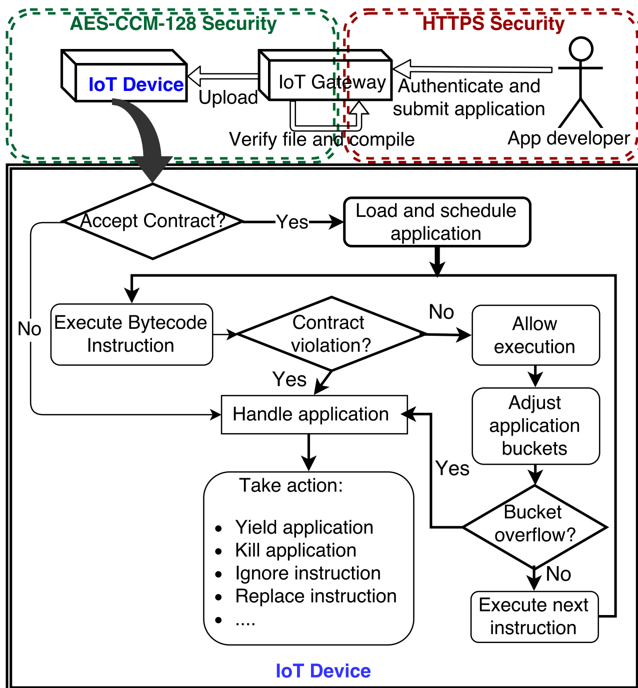


**Figure 3. Lifecycle of an app. The top shows loading and scheduling of apps and the bottom the monitored flow of a bytecode instruction through the interpreter.**

upper part of Figure 3 shows the file transfer and loading part of the app lifecycle.

## 4.3 Resource Contracts and Buckets

App developers specify a *resource contract* in the code of the app before submitting the app to a node. The contract determines the resources the app requires from a device, similar to a service level agreement. Resource contract are extensible, lightweight and user-friendly to, respectively, cover the device capabilities, to be usable by IoT nodes, and be understandable by app developers. The upper part of Figure 3 shows the loading and contract verification part of the app lifecycle. At load time, the node verifies if it can satisfy the contract while staying within its resource constraints. If it can, the device loads the app, provides it with the requested resources and schedules it in the scheduling queue. If it cannot, the device rejects the app and does not load it.

The abstract `UJApp` Java class gives the contract format through methods that indicate the specifiable resources. Apps extend `UJApp` and implement its abstract methods, specifying their desired resource amount or access through static fields. Listing 1 shows a sample of `UJApp`. The `getROMMemorySize()` method indicates that apps need to specify their ROM memory requirements. Listing 2 gives a sample of an example app which extends `UJApp`. The app specifies its desired ROM memory size and implements the relevant abstract method from `UJApp`. `UJApp` simplifies contract creation for developers by serving as a template.

CerberOS implements a proof-of-concept version of the contract where an app specifies resources and capabilities. Table 2 shows the current resources an app contract can specify and what they mean. This specification does not capture the full complexity of IoT peripherals or resources which is important future work. However, the current contract implementation allows apps to specify the required memory, sensor and network access and the needed priority and lifetime of the app on the device, which is necessary for common IoT scenarios. In addition, this implementation is rich enough to enable leasing of IoT hardware to third parties with different levels of priority and resource needs.

```
public abstract class UJApp{
  public abstract int getROMMemorySize();
  public abstract boolean[] getPeriphAccess();
  ...
}
```

**Listing 1. Code sample of the UJApp Java class.**

```
public class ExampleApp extends UJApp{
  public static int ROM_MEMORY_SIZE = 300;
  public static byte SENSOR1 = LIGHT_SENSOR;
  ...
  @Override
  public int getROMMemorySize() {
    return ROM_MEMORY_SIZE;
  }
  @Override
  public boolean[] getPeriphAccess() {
    //implementation
  }
}
```

**Listing 2. Code sample of an example app.**

The interpreter enforces app contracts through per-app per-resource *buckets* that track resource usage. The size of a bucket is determined at load time by the contract. For example, a bucket can track how long an app has existed on the device or how much bandwidth it has consumed. In essence, a contract specifies allowed capabilities and a bucket the measure in which they have been used. Both contracts and buckets are checked during execution. For instance, app network communications is first allowed by the contract and then checked through the bucket to determine if the app is within its allotted bandwidth consumption.

In conclusion, every loaded app on a node has a linked contract and associated buckets that determine what resources are accessible to the app, in what quantities and in what measure the app has used the resource.

**Table 2. Possible app contract resources, their type and what they specify.**

| Resource | Type | Specifies |
|---|---|---|
| RAM | value | Amount of dynamic memory for the app |
| ROM | value | Amount of storage memory for the app |
| Peripherals | list | Peripherals the app has access to |
| Network | value | Amount of bandwidth for the app |
| Priority | 0-255 | App priority in the scheduling queue |
| Lease | value | Amount of time the app is on the device |

## 4.4 Instruction Monitoring and Execution

Key to CerberOS is runtime interpretation of the code for instruction-level monitoring of apps. During bytecode execution of an app, the interpreter first checks the contract and the buckets related to the app to determine if it is allowed to execute that bytecode. Resource checks have a modifiable granularity, depending on the bytecode. For example, CerberOS can check every ten instructions if the allowed number of operations has been reached. This is a trade-off between accuracy and the overhead of instruction-level monitoring. If the contract is violated, CerberOS prevents further execution and handles the app (e.g., by removing it or replacing the instruction). If the instruction is allowed, it interprets the bytecode and executes it. The bottom part of Figure 3 shows the flow of a bytecode instruction through the interpreter.

## 4.5 CerberOS API and Platform Support

CerberOS is a general solution that works with various hardware platforms and OSs and provides Java apps controlled access to board functionalities. Apps use the API from `UJApp` to, for example, set timers or interact with peripherals. This layer of indirection between apps and the μJVM simplifies adapting apps to other platforms. The μJVM makes select native C functions accessible to `UJApp` through Java stub classes which serve as drivers for `UJApp` and the extending apps. They contain no implementation but form the link to the BSP. The BSP is the layer of indirection between the μJVM and the hardware board and underlying OS functionality, if any. Integration with another OS is optional. If not desired, the BSP itself needs to implement desired features on top of the hardware board such as scheduling, timers and networking.

Accordingly, porting to a platform requires four changes: (i) creating a BSP which accesses the OS and board features, (ii) integrating the μJVM in the OS scheduler, if present, or scheduling itself if not, (iii) implementing Java stub classes to make them accessible to `UJApp`, and (iv) extending `UJApp` with the new functionalities, if any.

We have implemented CerberOS with an integration of Contiki on the μPnP board [30] with an ATmega1284P microcontroller (MCU), as a stand-alone edition for the Zigduino platform with an ATmega128RFA1 MCU and for a regular Windows PC. For the stand-alone version, the BSP provides direct access to the devices hardware. Only CerberOS is running and controls its own timers, scheduling and duty-cycling. For the Windows version, the BSP provides standard board features using the Windows API. For the μPnP version, the BSP leverages existing Contiki and μPnP features. Here, the μJVM is a process that runs on Contiki and is scheduled in its cooperative process design [9] and uses the μPnP network stack. The μPnP version with Contiki is our main test and evaluation platform.

## 4.6 Scheduling and App Management

There are two levels of scheduling: at the VM-level for apps and at the OS-level for the μJVM and other OS tasks. VM scheduling is multi-threaded where apps (i.e., threads) run in their own context, manage their own stack and can be context-switched by yielding or preemption. Therefore, CerberOS provides preemptive multi-threading for apps even when the BSP underneath uses an event-driven scheduler. OS scheduling can be implemented by CerberOS or through the integrated OS. CerberOS uses the scheduler in Contiki which allows to suspend the μJVM when all apps are idle.

Apps are scheduled consecutively in a fixed circular queue, in order of load time on the node. An app executes for a time slice, defined by the priority specified in its contract, and then the scheduler switches to the following one. Since the position of an app in the queue is fixed, all apps are guaranteed to run. An app can also yield, wait or terminate itself. Yielding stops the execution of the app, allowing the next one in the queue to go. Waiting is based on events such as communications or timers and means the app will be skipped in the queue while it is waiting. If the event occurs during the execution of another app, the app will stop waiting and run the next time it is reached in the queue. If all apps are waiting, the node sleeps to conserve power.

We do not support real-time requirements for apps as-of-yet. However, a node controls its runtime by managing the apps and can therefore control the timing of app execution. It could support soft-real time requirements by being strict in what apps it accepts. Exploring scheduling and timing in terms of apps and their contracts is future work.

# 5 Evaluation

We evaluate CerberOS on key metrics of constrained nodes: memory size of the base image, execution performance, app code size and battery lifetime. We analyze these metrics for representative benchmarks and two realistic IoT use cases, including a multi-app deployment.

We implemented CerberOS on top of μPnP, a full-featured IoT platform [30], with Contiki 3.0 on the ATmega1284P board, an 8-bit MCU with 128KB flash, 16KB RAM and clocked at 10MHz. μPnP motes offer 802.15.4e mesh network support via a SmartMesh IP radio [28]. Evaluation on μPnP provides an efficient network stack and access to Contiki features such as timers and an I/O API. In addition, it demonstrates the feasibility and generality of our approach to enable resource-secure multi-app IoT platforms on representative constrained hardware and software. Finally, we compare CerberOS with the TakaTuka and Darjeeling JVMs as a point of reference. Metrics and figures for these JVMs come from existing research [1, 6, 10, 20]. They are not directly comparable since they use different platforms, apps and implementations and support other features. However, the comparison serves to provide insight into overall performance of embedded VMs and to further demonstrate the reasonableness of our resource-secure μJVM.

## 5.1 Memory Impact of CerberOS

CerberOS requires additional RAM and ROM on top of the BSP and, if applicable, integrated OS. This analysis does not consider the memory cost of apps, which is discussed in Section 5.3. CerberOS is modular and can turn off support for Java features, such as exceptions or 64-bit floating point operations. Therefore, we compare three cases: (i) the memory size of a minimal implementation of μPnP, (ii) the memory size of a minimal CerberOS implementation which drops support for floating point calculations, 64-bit long values, advanced exception handling and Java thread synchro-

Table 3. **Memory cost in ROM and RAM in bytes for versions of CerberOS and their difference.**

| | ROM | RAM | ΔROM | ΔRAM |
|---|---|---|---|---|
| Minimal μPnP | 9866 | 1731 | - | - |
| Minimal CerberOS | 39176 | 4574 | 29310 | 2843 |
| Full CerberOS | 64458 | 4616 | 25282 | 42 |

**Table 4. Execution times of C and Java apps in milliseconds and their ratio.**

| | C | Java | Java/C Ratio |
|---|---|---|---|
| Encryption | 3.963ms | 205.420ms | 51.829 |
| Storage | 166.1ms | 169.6ms | 1.024 |
| Network | 12.074ms | 15.782ms | 1.307 |
| Sensing | 69.260ms | 71.150ms | 1.027 |

nization, and (iii) the memory size of a full CerberOS implementation with CDLC-compliance. Minimal μPnP features a network stack, timers and Contiki scheduling functionality. CerberOS integrates this with our design (discussed in Section 4) and provides multi-app management, resource security and Java support. We compile with the "avr-gcc" toolchain, version 4.9.2, and optimize for size and speed. Code size is given by the "avr-size" tool. App file size is evaluated separately in Section 5.3.

Table 3 shows the three cases in terms of ROM and RAM and the difference between consecutive versions. μPnP uses around 10KB ROM and 2KB RAM. Minimal CerberOS requires an additional 30KB ROM and 3KB RAM. Full CerberOS needs an extra 25KB ROM and negligible added RAM. The total size of the full-featured solution is around 65KB ROM and 5KB RAM. Accordingly, CerberOS is feasible for Class-1 devices [5] and a minimal implementation can even be used on more constrained devices.

The memory cost of TakaTuka and Darjeeling is around 40KB-60KB ROM and 0.5-4KB RAM, depending on the app. These figures are comparable with CerberOS, taking into account the level of Java compliance and trade-offs made between the μJVMs. TakaTuka achieves its low RAM usage by customizing JVM bytecode support down to the set used by the given program. This complicates adapting the platform over time and prevents our security features.

## 5.2 CerberOS Execution Performance

CerberOS impacts software execution performance due to bytecode interpretation overhead for individual apps and context switch time between apps.

### *5.2.1 Bytecode interpretation performance*

We measure the performance overhead of interpretation in six benchmarks, implemented in both Java and native Contiki C apps. The apps are encryption, storing an array, sending a network packet and reading a sensor, which are core IoT device functions; and bubble sort and pin toggle, which serve to determine the μJVM performance characteristics. Fine-grained timings are obtained with a logic analyzer. We use the minimal CerberOS implementation for evaluation.

The core IoT function benchmarks reveal how CerberOS performs for typical IoT device duties. The encryption test measures the time to encrypt and decrypt a value. This is commonly the most computationally heavy task a node does for standard IoT apps. We use the lightweight Speck block cipher [3], developed by the NSA for high performance on constrained hardware and software, and implement it completely in Java. Speck has been tested on our range of platforms (8-bit ATmega) and generally outperforms other IoT block ciphers [8]. The test uses a 128-bit key to encrypt a 64-bit block consisting of two random 32-bit values. The storage test measures the time to write a 50-byte array as a block to flash. The network test measures the time to send a 50-byte payload packet to the gateway. Only the time to send the packet is measured, since the response time is dependent on the network. The sensor reading test is a sensor measurement from a sensor peripheral. It measures the time to communicate over the I2C bus to get the sensor value.

Table 4 shows the execution time of the core IoT function apps in milliseconds and the Java/C ratio. The storage, network and sensing evaluation show comparable execution times despite the overhead of virtualization since they call native functions made accessible by the BSP. The overhead incurred is through code interpretation and the resource contract checks on this call. This penalty is significant for short tasks, such as a 30% overhead for sending a packet, but remains feasible, in the order of milliseconds. In contrast, the compute-heavy test, encryption, has a significant overhead of about 50x when implemented completely in Java.

The bubble sort benchmark sorts an array of 256 decreasing 32-bit integers, the worst-case bubble sort scenario with a time complexity of $O(n^2)$. This test is not typical of the duties performed by sensor and actuator nodes but presents a worst-case scenario for a computationally heavy task. Bubble sort has a performance penalty of about 300x, $0.164s$ for C and $46.99s$ for Java, when implemented completely in Java with no native function calls. Therefore, we consider this the worst-case scenario but unrepresentative for typical IoT apps. The pin toggle benchmark measures the time for a function call to pass through the interpreter. Since we are measuring the time of a trivial operation in C and comparing that to an interpreted and controlled operation in Java, it shows a performance penalty of about 1000x, $0.7\mu s$ for C and $0.779ms$ for Java. This time reveals that the time to call a Java function takes around a millisecond and so forms the lower-bound for any Java operation done by apps.

The cost of interpreting Java versus executing native C code is similar in both Darjeeling and TakaTuka for the worst-case bubble sort example. Darjeeling performs better, with a cost of up to 113x, and Takatuka performs similarly, up to 324x slower. For other apps, the results are similar across the VMs: for compute-heavy tasks a performance penalty of two orders of magnitude is possible.

Evaluation indicates that an interpretation approach is less-suited to compute-heavy tasks implemented completely in Java. However, these tasks are relatively rare in IoT scenarios due to their impact on battery life. Battery lifetime and app execution time are further compared in Section 5.4. In addition, it is possible to implement common compute-heavy algorithms in C and make them accessible to apps as a board call. Our evaluation shows that this will achieve com-

**Table 5. App ROM code sizes in bytes for optimized Java bytecode and for C Contiki ELF format.**

| | C Code | Java Bytecode | Difference |
|---|---|---|---|
| Encryption | 3146 | 1392 | -1754 \| -55.8% |
| Storage | 1894 | 1108 | -786 \| -41.5% |
| Network | 2090 | 1296 | -794 \| -38.0% |
| Sensing | 1684 | 963 | -721 \| -42.8% |
| IoT App A | 3207 | 2037 | -1170 \| -36.5% |

parable execution times as when it is implemented purely in C, for example, as in the storage test. Note however that this analysis does reveal that apps with many Java operations or rapid response requirements are out of scope for CerberOS.

*5.2.2 App context switching performance*

Context switching between apps introduces overhead because the scheduler needs to execute and a new app stack needs to be loaded. We measure the time required to switch between apps for different amounts of total apps. The evaluation reveals that context switching takes around 2.16*ms* and that the difference is negligible for more apps, increasing by just 0.98% when going from two apps to seven apps.

## 5.3 App Code Size and Deployment

This section evaluates the code size of apps in CerberOS for ROM and RAM and the impact of this size on deployment time and energy cost. For memory, we only consider the default cost of apps in both ROM and RAM and not the memory resource amount they specified in their contracts since this is variable. Deploying apps on a node efficiently requires OTA code transmission. Thus, deploying apps costs time and consumes bandwidth and energy on the receiver and all intermediate routing nodes. In addition, the destination node has to accept and store the app which costs memory and energy. Therefore, app code size is important for multi-app IoT platforms. In CerberOS, apps are coded in Java, a high-level language, and compiled to an optimized bytecode format which can be directly loaded on a node. Since high-level languages are more compact, this will result in reduced app code sizes, as discussed in Section 4.2.2. For native Contiki apps, apps are coded in C and have to be compiled into an ELF module format before they can be dynamically loaded, which incurs a significant size overhead [25].

Table 5 shows the app ROM code sizes of the core IoT functions benchmarks and for a representative IoT app (IoT App A, see Section 5.4). As expected, bytecode requires between 36%-55% less bytes, even though apps in CerberOS have an additional cost in code size because they specify contracts. The contract has a fixed one-time cost of around 400 bytes, a significant relative impact for small apps but less important as they increase in size. App ROM code size is the size of the code that has to be transmitted OTA.

Apps also require a default amount of RAM to work. Every app is a thread which introduces a minimal cost of 218 bytes for the stack. Each app also incurs a non-fixed RAM cost for the contract and buckets. In practice, an app costs between 400 to 600 bytes of RAM, depending on the resources it specifies. Therefore, we support up to seven coexisting apps, since permitting more brings us outside the scope of

**Table 6. App deployment time to node in seconds, node energy cost in Joule for C and Java apps and difference.**

| | Time (s) | | Energy (J) | | Difference (%) |
|---|---|---|---|---|---|
| | C | Java | C | Java | |
| Encryption | 105.33 | 49.16 | 2.16 | 1.01 | −53.2% |
| Storage | 65.24 | 40.06 | 1.34 | 0.82 | −38.9% |
| Network | 71.51 | 46.09 | 1.46 | 0.94 | −35.6% |
| Sensing | 58.51 | 35.42 | 1.20 | 0.73 | −39.2% |
| IoT App A | 107.28 | 69.82 | 2.20 | 1.43 | −35.0% |

Class-1 devices when accounting for the 5KB RAM cost of CerberOS itself. Up to ten concurrent apps have been tested but can exceed device resource limits.

We obtain energy cost estimates for app deployment using previous research which is based on a 50-node µPnP-Mesh test bed with a maximum hop depth of 3 [21, 25, 30]. Table 6 gives the time and energy for deploying Java and C apps. As expected, Java apps deploy faster and consume less energy, up to 50% savings. This shows the benefits of compact high-level language apps for IoT deployments.

The size of these benefits depends on the frequency of app reconfiguration and app size. As an illustration, we consider the energy saved with CerberOS when deploying a single app. Our analysis shows this varies between 0.5 and 1 Joules for relatively small apps. The µPnP board has a 3000*mAh* battery at 3.2V, which stores 34560 Joules. If deploying costs 1 Joule less, it saves 0.0029% of the battery capacity per code deployment for the reconfigured node. In dynamic use cases over a multi-year timespan, these savings accumulate. For instance, saving 1 Joule a day saves 5.3% of the battery capacity over five years. In addition, this analysis does not consider the savings at the intermediate routing nodes in a multi-hop network. A node can route for many other nodes which means the above savings accumulate each time one of its children nodes is reconfigured. Analogous to the above scenario, when routing for 10 such nodes, the routing node would save 10 Joule a day, or 53% of the battery capacity over five years.

## 5.4 Realistic IoT App Analysis

This section analyzes two representative IoT apps, *IoT App A* and *IoT App B*, to measure CerberOS in two different realistic use case scenarios. The IoT App A evaluation focuses on a single IoT app to explicitly compare the characteristics of a Java implementation to a C implementation. The IoT App B evaluation focuses on the impact of multiple Java apps on a constrained device to explicitly compare a multi-purpose device to a single-purpose device.

*5.4.1 Single IoT App A analysis*

This section compares the execution time, network latency and battery lifetime of an IoT device with a single Java IoT app to that of devices with a single C IoT app. The app is an example of environment monitoring and combines core IoT functions. It receives temperature requests from a gateway, senses the temperature, encrypts the value, stores it in memory, reads it from memory and replies to the request with the value. The node sleeps between requests. We implemented the app in Java and C and deployed it alone on

the same hardware as before in a single-hop mesh network. The code size and deployment cost are analyzed in Section 5.3. Here we consider the execution time and its effect on response latency and battery life time.

The time between request arrival and response transmission is 0.243$s$ for the C app and 0.659$s$ for the Java app, an overhead of 2.712x. Response latency is higher due to the longer processing. However, this cost is small relative to the total round-trip time of the request. Due to the variable nature of mesh networks, this time is not deterministic. Even in our single-hop network it varies between 0.5$s$ and 2.5$s$. As the added time for the Java version is smaller than the latency variability, we consider CerberOS feasible in terms of network latency.

Battery lifetime depends on the sampling frequency of the gateway, which determines the duty cycle and the average current. Analysis reveals that the µPnP board consumes around 10$\mu A$ in sleep and 3.5$mA$ while awake, for which we use worst-case figures. The average power consumed by the accompanying SmartMesh IP board is 40.5$\mu A$, determined through the 'SmartMesh Power and Performance Estimator'[2] as used in [28]. Accordingly, consumed power during sleep is 50.5$\mu A$ and while awake is 3.54$mA$. From these figures, we determine the battery lifetime in function of how often the device is sampled. Figure 4 charts the lifetime of a node with a 3000$mAh$ battery in terms of a logarithmic sampling rate expressed in seconds. The impact of CerberOS is highest for fast sampling rates, halving the lifetime for sampling rates faster than once every 10 seconds. Beyond this point the impact decreases rapidly, as the node is sleeping longer. For a 100-second sample rate, the relative reduction is 20%, for a 4-minute rate, 10%, and for a 15-minute rate, 3%. Slower sampling quickly leads to a low energy impact which becomes negligible for sampling rates above hours. Even for fast sample rates, CerberOS achieves multi-month lifetimes and does not exceed more than a 60% reduction compared to a native implementation.

### 5.4.2 *Multiple IoT App B analysis*

This section analyzes the impact of running multiple apps on a single device in terms of performance and battery lifetime. Ideally, the impact scales linearly with the amount of apps. In practice, there is additional overhead due to scheduling and app management. We test CerberOS with the same Java IoT app deployed one, three, five and seven times. The Java apps are compared against the ideal case where additional apps require no extra overhead. We do not compare against multiple C apps since concurrent apps in Contiki are not usable in realistic use cases, due to the lack of preemptive multi-threading and as discussed in Section 2.2, and because the ideal case is a stronger comparison. We do analyze the battery lifetime of a single-purpose device with a single C app. The performance and lifetime of a single-purpose C app device compared to a multi-purpose Java app device is the basic economic argument for sharing IoT devices with CerberOS.

*IoT App B* is, similar to the one in the previous section, representative of real use case IoT apps. It senses the humidity every user-specified period, encrypts the value, stores and reads it, does I/O to a local screen and sends the value to the gateway. Here we consider the increase in execution time and its effect on the battery lifetime as more apps are deployed and the sampling rate is increased.

[2] Available on: `http://www.linear.com/docs/42452`.

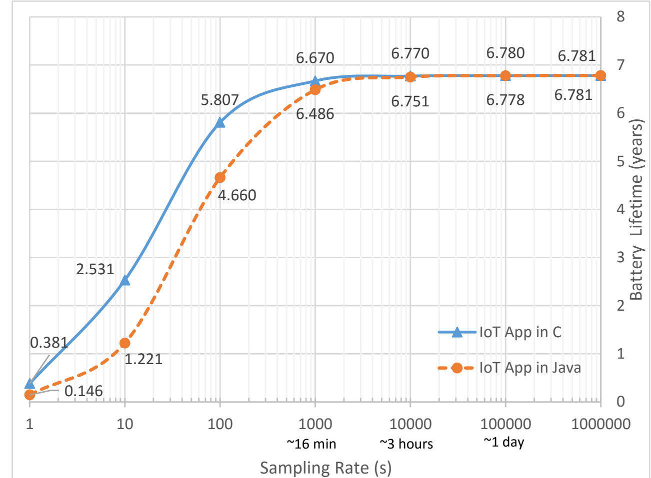


**Figure 4. Battery lifetime in years for node sampling rates in seconds with IoT app A.**

**Table 7. Execution times of a C app and Java apps in seconds and the overhead compared to linear scaling.**

| #Apps | Tot. Exec. time | Overhead to Linear Scaling | |
|---|---|---|---|
| 1 C App | 0.101s | N/A | N/A |
| 1 Java App | 0.477s | 0.000s | 0% |
| 3 Java Apps | 1.560s | 0.128s | 9% |
| 5 Java Apps | 2.637s | 0.250s | 10% |
| 7 Java Apps | 3.716s | 0.374s | 11% |

Table 7 shows the total execution time for all the apps to run consecutively and the absolute and relative difference of that time compared to the time of a single app linearly scaled to the total amount of apps. A single Java app requires 0.477$s$ to run to completion while the C app requires 0.101$s$ for this case. Scheduling, context switching and managing three, five or seven Java apps causes additional overhead beyond linear scaling which is adding the extra app execution times. The relative size of the overhead increases as more apps are added since scheduling and managing more apps takes longer. For the worst-case seven app scenario, this overhead can be up to 0.374$s$. The increase in overhead is reasonable when compared to linear scaling and varies between 9% for the three app case to 11% for the seven app.

Important is the feasibility of running multiple apps on a constrained device when compared to a single-purpose device. Figure 7 shows how multiple apps compare in terms of battery lifetime and sampling rate to a device with a single C app. For this, we reuse the same energy figures as in Section 5.4.1. As before, battery lifetime largely depends on the sample rate. For a 10-second sample rate, the C app allows for a 4-year battery lifetime, the single Java app still manages a 1.58-year lifetime but the seven Java app case reduces lifetime to 3 months which is a 16x reduction in lifetime

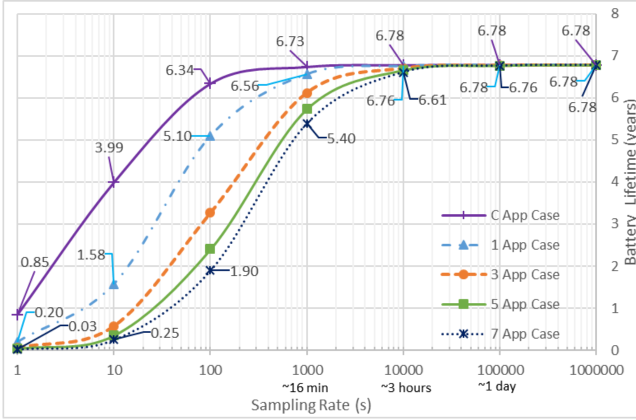


**Figure 5. Battery lifetime in years for node sampling rates in seconds for multiple apps (IoT app B).**

compared to C. However, as before, with slower and more realistic sampling rates, the impact quickly reduces. For a 100-second sample rate, the seven app case reduces lifetime to a third while supporting seven times the amount of apps, in Java. For a 15-min sampling rate, the reduction in lifetime for the seven app case reduces to 20%. For a (multi-)hour sampling rate, the impact becomes negligible ($< 5\%$).

### 5.4.3 *Discussion on feasibility*

Evaluation shows that CerberOS is possible for multi-app IoT deployments in two different realistic use cases depending on the sampling rate (whether request/reply- or period-based). Common sample rates in the non real-time embedded IoT world are in the order of minutes to hours for which the analysis shows that the energy impact of Java apps is feasible. Furthermore, the results reveal that, even for multiple apps, battery lifetime is determined mainly by the sampling rate and scheduling, context switching and app management of multiple apps incurs an acceptable increase in execution time overhead. Finally, while the reduction in lifetime for the seven app case for faster sampling rates ($<$ 10-minutes) is significant, the device supports seven apps instead of one which can be a net benefit, depending on the use case. These results prove that a multi-purpose device with CerberOS is feasible compared to single-purpose devices while unlocking several benefits such as the use of a high-level language, resource security and secure shareable infrastructure.

## 5.5 Security Discussion

In this section, we informally discuss that CerberOS meets the threat model as described in Section 2.3. Specifically, we argue that the three identified attacks by a deployed malicious app are impossible on a system that is designed like CerberOS. These attacks are: (i) rendering the device inoperable, (ii) disrupting other apps and (iii) modifying or stealing data from other apps.

**Rendering the device inoperable.** An app can try to break the device through three broad categories of attack: (i) crashing through execution of nonsensical instructions, for instance, divide-by-zero operations, (ii) by writing into essential memory spaces, such as the OS, and (iii) by depleting the battery deliberately. The first attack is prevented since an app can only execute Java code. This code is interpreted and, if it is nonsensical, the device will not execute it. The second attack is prevented since apps are isolated; they only have access to their own reserved memory space and are prevented from accessing other memory, such as where the OS resides. The third attack is prevented since an app can only execute a contractually limited number of operations (e.g., as determined by priority) on the CPU, network, etc. It cannot keep the device busy for longer than the agreed upon contract; once it reaches its resource limits it is stopped from executing further.

**Disrupting other apps.** An app can try to attack another app on the device through two broad categories of attack: (i) breaking the other app by directly overwriting its code, and (ii) interfering with the other app by blocking or stealing its essential resources, such as hogging the sensors the other app requires. The first attack is impossible, since, as discussed before, concurrent apps are isolated and cannot write to outside memory. The second attack is prevented since CerberOS ensures resource security. The app cannot permanently block resources so that other apps have no access since its resource usage is tracked by buckets.

**Modifying or stealing data from other apps.** An app can try to steal or modify data from another app, for instance, by modifying sensor readings or looking at stored app data. This is prevented since apps are isolated from each other, therefore, an app cannot read or write in the address space of another app. Furthermore, an app cannot influence sensor readings since the app is also isolated from the OS and hardware board. Thus, an app cannot break the confidentiality or integrity of the data of another app.

# 6 Conclusions and future work

This paper presents *CerberOS*, the first resource-secure OS for multi-app IoT devices. CerberOS maximizes RoI on IoT deployments by supporting secure shared platforms between multiple parties and stakeholders. CerberOS realizes the vision of secure tiny multi-app embedded nodes by providing not only strong guarantees on the confidentiality and integrity of app data but also guarantees on the provision and use of key resources such as computation, storage, sensors, actuators and energy.

A virtualization layer, mandatory contract specification and resource tracking through monitoring buckets allow for the management and security of coexisting apps with fine-grained resource management. Together, these elements meet the discussed requirements and provide an answer to the threat model. Furthermore, the interpreter enables the use of Java for app development, unlocking high-level language benefits such as increased developer productivity and portability. CerberOS works on IoT nodes as constrained as Class-1 devices and requires no special hardware or software modules but can be implemented stand-alone or integrate popular IoT OSs such as Contiki. Results show that CerberOS has a comparable memory cost and execution performance to other similar systems while also supporting multiple apps and resource security. In addition, we have demonstrated that app bytecode deployment is more efficient in terms of time, energy and memory savings at the node. Finally, an analysis

of representative IoT apps shows that the energy impact of our system is feasible for real use cases, even when considering multi-app deployments.

Since CerberOS is a fully fledged OS, there are plenty of interesting opportunities for future work. Primary future work includes expanding the resource contracts and buckets to better handle the heterogeneity in the IoT and implementing the ability to handle soft real-time requirements. Also of interest is exploring how the secure resource management scheme can be made more dynamic and incorporate context awareness to further improve resource security, e.g., in the case of outside denial-of-service attacks. Finally, we would like to leverage existing research for high-level languages to unlock more of their benefits on embedded devices. An example is doing bytecode analysis to help predetermine node and app behavior. To conclude, we are planning to release the source code in the future with the aim to build a community of app and OS developers around CerberOS.

# 7 Acknowledgments

This research is partially funded by the Research Fund KU Leuven and the imec High Impact Initiative Distributed Trust. The authors thank Dmitry Grinberg for allowing the use and adaptation of uJ, available on `http://dmitry.gr`.

# 8 References

[1] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rührup, and Z. A. Uzmi. Optimized java binary and virtual machine for tiny motes. In *Distributed Computing in Sensor Systems*, pages 15–30. Springer, 2010.

[2] R. Barry. Freertos, a free open source rtos for small embedded real time systems. `http://www.freertos.org/`.

[3] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The simon and speck lightweight block ciphers. In *Proc. of the 52nd Annual Design Automation Conference*, page 175. ACM, 2015.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proc. of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[5] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. RFC 7228, RFC Editor, May 2014. `http://www.rfc-editor.org/rfc/rfc7228.txt`.

[6] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proc. of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 169–182. ACM, 2009.

[7] Z. Chen. *Java card technology for smart cards: architecture and programmer's guide*. Addison-Wesley Professional, 2000.

[8] D. Dinu, Y. Le Corre, D. Khovratovich, L. Perrin, J. Großschädl, and A. Biryukov. Triathlon of lightweight block ciphers for the internet of things. *IACR Cryptology ePrint Archive*, 2015:209, 2015.

[9] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[10] J. Ellul. *Run-time compilation techniques for wireless sensor networks*. PhD thesis, University of Southampton, 2012.

[11] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-rk: an energy-aware resource-centric rtos for sensor networks. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10–pp. IEEE, 2005.

[12] O. Hahm, E. Baccelli, M. Günes, M. Wählisch, and T. C. Schmidt. Riot os: Towards an os for the internet of things. In *Proc. of the 32nd IEEE International Conference on Computer Communications (INFOCOM), Poster Session*, 2013.

[13] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[14] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proc. of the Ninth European Conference on Computer Systems*, page 10. ACM, 2014.

[15] E. Lattanzi and A. Bogliolo. Virtualsense: A java-based open platform for ultra-low-power wireless sensor nodes. *International Journal of Distributed Sensor Networks*, 2012, 2012.

[16] E. Lattanzi, V. Freschi, and A. Bogliolo. Supporting preemptive multitasking in wireless sensor networks. *International Journal of Distributed Sensor Networks*, 2014, 2014.

[17] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[18] Linux Foundation. Introduction to the zephyr - zephyr project documentation; 2016. `https://www.zephyrproject.org/about`, Last access: September 2016.

[19] K. Lorincz, B.-r. Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource aware programming in the pixie os. In *Proc. of the 6th ACM conference on Embedded network sensor systems*, pages 211–224. ACM, 2008.

[20] O. Maye and M. Maaser. Comparing java virtual machines for sensor nodes. In *Grid and Pervasive Computing*, pages 181–188. Springer, 2013.

[21] G. S. Ramachandran, N. Matthys, W. Daniels, W. Joosen, and D. Hughes. Building dynamic and dependable component-based internet-of-things applications with dawn. In *Proc. of the 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, number 19, 2016.

[22] R. Rischpater. *Beginning Java ME Platform*. Apress, 2008.

[23] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226. IEEE, 2010.

[24] L. Sanchez, J. A. Galache, V. Gutierrez, J. M. Hernandez, J. Bernat, A. Gluhak, and T. Garcia. Smartsantander: The meeting point between future internet research and experimentation and the smart cities. In *Future Network & Mobile Summit (FutureNetw), 2011*, pages 1–8. IEEE, 2011.

[25] G. Sankar Ramachandran, W. Daniels, N. Matthys, C. Huygens, S. Michiels, W. Joosen, J. Meneghello, K. Lee, E. Canete, M. Diaz Rodriguez, et al. Measuring and modeling the energy cost of reconfiguration in sensor networks. *Sensors Journal, IEEE*, 15(6):3381–3389, 2015.

[26] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *Proc. of the 2nd international conference on Virtual execution environments*, pages 78–88. ACM, 2006.

[27] N. Walravens. Operationalising the concept of the smart city as a local innovation platform: The city of things lab in antwerp, belgium. In *International Conference on Smart Cities*, pages 128–136. Springer, 2016.

[28] T. Watteyne, J. Weiss, L. Doherty, and J. Simon. Industrial ieee802.15.4e networks: Performance and trade-offs. In *Communications (ICC), IEEE International Conference on*, pages 604–609. IEEE, 2015.

[29] Y. Xiao, S. Sethi, H.-H. Chen, and B. Sun. Security services and enhancements in the ieee 802.15.4 wireless sensor networks. In *GLOBECOM'05. IEEE Global Telecommunications Conference, 2005.*, volume 3, pages 5–pp. IEEE, 2005.

[30] F. Yang, N. Matthys, R. Bachiller, S. Michiels, W. Joosen, and D. Hughes. µpnp: plug and play peripherals for the internet of things. In *Proc. of the Tenth European Conference on Computer Systems*, page 25. ACM, 2015.

[31] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *Proc. of the 4th international conference on Embedded networked sensor systems*, pages 139–152. ACM, 2006.