Velox VM: A Safe Execution Environment for Resource-Constrained IoT Applications

Nicolas Tsiftes^{a,*}, Thiemo Voigt^{a,b}

^a*RISE SICS, Box 1263, SE-164 29, Kista, Sweden*

^b*Department of Information Technology, Uppsala University, Box 337, SE-751 05, Uppsala, Sweden*

Abstract

We present Velox, a virtual machine architecture that provides a safe execution environment for applications in resource-constrained IoT devices. Our goal with this architecture is to support developers in writing and deploying safe IoT applications, in a manner similar to smartphones with application stores. To this end, we provide a resource and security policy framework that enables fine-grained control of the execution environments of IoT applications. This framework allows device owners to configure, e.g., the amount of bandwidth, energy, and memory that each IoT application can use. Velox’s features also include support for high-level programming languages, a compact bytecode format, and preemptive multi-threading.

In the context of IoT devices, there are typically severe energy, memory, and processing constraints that make the design and implementation of a virtual machine with such features challenging. We elaborate on how Velox is implemented in a resource-efficient manner, and describe our port of Velox to the Contiki OS. Our experimental evaluation shows that we can control the resource usage of applications with a low overhead. We further show that, for typical I/O-driven IoT applications, the CPU and energy overhead of executing Velox bytecode is as low as 1-5% compared to corresponding applications compiled to machine code. Lastly, we demonstrate how we can use application policies to effectively mitigate the possibility of vulnerable applications to be exploited.

Keywords: Internet of Things, embedded systems, virtual machine, resource management, policy enforcement, high-level programming

*Corresponding Author

Email address: nicolas.tsiftes@ri.se (Nicolas Tsiftes)

© 2015-2017. This manuscript version is made available under the CC-BY-NC-ND 4.0 license <http://creativecommons.org/licenses/by-nc-nd/4.0/>

The final version of this manuscript is published in *Journal of Network and Computer Applications*, Volume 118, pages 61-73, 2018. DOI: <https://doi.org/10.1016/j.jnca.2018.06.001>

1. Introduction

Developing and deploying safe software for resource-constrained devices has been challenging for a long time. The traditional procedure of developing and deploying such software for resource-constrained devices has been to implement it using a low-level language such as C, compile it into a monolithic system firmware of native machine code, and reprogram each device. Instead, we argue that safe software should be developed and distributed in a manner similar to what has become common for smartphones with application stores, enabling software to be deployed on a mass scale.

Realizing this vision is challenging because the execution environment of many types of IoT devices is unsafe. Due to requirements regarding energy-efficiency and low cost, the devices commonly lack hardware protection of memory and peripheral devices. The software is thus given full access to the system, with no barriers between applications or the OS. Security vulnerabilities can enable intruders to control actuators or inject malicious packets into the wireless network. Erroneous software can also cause batteries to drain quickly by keeping various system components active longer than needed, or damage system data. Hence, the execution of native code in an unsafe execution environment precludes installation of externally developed software, and reduces the set of developers that can program such systems.

Since it is imperative to ensure safe execution of applications, one must thus provide a solution in software. For instance, by compiling in run-time checks with native code [16], or by analyzing the software before deployment using model checking or symbolic execution [6, 24]. Previous works typically offer solutions to a subset of challenges that we aim to solve; either by designing runtime environments with high-level language support (e.g., PyMite [17] and Darjeeling [5]), or by providing resource management frameworks (e.g., Pixie [26] and Energy Levels [21]). We elaborate on the differences between our work and the literature in Section 2.

Our main contrast to the literature, however, is that we address a specific combination of challenges to provide a safe and dynamic execution environment for resource-constrained IoT devices. First, supporting remote installation and management of software—possibly from an independent developer—requires a comprehensive framework for application policy enforcement. Second, the execution must be efficient with regard to energy consumption and memory requirements. Third, the software must be compact to facilitate fast and energy-efficient uploading over radio, and reduce the flash storage requirements. This requirement entails that the bytecode format is optimized for IoT applications, but still supports general-purpose programming.

To address these challenges, we present Velox, a virtual machine (VM) for resource-constrained IoT devices. Velox’s salient features are (1) a framework for specifying and enforcing fine-grained resource policies for applications, (2) support both for high-level functional programming (Scheme [20]) and imperative script languages (Cyclus), (3) an execution model in which preemptive multi-threading and exception handling are key elements; (4) remote manage-

Figure 1: An IoT architecture for resource-constrained devices, encompassing the Velox virtual machine, which executes as a process in a host OS such as Contiki.

ment of applications and their policies using CoAP/LWM2M [27]; and (5) a new *domain-specific* bytecode format optimized for resource-constrained IoT software. Our VM is designed for IoT devices of Class 1 and Class 2, which have RAM in the range of 10-50 kB, and ROM in the range of 100-250 kB [4].

The architecture of Velox is depicted in Figure 1, and described in detail in Section 3. Essentially, Velox executes as a process atop an IoT operating system such as Contiki [10]. Within this process, multiple VM applications can be hosted in their own protected environment, each having one or more execution threads. Applications are controlled through a set of individual resource and security policies, which can be configured and updated by the VM administrator.

Although the idea of using application-specific policies is not new [28], we combine the enforcement of both security and resource usage policies into a single framework. Additionally, we are faced with the challenge of accurately monitoring the resource usage of interpreted bytecode instructions by using more coarse-grained profiling data provided by the host OS. We also enable VM users to specify different reactions to policy violations, including forcing the application to use less resources, emitting an exception, or simply logging the event. Hence, Velox can be configured to react differently in testing or deployment environments. In Section 4, we further describe our policy framework.

The contributions of this paper are threefold:

- We present the design and implementation of Velox in Section 5, addressing the aforementioned challenges of providing a safe and energy-efficient runtime environment for resource-constrained IoT devices. In particular, we provide a policy framework that monitors the execution of all bytecode instructions, and enforces application-specific policies.
- We experimentally evaluate Velox’s performance in Section 6 with a set of micro-benchmarks. We show that the bytecode interpretation cost is only

1-5% for a typical data collection application. Additionally, we show that Velox bytecode format is considerably smaller than the common Python and Java bytecode formats, and up to 38% smaller than Darjeeling’s optimized Infusion format [5].

- Through a case study, we show that Velox’s resource policy framework can mitigate the effects of a compromised application used in a DDoS attack, similar to a real attack on a DNS service provider using 500,000 IoT devices that was reported recently [19]. We demonstrate that Velox effectively prevents applications from performing actions or consuming resources in violation to their policy.

2. Related Work

We divide the related work into two categories: resource policies and runtime environments. Both categories contain literature that has had a considerable influence on the research on Velox, but the categories have predominantly been considered separately for the IoT and sensor networks. In the following, we differentiate our work against a selection of the most relevant work in these two categories.

2.1. Resource Management

In the sensor networking field, resource policies have been implemented and evaluated in, for example, the Pixie operating system and the Energy Levels framework. Pixie is based on a resource-aware programming model, in which policies can be specified for applications [26]. Pixie provides a programming abstraction for representing resource availability and resources. Applications become aware of the resources that they consume, and are encouraged to adapt their behavior in a way that is commensurate to the resource availability. Similar to Pixie, Energy Levels is a programming abstraction that makes it possible to specify energy consumption policies for applications, which can adapt by behaving differently depending on how much energy is available to them [21].

Although Energy Levels and Pixie provide effective means for applications to satisfy user-specified resource usage constraints, the application must cooperate to this end. When applications still execute in native mode, and typically in a runtime environment with limited or no privilege separation, erroneous or malicious software can cause security breaches, system malfunction, or increased energy consumption. In a different context, Docker [7] provides resource isolation to applications in the cloud or regular servers. It is different from hypervisor-based systems in that the applications execute on the same operating system, sharing a common layer beneath the applications.

Energy provisioning for mobile phone applications has been provided by systems such as Cinder OS [29], Currentcy [34], and ECOSystem [33]. We have been inspired by such methods when designing Velox, as it can be even more critical to ensure that applications in IoT devices do not surpass their expected energy consumption. Unlike mobile phones, IoT devices typically do not have

hardware that provides privilege separation, and therefore it is more difficult to control applications' consumption of energy without their cooperation. Similar ideas of performing energy accounting have been made for debugging purposes in operating systems for mobile phones, where applications typically execute in a restricted environment or a virtual machine. A different approach is taken by JouleGuard [18], which provides energy guarantees for applications on mobile devices. It manages the trade-off between accuracy and performance to ensure that the applications do not exceed their energy budget.

By contrast, Velox combines the ideas of resource management and security policies in a virtualized execution environment, monitoring a large variety of policy types at the bytecode level. Velox collects the resource statistics provided at the full system level or OS process level, and attributes it to the responsible VM app. Furthermore, Velox offers four different types of reactions that can be configured dynamically for each VM application and policy type, whereas the aforementioned systems typically monitor a single resource with a single type of reaction at the OS process level.

2.2. Run-Time Environments

A plethora of different run-time environments exist in the sensor networking literature. Among the earliest, Maté is an communication-centric virtual machine for sensor networks [22]. Maté later evolved into a VM architecture called Application-Specific Virtual Machines (ASVMs), which reduced some of Maté's limitations, increased support for high-level languages, and enabled instruction set customization [23]. The design of Velox is inspired by the ASVM concept, but differs by having a fixed instruction set designed for efficient representation of a plethora of different IoT applications. This design enables bytecode compatibility across different VM deployments, while providing optimized code representation compared to a low-level, general-purpose instruction set.

Various works have also addressed the problem of providing runtime environments for general-purpose programming languages rather than just domain-specific languages in resource-constrained devices. Darjeeling [5] and TakaTuka [2] implement different subsets of the Java language and its abstract machine specification. Additionally, there are runtime environments for IoT devices that enable high-level languages such as Erlang [31], Lua [1], and Python [3]. Virtual machines have also been developed to support heterogeneous sensor network applications. Servilla provides service provisioning middleware that can host applications in sensor networks, where there are devices with different capabilities and tasks with different complexity [14]. The applications are written in a task programming language, and compiled into platform-independent bytecode that is executed in a virtual machine. A key difference between Velox and its predecessors in the IoT and sensor networking domains is Velox's focus on a full framework for hosting multiple IoT applications with central support for resource protection and provisioning.

Most of the aforementioned systems execute application software compiled to bytecode, but it is also possible to impose safety checks on native code, as the t-kernel does [16]. It provides software-enforced privilege separation between the

OS kernel and applications by rewriting native instructions that access different parts of the memory. Velox extends this concept by allowing users to specify their own security policies, and making it possible to make further restrictions beyond memory accesses.

Another approach to improve IoT application security is to make comprehensive safety checking and testing before deployment. FIE uses symbolic execution to find vulnerabilities in embedded system firmwares [6], whereas SIFT [24] provides a high-level declarative programming framework for IoT applications. SIFT uses model checking and symbolic execution to detect possible policy violations of application logic, and conflicts between applications. Such approaches are orthogonal to that of the Velox runtime policy framework, and the two can be combined to provide strengthened control over application safety.

3. Velox Overview

Fundamentally, Velox is a virtual machine that executes as a regular process in a host OS on IoT devices. It can host multiple IoT applications that are executed using a bytecode interpreter and scheduled preemptively. Velox includes features such as a resource policy framework, a bytecode instruction format optimized for IoT applications, multi-threading, exceptions, and support for high-level programming with an API designed for IP-based applications. Administrators can manage the VM and upgrade software remotely using LWM2M over CoAP, which supports end-to-end DTLS encryption [27].

We have designed Velox to be portable to different IoT operating systems and hardware platforms, with all OS-dependent functionality factored out into a separate component. Currently, it supports Contiki and POSIX-compatible operating systems. Additionally, the IoT applications implemented for Velox are agnostic to the OS environment and hardware architecture that they are being executed in.

3.1. Programming Languages

A key feature of Velox to enhance application security is its support of high-level languages. Such languages eliminate a major class of low-level programming errors that can introduce security vulnerabilities (e.g., buffer overflows and format string attacks). Velox currently supports the Scheme programming language [20], and a new script language named Cyclus. The VM is not limited to these languages, however, and it may thus serve as a research platform for high-level programming abstractions and language features in the IoT domain. Because Scheme is a general-purpose language with a comprehensive set of advanced features, it is possible to support a plethora of languages either through source-to-source compilation, or by direct compilation to Velox’s bytecode language.

3.1.1. Scheme

We select Scheme as the first language to support for two main reasons: 1) it is a high-level programming language, which facilitates safe programming, and 2) the programming constructs in Scheme are simple to map to a high-level instruction set, which has been essential in the design of Velox to enable compact representation of IoT apps. We support most of Scheme R5RS [20], except continuations functionality, which is under development. Velox supports proper tail recursion, as defined by Scheme. When a procedure call is made in a tail context, Velox ensures that the stack frames relating to this procedure call are eliminated. With this feature, iterative programming constructs can be expressed through recursion, but still use constant stack space.

Listing 1: A Scheme implementation of a simple socket app.

```
(guard (obj ((eq? obj 'SocketException)
              (display "Socket failure"))
        (eq? obj 'IOException)
        (close sock)
        (display "I/O failure"))))
(let ((sock (make-client 'UDP "::1" 9800)))
  (let ((loop-func (lambda (i)
                    (when (< i 10)
                      (write "Hello" sock)
                      (thread-sleep! 1000)
                      (loop-func (+ i 1)))))
        (loop-func 0)))
    (close sock))))
```

Through the support for Scheme, Velox enables high-level functional programming, and an advanced macro functionality for creating new programming abstractions in the language itself. This provides an alternative to the predominant C-based imperative languages in use on resource-constrained devices today. Another compelling feature of Scheme is its consistent and compact structure, in which essentially all programming language constructs are expressions consisting of a list of objects, with the head of the list being the function to call. This structure is suitable to compile to Velox's compact bytecode format, with each language construct being mapped to a native function in the VM. Furthermore, the Scheme standard is sufficiently small to make it possible to adhere to it even in implementations for resource-constrained IoT devices.

Listing 1 shows a simple socket app implemented in Scheme. This source code is compiled to 277 bytes of Velox bytecode, including program header, symbols, and instruction codes. A corresponding implementation in C, compiled with msp430-gcc and the size optimization flag set (-Os), yields an ELF module of 1300 bytes. Hence, the Velox bytecode is 78.7% smaller than the ELF module when compiled for the 20-bit MSP430X architecture. A smaller app size is beneficial for reprogramming an IoT network because less energy is needed to distribute the application, and there will be less traffic interfering with the main operations of the network.

Listing 2: A Cyclus implementation of the app in Listing 1.

```
try {
  sock = make_client(*UDP*, ":::1", 9800);
  for(i = 0; i < 10; i++) {
    write("Hello", sock);
    thread_sleep(1000);
  }
  close(sock);
} catch(*SocketException*) {
  println("Socket failure");
} catch(*IOException*) {
  close(sock);
  println("I/O failure");
}
```

3.1.2. Cyclus

To simplify application development, Velox provides a new scripting language called *Cyclus*, which caters to developers that prefer imperative programming languages similar to C and JavaScript. During compilation to Velox bytecode, source code written in Cyclus is translated to Scheme source as an intermediate step. Cyclus supports exception handling, which is of importance to our resource policy framework because one of the possible actions for violations is to notify the application.

Cyclus provides the same API as is available for the VM applications written in Scheme, with some minor differences in naming only. In Listing 2, we show the Cyclus implementation that implements the same functionality as the socket example written in Scheme. All errors are handled by exception handlers, which are mapped to Scheme’s *GUARD* function in the source-to-source translation. The resulting bytecode files for the two examples are close in size: 298 bytes for the Scheme version, and 303 bytes for the Cyclus version. Henceforth, we will show the source code listings in Cyclus.

3.2. Bytecode Design

Velox applications are stored in a new, high-level bytecode format, which provides 193 instructions designed to store resource-constrained IoT applications compactly. This design turns typical operations of such applications, such as opening a socket, sending a packet, and parsing a packet into bytecode instructions. Another key benefit of a domain-specific bytecode format is that it increases execution performance compared to a generic bytecode design that requires multiple low-level instructions to perform the same action, as is the case with the Java bytecode language [25].

This idea is inspired by *application-specific* bytecode formats, which was introduced by the Maté VM in the sensor networking field [23]. While providing great possibilities for bytecode optimization, application-specific bytecode formats require modifications of the VM for each type of application, and therefore reduce the portability. Hence, we instead seek to balance the portability of a constant instruction set with the optimization opportunities offered by tailoring the instruction set for the domain that Velox is designed for.

Figure 2: VM applications are formed by four distinct parts. The program header serves to identify the VM bytecode format, whereas the three following parts are tables that specify strings, symbols, and expressions.

Velox applications are stored in files that consist of a program header, and three different sections, as shown in Figure 2. The program header contains information that allow the VM and external tools to identify VM applications and different bytecode versions. After the header, three different tables are stored: the string table, the symbol table, and the form table. A table stores a set of variable-length items, each prepended by the length of the item.

The string table contains all strings that are used in an app's source code. For example, the expression `(print "Hello")` generates an item in the string table, and this item is then referenced in the byte code by its table index. The symbol table is used similarly when referencing symbols in the source code. The form table contains the byte code itself. Each form represents a Scheme expression in compiled form. Form 0 is special, as it contains the list of expressions on the top level of the application. Once the VM has reached the end of form 0, the application execution is stopped and all resources (i.e., memory, I/O ports) held by a program are automatically released.

4. Resource Monitoring and Control Framework

Velox provides the means for VM administrator to specify fine-grained resource and security policies for each application. The specification can be made immutable at compile-time, but Velox also provides the possibility to make runtime changes of policies. During runtime, Velox monitors the execution of all bytecode instructions, and checks whether those instructions that induce access to external resources are allowed according to the policy specification for the app. We evaluate the overhead cost of this monitoring in Section 6.2.

The *principle of least privilege* entails that applications should be forbidden to access other resources than needed to fulfill their functional requirements [30]. To make it easier for IoT systems to adhere to this principle, Velox provides a strict runtime environment in which apps are isolated from each other and the OS kernel. OSs for PCs can effectively restrict the resources that applications can access by using hardware features such as a memory management unit, I/O port protection, and different privilege levels separating the OS kernel from

applications. On IoT devices, however, this is typically not the case, as the OS and software are usually merged into a single firmware image that has full access to all system resources. Hence, napps executing natively on IoT operating systems such as Contiki, RIOT, and TinyOS, usually have full access to all system resources.

4.1. Policy Specification

The definition of policies is the responsibility of the administrator, but application developers may provide suggested policies along with their application. This simplifies the task of the administrator, who can simply control whether the policy is acceptable, and possibly modify it a bit. Creating a policy from scratch typically involves Velox’s policy tracing functionality, which one can use to make dry runs of the software and get suggested policy rules based on the maximum resource usage in these runs. To ensure that different code paths are tested, the person responsible for specifying the policy may want to vary the input. This problem is similar to that of achieving high coverage in software testing, and improving this procedure is an interesting topic for future work.

Once generated, all policies are stored in a policy file, which at VM compilation time is translated to a C module that is compiled and linked with the VM. The policies can also be updated at runtime. If no policy has been specified for a program that is later loaded into the VM, then that program will get a default policy with highly restrictive policy rules. In Velox, there are three categories of policy rules: power consumption, system resources, and network communication. We describe each one of these below.

Power Consumption. Unlike regular computers, IoT devices have an extremely constrained energy budget since they typically operate on batteries. Hence, one needs to protect the system from energy-based denial-of-service attacks that focus on draining the batteries quickly. Furthermore, unexpectedly high energy consumption can also indicate faulty or fragile software. To combat these problems, Velox enables application-specific energy consumption policies.

By leveraging existing power profilers, Velox is able to account for the power consumption of each its applications despite the challenge of having several bytecode instructions that can induce energy-consuming operations extending beyond the time it takes to execute the instruction. For example, a data packet transmitted with the IP protocol might lead to multiple retransmissions at the link layer, and the cost of these transmissions must be attributed to the responsible application. Velox takes the power profiling data supplied by the host OS, and further divides it among the applications based on Velox’s internal monitoring of application resource usage.

In our Contiki port, the power consumption is monitored with the help of software-based energy profiling [11] in combination with the *Powertrace* tool [9]. The former measures the full system power consumption, whereas the latter traces the power used by each network protocol. Similar frameworks, such as Quanto [15], exist for other operating systems, which highlights that our policy framework is not limited to Contiki. These frameworks keep track of

approximate current consumption by multiplying the time that the system has spent in different states with the current draw in that state.

System Resources. Resources such as files, devices, and sensors can be protected from reading and writing. Policies for device access control enable users to delimit which sensors and actuators can be accessed by an application, and at what time they can do so. This type of policy allows users to download software coupled with policies from sensor cloud services, while being able to trust that the software operates only as intended; e.g., to read temperature periodically and report the results to a specific IPv6 address.

Network Communication. Making it possible to limit network access for IoT applications helps to prevent IoT devices from sending data to false endpoints or participating in DDoS attacks [19]. Administrators can effectively control which hosts, protocols, and ports that VM applications can communicate with. For instance, a data collection application may only be allowed to communicate through UDP on a specific port with a data collection server in the cloud. Hence, there is no reason to allow it to communicate with other hosts, as a vulnerable native application could be exploited to do.

Listing 3: An example of a policy for an IoT app.

```
PROGRAM-POLICY iot-app <SHA256 hash> {
  BANDWIDTH 100 bps
  CPU 5%
  FILE /vm-dev/temp READ
  FILE /sensor-log.txt WRITE
  MEMORY 300
  NET <cloud-service-domain> 80 TCP client
  POWER 500 uW
  RESOURCES console, stats
  THREADS 1 }
```

Listing 3 shows a policy example for a data collection application, which reads sensor samples periodically. The policy restricts the application to use only the resources that are necessary in order to carry out the tasks that it was designed for.

4.2. Policy Enforcement

We rely on runtime monitoring of applications to enforce their accompanied policies. Runtime monitoring comes with the trade-off between using detailed monitoring to support fine-grained policies, and keeping the monitoring overhead low. Furthermore, the monitoring of an application’s resource usage is not limited to when it is executing. Figure 3 shows an overview of the various parts of Velox that interact with the policy enforcement module, and how statistics are gathered both from the host OS and from internal sources.

Configurable reactions. Policies are enforced differently by Velox according to which *action* is bound to the policy rule in question. If no action is specified for a rule, the system uses a default action that is selected based on two factors: 1) the application’s possibility to recover from the rule violation, and 2) Velox’s

Figure 3: The policy enforcer is consulted when executing bytecode instructions that access or consume various types of resources. Each VM application is monitored, and forced by Velox to comply with the policy rules that have been configured for it.

ability to prevent rule violations regarding resource usage. The following four actions exist.

Log. The rule violation is logged to a configurable destination that depends on the platform’s capabilities; e.g., serial output or flash memory. This action is typically used for non-critical errors, or for the case when a user wants to make a dry-run of an application to test a rule set.

Shut down. All of the application’s threads are stopped, and the program is unloaded from Velox, with the action also being logged as described above. This action is typically used for non-recoverable errors.

Slow down. For certain types of resource policy violations, it is possible for Velox to enforce a lower resource consumption from the application without stopping it. Velox simply schedules applications that make resource-consuming function calls with longer intervals until the rule is no longer violated. This action is typically used for the rules regarding bandwidth, CPU utilization, and power consumption. It is more complicated to control the power consumption as accurately as the CPU utilization, because the former can be affected by stochastic factors in the radio environment.

When an application enters slow-down mode because of a power policy violation, Velox records how much the applications estimated power consumption $P_{measured}^t$ surpasses the policy limit P_{limit} , where t is the time when the estimation was made. We denote this difference as $P_{overuse}$. The application is thereafter forced to sleep for a platform-dependent amount of time, X , before obtaining an updated power measurement, $P_{measured}^{t+X}$. At this point, Velox checks whether it is possible to leave the slow-down mode. This is the case if the formula $P_{measured}^{t+X} \leq P_{limit} - P_{overuse}$ holds. If the rule is still violated, the process is repeated.

Throw exception. The thread that executed when the rule violation occurred receives an exception, which can be caught and dealt with. This action is typically used for security and resource policy violations that the application possibly can recover from, such as a rejected file opening attempt, or a rejected heap memory allocation.

5. Design & Implementation

The design of Velox is primarily aimed at enabling fine-grained resource protection and provisioning described in Section 4), and secondarily efficient application execution and memory management. In this section, we expound on a number of design decisions, weighing in factors that are sometimes at odds with one another. Because we target resource-constrained IoT devices, it is desirable to provide functionality that not only supports our main objective of resource monitoring and control for applications but simplifies the programming of such devices. One must also consider that any added feature will impose further complexity on the VM, and thus larger RAM and ROM footprints. In addition to covering the design, we go into more technical detail regarding the implementation when it is relevant, and focus on our port of Velox to Contiki—a widely used open-source OS for IoT devices.

5.1. Execution Model

To ensure that VM applications cannot execute for too long, and thus block other VM applications and system processes from executing, we build our execution model on preemptive threads. This execution model is used irrespective of the host OS thread capabilities, since all context switching is handled within Velox. Although the VM itself operates in a single thread within its host environment, it provides preemptive threading for its own applications. Hence, VM applications can be constructed without forcing developers to be concerned with split-phase state management, as in the TinyOS programming model. Neither do developers need to manually split long-running operations to yield control to a cooperative scheduler, as is sometimes required when developing larger Contiki applications [32].

5.1.1. Preemptive Thread Scheduler

The VM scheduler is responsible for giving each VM thread a share of the execution time that the host OS’s own scheduler gives to the VM. When the host OS is Contiki, the VM runs as a Protothread [12]. The VM protothread is given the opportunity to run as much as possible, and the VM must cooperatively relinquish control of the CPU. We illustrate the main processing chain tracing from the OS scheduler to an individual thread in Figure 4.

As shown in the figure, the main entry point for the VM is the `vm_run` function. This function, which is called whenever the VM protothread runs, invokes the scheduler to execute the next thread that is ready to run. Before scheduling the thread, however, the scheduler turns to the policy framework to check whether the thread is allowed to run. If that is the case, the scheduler executes a finite number of bytecode instructions, before returning to the VM protothread. The `vm_run` function can return two status codes, indicating whether (1) it has more threads ready to run, or (2) all threads are sleeping. A thread is put into sleep mode when it is in slow-down mode because of a policy violation, it is waiting on a timer, or it is executing a blocking I/O operation that cannot be completed immediately.

Figure 4: Overview of Velox’s processing flow, which is repeated at each time Velox is scheduled by the host OS. Velox itself schedules runnable threads in a round-robin fashion, as long as the policy framework allows it.

The VM protothread takes different actions depending on the status code from `vm_run`. If there are more runnable threads, the VM protothread pauses execution, which means that control is yielded to the scheduler, but with a signal that indicates that the VM protothread should be called again immediately after other protothreads have been given their chance to execute. If all threads are sleeping, the VM protothread yields to the Contiki scheduler, which entails that the VM will not execute again until it receives an OS event. Such events are received when timers trigger or when incoming IP packets have arrived. The VM can then determine which thread the event belongs to, and prepare it for execution at the next invocation of the scheduler.

Through this design of the scheduler, the VM does not claim more processing time than what is needed by the VM applications, and it does not claim the processor for too long time, regardless of how the VM applications are implemented. Each VM port has its own scheduler configuration with regard to the number of instructions to schedule per invocation.

5.1.2. High-Level Instructions

The instruction set encompasses 178 instructions, ranging in complexity from basic mathematics functions to higher-order functional constructs such as *map* and *reduce*. About half of the instruction set covers standard Scheme functions, and the other half implements functionality that supports the development of IoT applications. The Scheme functions are in most cases a direct implementation of a corresponding function in the Scheme standard [20], but in some cases there is a difference that requires the Scheme-to-bytecode compiler to translate a Scheme expression to another functionally equivalent Scheme expression that can be executed in the VM.

After performing the initial source-level translation of an expression, the compiler moves on to recursively compile each sub-expression, generating a new *form* for each expression. When an expression E is an argument to a function call, the argument becomes a reference to that expression, $ref(E)$, which represents the expression’s index in the form table.

5.1.3. Instruction Execution

To provide an execution model with preemptive threading, as described above, we need to split long-running operations in VM applications. This ensures that Velox provides a consistent execution model to VM applications regardless of whether the host OS implements preemptive or cooperative thread scheduling.

At VM bytecode level, we consider an instruction to encompass the binary code that describes an expression at the Scheme source code level. Our bytecode format does not include instructions for jumping to certain addresses, returning from function calls, or writing to registers. Such operations are implicit in the higher-level calls, making the format compact and suitable for resource-constrained devices. This is unlike the Java bytecode format, for instance, where low-level operations similar to assembly code are needed. We have seen that this poses problems when used in sensor networks; e.g., requiring the Darjeeling VM to implement a non-standard compressed bytecode format [5].

In Velox bytecode, an instruction consists of a function call, which can be executed by either an internal VM function (e.g., `define`, `add`) or an application-defined function. The call is accompanied by an argument count, and a set of arguments when the argument count is above zero. Instructions are stored in an array abstraction in the VM’s heap, allowing each instruction to be indexed and referred. Since the arguments of an expression can consist of other expressions, such arguments are represented by a succinct reference to the array index of that expression.

During execution of an instruction, Velox evaluates each argument from left to right, except for certain special internal functions that control the order of evaluation themselves (e.g., the `if` function). If the next argument to be evaluated is a reference to an expression, the VM pushes the current expression on the thread’s stack, and switches the evaluation to the expression argument. Once the expression is evaluated, the previous expression is popped from the stack. The result of the expression argument replaces the reference to the expression, and the evaluation moves to the next argument. When all arguments that should be evaluated have also been so, the VM calls the requested function with these arguments.

Velox can break the execution of a thread between the evaluation of arguments, or before continuing to the next expression on the same depth. The only point where it cannot break an operation is within the internal VM function that implements the operation. Hence, we carefully make sure that all internal functions execute for a short and bounded time in the worst case—even when executing a higher-order function on a long list, or opening a socket. This property is attained by having long-running instructions yield control back to the VM scheduler at well-defined points. The instruction execution state is saved on the thread’s stack, and restored when the thread is scheduled again. For instance, when execution the *MAP* instruction with a long list and a mapping function as arguments, Velox will apply the function on one element at a time between yield points.

5.2. Error Handling with Exceptions

Velox provides a framework for exception handling, making it possible not only to gain feedback from the resource policy framework but also to debug software with greater visibility and robustness than what is available for applications written in low-level languages. Such languages put the responsibility of correctly checking for all errors on the programmer. Once an error occurs in native applications, it may also not be possible to recover from the error, and the whole system may crash, making it difficult to perform debugging afterwards.

When the VM executes a bytecode instruction on the behalf of an application, a number of things may cause an error; e.g., the application’s designated resources have been used up, a security policy has been violated, or an illicit argument has been supplied to an instruction. Such errors most often result in an exception being raised, and the application can catch the exception and take an action deemed appropriate by the developer. Once an exception is caught, the application can easily determine the faulty expression in the application. In cases where the exception is not caught, or there is no possibility for the application to continue execution (e.g., when it has spent its energy budget), Velox terminates all of the application’s threads and makes a log entry of the event in a port-specific manner.

5.3. Memory Management

Velox relies extensively on dynamically allocated memory with garbage collection. Therefore, the memory management must be designed to have low processing and space overhead. Each application has a designated stack, and can make allocations on the shared VM heap. Velox provides memory protection by restricting applications from accessing memory directly, and instead only allowing them to read and write VM objects through a well-defined, safe API.

Dynamic memory allocations are made through two different modules: the heap allocator and the object pool. The former is used for larger allocations that are typically of varying sizes, such as a large vector allocated in a VM application. The latter is used for small allocations that are typically of uniform sizes, such as VM objects. In the context of Velox, we consider larger allocations to be those that exceed the size of list items, which is 12 bytes.

5.3.1. Heap Allocator

Different implementations of a heap allocator can be used depending on which platform Velox is compiled for. Currently, it can either be the standard C *malloc* library provided by the host environment, or the *HeapMem* library that we have implemented as an alternative for environments that do not provide a malloc library. The importance of providing this alternative is highlighted by the fact that widely used IoT operating systems such as Contiki and TinyOS do not provide such a library.

5.3.2. Object Pool

The object pool is an allocator for small objects that can operate with considerably less memory and processing overhead than the regular heap allocator. The implementation is simple and improves the performance compared to using the heap allocator alone. An object pool instance is stored in a data structure containing an array of elements of uniform size, and a bitmap indicating which elements are occupied. Object pools can be created and destroyed dynamically since they are allocated as a large contiguous block through the heap allocator.

5.3.3. Garbage Collection

Applications that make heap allocations (e.g., by creating a list) need not be concerned with deallocating such memory manually. Instead, Velox provides a garbage collector (GC) that employs the mark-and-sweep algorithm. By performing garbage collection on heap memory, it simplifies the task of developing robust programs free of memory leaks and fragmentation-inducing allocation patterns. Since this algorithm can execute for several milliseconds on resource-constrained IoT devices, applications with realtime constraints are not suitable to run within Velox.

The garbage collector is invoked in two cases: (1) once a heap allocation fails because there is no further memory left, and (2), when a certain configurable amount of memory has been allocated since the last invocation of the garbage collector. Case 2 is useful for platforms in which Velox uses the standard C memory allocator so that the memory footprint does not become overly large. This is not necessary on an IoT operating system such as Contiki, where the VM uses a different allocator such as HeapMem, which has a predetermined memory area for its use. Hence, only case 1 is needed for the GC to operate in the primary target platforms of Velox.

GC Algorithm. Whenever the VM allocates memory on the heap, the allocation is inserted into a hash table. Its hash keys correspond to the allocated memory addresses, and the hash values are used only during the mark phase, where they specify whether the memory is currently referenced by a VM thread. Memory that has been allocated on the object pool is not accounted for in the heap because of the relative memory overhead for the small objects. These objects are instead marked by the garbage collector by setting a bit in a bitmap. Velox does not use a similar bitmap for regular heap allocations because it assumes that it has a black-box view of the heap allocator, which is typically provided by the host OS.

For each thread, the GC goes through three distinct areas to find memory references: 1) objects currently residing on the execution stack of the thread, 2) lexically bound objects (e.g., through the *LET* function), and 3) dynamically bound objects (e.g., through the *DEFINE* function).

After completing the marking phase, the garbage collector sweeps over the active allocation set, and deallocates any object that is unmarked by removing it from the hash table and calling the deallocation function of the heap back-end.

5.4. OS Adaptation Layer

To make it possible for IoT applications to use OS-specific functionality Velox provides an OS adaptation layer. All ports of the VM are required to implement a core set of functions specific to the VM. In addition, this layer ensures that the standard I/O functions in Scheme behave identically across different ports. For example, the Contiki port of the VM provides access to sensors, actuators, and LEDs.

Because VM threads are preemptive, this layer is also responsible for making sure that external calls are not blocking, typically by using a generic poll mechanism provided by the OS; e.g., the `poll()` function in POSIX. In some cases where this is not available; e.g., performing device driver I/O in Contiki, the VM must itself poll such devices directly, possibly by using non-blocking function calls. When a thread makes a request that cannot be satisfied immediately, the VM puts the thread in waiting mode until the call is ready to be made without blocking.

The Contiki port uses two OS subsystems extensively: the Coffee file system is not only used to store and load VM applications but also to provide the capability for VM applications to access the local file system on IoT devices. The uIP networking stack is used to implement the IPv6 API provided to VM applications. Furthermore, the port uses the Sensors and LED APIs in Contiki to implement virtual VM sensor devices that provide the same services.

6. Evaluation

We evaluate the benefits and costs of using Velox to provide security for IoT applications. The evaluation is divided in two parts that provide different perspectives on the characteristics of Velox. In the first part, we conduct a series of micro benchmarks to measure the performance of different types of applications. We also measure the ROM and RAM footprints of the VM implementation. In the second part, we carry out two case studies. The first case study evaluates the performance of a realistic IoT application, comparing two functionally equivalent implementations: one executing in native mode, and one executing in the VM. The second case study evaluates Velox’s ability to mitigate a DDoS attack using different application policies.

6.1. Experimental Setup

We experimentally evaluate Velox using two different resource-constrained IoT devices that run the Contiki operating system: Zolertia RE-Mote and Arago Systems WisMote. Our experiments are conducted both on real devices and in a simulator capable of emulating real nodes with cycle-accurate timing. We use Contiki’s low-power IPv6 stack to evaluate the networking functionality of Velox, including the use of protocols such as 6LoWPAN, IPv6, RPL, and UDP. Beneath the networking stack, we run ContikiMAC [8], an asynchronous, sender-initiated duty-cycling protocol, configured with the default channel check rate of 8 Hz.

6.1.1. Real Devices

In the real-world experiments, we use Zolertia RE-Motes, which are equipped with a Texas Instruments CC2538 microcontroller. These devices are based on the 32-bit ARM Cortex-M3 architecture, and are equipped with up to 32 kB RAM and 512 kB ROM. The CC2538 also has an integrated 2.4 GHz, IEEE 802.15.4 radio and a variety of peripherals.

We choose this platform because its software support in Contiki is under active maintenance, and because it has a relatively high memory capacity. In order to enable the full 32 kB of RAM, however, we need to restrict the device to limit itself to Low Power Mode (LPM) 1 when sleeping. This incurs a higher sleep current compared to Contiki’s default use of LPM 2, and this cost is accounted for in our power measurements.

6.1.2. Emulated Devices

We further evaluate Velox use the cross-layer network simulator and hardware emulator *COOJA/MSPsim* [13]. This method of evaluation allows us to gain a level of insight that would not be possible using the real devices alone. COOJA enables simulations with networks of nodes, each running an instance of the MSPsim emulator. MSPsim executes the same firmware image as would run on a real node, which makes it suitable to get detailed measurements of the VM performance.

The IoT device that we emulate in these experiments is Arago Systems’ *WisMote*, which is based on Texas Instruments’ MSP430 architecture. We select this platform because, with its 16 kB of RAM and 256 kB of ROM, it is representative of the more resource-constrained devices. The WisMote is equipped with a CC2520 2.4 GHz, IEEE 802.15.4-compatible radio. In contrast to the Zolertia RE-Mote, it has ample emulation support in MSPsim, with most of its peripheral chips being emulated.

6.2. Micro Benchmark

Listing 4: Memory test application used in the benchmark.

```
list_size = 100;
iterations = 1000;
X = list();

for(i = 0; i < list_size; i++)
{
    push(i, X);
}

for(i = 0; i < iterations; i++) {
    X = reverse(X);
}
```

In this experiment, we measure the execution time profiles of three applications with vastly different workloads: A) a memory-intensive application, B)

Figure 5: Execution time profiles of three benchmark apps, which shows how Velox divides its time between different tasks depending on the type of workload. The total overhead for memory management, policy control, and thread scheduling constitutes a minority of the execution time.

a processing-intensive application, and C) an I/O-intensive application. Application A, shown in Listing 4, is implemented in Cyclus. It stress-tests the garbage collector by repeatedly creating a new list with the *reverse* function, and dropping the reference to the memory of the original list that was supplied as an argument. Application B is an infinite loop of mathematical calculations. Lastly, Application C is an IP networking application that performs relatively costly packet transmissions at 10 s intervals.

Figure 5 shows the resulting execution profiles, divided between different categories of Velox operations. We show how Application A is spending more time on object pool management and garbage collection than the other applications. Application B spends a large part of its time on instruction fetching and execution, but scheduling is also a considerable cost since Velox must preempt its execution periodically. Application C, which spends most of its active runtime executing high-level socket instructions, has a low time share in the other categories. The cost of policy control did not exceed 2.3% in any of our experiments.

Hence, we have shown that even when there is a large number of context switches from VM application to the host OS—and thus a large corresponding number of resource policy checks—the overhead of policy control is a minor part in the execution profiles of all the tested applications.

6.3. Implementation Complexity

The implementation of Velox consists of the VM core, an instruction set, and platform-specific functionality. We have measured the static memory footprint, the ROM footprint and the source lines of code as a gauge of implementation complexity. The ROM footprint is measured on a Velox firmware compiled for the CC2538 microcontroller. We use the *arm-none-eabi-size* tool for ARM

Table 1: Implementation complexity.

Component	Data (bytes)	ROM (bytes)	SLOC
VM core	14928	25201	4478
VM instruction set	16	13594	3344
VM-to-OS port	902	4973	2094
<i>Sum</i>	15846	43768	9916

Embedded Processors to measure the ELF sections of Velox’s modules, which can be divided into three different categories: the core VM functionality that is platform-independent, the instruction set, and the VM-to-OS port, which in this case entails Contiki-related functionality. Additionally, the source lines of code (SLOC) values of these modules are generated using David A. Wheeler’s *SLOCCount* tool. For comparison, our configuration of Contiki compiles to a firmware that uses 11455 bytes of RAM and 53287 bytes of ROM.

Table 1 shows that the VM in its entirety requires less than 50 kB of ROM space, entailing that it fits with a healthy margin on modern IoT devices, which typically have at least 128 kB of ROM, including MSP430-based nodes such as the WisMote; and ARM Cortex M3-based nodes such as the RE-Mote. The static memory footprint is approximately 16 kB with the default configuration for Contiki. The largest share of the footprint is attributed to the heap memory area, which is managed by the VM core component. The data footprint is configurable, however, and can be trimmed down for even more resource-constrained devices than those that we primarily target, but this comes at the cost of restricting the ability of the hosted IoT applications.

6.4. Bytecode Efficiency

To evaluate the efficiency of Velox’s high-level bytecode instructions, we measure the information entropy and size of different applications in bytecode format. The virtual machines that we consider in this study are 1) PyMite [17] (Python bytecode compiled with Python 3.5.2), 2) TakaTuka [2] (standard Java bytecode compiled with JDK 1.8.0_121, without debug information), and 3) Darjeeling [5] (compressed Java bytecode, or *infusions*). These VMs have been designed for resource-constrained devices and support high-level programming languages, which make them suitable for comparison with Velox.

We examine three applications that have been implemented in a functionally equivalent manner for the different bytecode interpreters listed above: a serial I/O application (*I/O App*), a UDP-based networking application (*Networking App*), and a prime number sieve application (*Math App*). Darjeeling lacks an API for programming UDP sockets, so we use the simple radio API provided by Darjeeling, which might contribute to slightly smaller code.

Table 2 shows that Velox bytecode format is more compact than the others, including Darjeeling infusion, which has been optimized for resource-constrained devices. The table shows that the the size increase compared to the Velox format

Table 2: Bytecode size comparison.

Format	<i>I/O App</i>			<i>Networking App</i>			<i>Math App</i>		
	Bytes	Entropy	Inc.	Bytes	Entropy	Inc.	Bytes	Entropy	Inc.
Java	628	5.23	451%	1228	5.54	238%	2286	5.62	146%
Python	260	3.96	128%	735	5.00	102%	2031	4.54	118%
Infusion	185	4.49	62%	404	5.12	11%	945	5.61	2%
Velox	114	4.74	–	363	5.74	–	930	5.90	–

ranges from 2% to 451% for our sample applications. We see that standard Java class files are considerably larger than the other formats, and that Python bytecode is closer in size to the optimized formats of Darjeeling infusions and Velox than Java classes. For the Math App, the Infusion and Velox bytecodes are of comparable size. The main reason for this is that Velox is not designed to optimize the bytecode size of calculation-intensive applications as well as that of I/O- and event-driven networking applications. Still, the Velox is approximately 1.6% smaller in this case.

Furthermore, the information entropies offer a hint of how much redundancy the files of different formats contain, where higher entropies imply a lower potential for compression. Java class files contain additional information such as a constant pool, which make their entropy not directly comparable to the other formats. Because Velox bytecode format is already designed to be compact, it cannot be compressed much more even by a state-of-the-art compression tool.

6.5. Power Policy Enforcement

In this experiment, we examine Velox’s ability to limit the power consumption when an application has a highly varying power consumption pattern. The power consumption is the most challenging to manage, since it relies both on accurate, yet low-cost measurements that have to be traced to the VM application, and because the power is affected by more stochastic factors than other resources.

We use a UDP-based data collection application (described in more detail in Section 6.6) because it switches between two extreme power consumption states: low-power sleeping and IPv6 packet transmissions over a duty-cycled radio. If the application would be allowed to execute freely, its actual power consumption is on some nodes higher than 500 uW, which is the highest policy limit tested in the experiment. The application operates on a multi-hop network with 30 client nodes, which send their packets to a sink.

Figure 6 shows the number of packets received by the sink from all nodes, as we vary the power consumption policy. The number of packets is reduced when less power is available to the nodes because Velox forces the application to sleep if it consumes too much power. In all experimental runs, the packet reception rate exceeded 99.7%, which implies that the policy enforcement does not diminish the application’s ability to successfully transmit data once it is allowed to do so. Furthermore, the forwarding is not affected by the resource

Figure 6: Mean number of packets received by the sink from all nodes when using different power policies. With a lower power policy, the application is forced by Velox to send fewer packets. When the power limit is higher than the application needs, the packet count is stable because the rate is controlled by the application alone.

policy in place because Velox does not interfere with the IP stack’s internal operation.

Figure 7 shows the mean, minimum, and maximum power consumption of all nodes with different power consumption policies. The nodes with the lowest power consumption are 1-hop neighbors to the sink, which is not duty-cycled. The lower transmission cost is a result of the sink usually responding on the first ContikiMAC strobe. For all nodes, Velox successfully enforces a maximum power consumption below the policy limit, despite the application’s regular attempts to use more. Hence, when giving a higher policy limit, the VM increases the application’s power consumption accordingly to give it as much resources as can be afforded.

6.6. Case Study: Data Collection

In this experiment, we study the cost of executing an IoT application in a virtual machine instead of in native mode. We focus on the type of workloads that Velox is mainly designed for: I/O-driven or event-driven IoT applications. Therefore, we implement two versions of an IPv6-based data collection application that are functionally equivalent. They both send a data packet with a sensor sample over UDP every 10 seconds on average, with a random jitter to each period length to decrease collision risk. We run each experiment for 1 hour in a simulated multi-hop network consisting of 30 clients and one server running Contiki.

One version of the data collection application is written in Cyclus and compiled to VM bytecode, whereas the other version is written in C and compiled to run as a Contiki process in native mode. We show the entire implementation of the Cyclus version in Listing 5. We have omitted the C version for brevity since it has approximately three times as many lines of code. The difference in source

Figure 7: Mean, minimum, and maximum power consumption of all nodes in the data collection network. The VM ensures that the application stays within its policy limit. Velox effectively controls the power consumption, despite the application’s many attempts to use more.

Figure 8: Mean and standard deviation of CPU and energy usage in relation to the per-node packet intervals in a data collection network. The overhead of the VM application compared to the native application diminishes as the packet interval increases.

code sizes also manifests into the compiled program sizes. In Velox’s bytecode format, the application size is 372 bytes, and 343 bytes when compressed with GZIP. The native application is stored as an ELF module, which has a size of 1244 bytes. Hence, the VM application is 70% smaller than the native version.

CPU usage. Figure 8a shows the mean CPU usage among network nodes when varying the packet transmission interval. As expected, the CPU usage overhead of Velox is highest when the packet transmission interval is the lowest. The native application then used on average 4.73% CPU time, whereas the Velox application used on average 5.58%—an overhead of 18.0%. With the longest transmission interval, 100 s, the CPU usage was 1.853% for the VM application, and 1.87% for the native application—an overhead of 1.9%.

Energy consumption. We also find that the energy consumption increases with shorter transmission intervals, but by a different factor because the CPU

Listing 5: Cyclus source code for the data collection application.

```

import("rpl"); /* Load Velox's RPL library. */

/* Configuration. */
send_interval = 10000;
remote_addr = "fdff::200:0:0:1";
remote_port = 5678;

/* Wait for the routing to initialize. */
while(!rpl_connected?()) {
    thread_sleep!(1000);
}

/* Create a UDP socket. */
sock = make_client(*UDP*, remote_addr, remote_port);

/* Periodically send a message with an average
   interval specified by "send_interval" in ms. */
for(count = 1;; count++) {
    msg = string_append("Message number", number->string(count));
    println("Sending ", msg, "");
    write(msg, sock);
    thread_sleep!((send_interval / 2) + modulo(random(), send_interval));
}

```

usage contributes to a relatively small part of the total energy. The radio will cause a more considerable energy expenditure, partly because it consumes a current that is approximately ten times higher than that of the CPU, and partly because it is duty-cycled, which entails that it has to listen for traffic while being idle.

Table 3 shows the incurred CPU usage, and the transmission (TX) and reception (RX) duty cycles for the VM app and the native app when configured with a 100 s data transmission interval. The *incurred* duty cycles are the fraction of time that the radio spends in each mode, and is affected not only by the *configured* idle duty cycle but also by interference and network traffic. In this experiment, we do not introduce external interference since this would add a stochastic factor that can distort the comparison. With respect to the RX duty cycle, we see matching numbers because the VM should not affect the timing of the reception of packets at the MAC layer. When transmitting, however, the TX duty cycle, which has a considerably smaller effect on the total energy cost, is increased from 0.092% to 0.106%. We attribute this increase to the slightly slower communication between the VM and the MAC layer than what a native application could achieve.

When considering the total energy consumption over the network's lifetime, the TX duty cycle has a considerably smaller share than the RX duty cycle. Figure 8b shows the mean energy consumption of the nodes. We test the application with different packet transmission intervals to see how much the overhead changes when the processing becomes more intense. With a 100 s and 10 s packet intervals, the VM's energy overhead is 2-3%. When sending at 1 s intervals, however, the average energy overhead is 27.1%. Although this overhead is an order of magnitude larger, such short packet intervals are uncommon in

Table 3: Comparison of CPU usage and radio duty cycle for the two functionally identical implementations of a data collection app, expressed in percentages of the total time.

Implementation	CPU usage (%)	TX duty cycle (%)	RX duty cycle (%)
Velox impl.	1.87	0.106	0.65
Native impl.	1.853	0.092	0.65
<i>Relative overhead</i>	0.92	15.2	0

duty-cycled data collection applications. For typical data collection workloads, the use of Velox provides the benefit of a secure run-time environment at low cost.

6.7. Case Study: DDoS Mitigation

Our final experiment is designed to evaluate Velox’s efficacy in mitigating a DDoS attack that employs IoT devices. As mentioned earlier, such an attack has already been carried out in reality on a DNS service provider [19]. To simulate a compromised device’s participation in such an attack, we implement an application that intentionally generates DNS look-up requests at the highest possible rate attainable with the device’s capabilities. This DNS app implemented in Cyclus, and uses Velox’s DNS resolve procedure to look up randomly generated host names. In compiled format, the application requires 478 bytes of storage, including code for full error checking and statistics printing.

We deploy Velox and our DNS app on top of Contiki on a Zolertia RE-Mote device. We connect this device with IPv6 over 6LoWPAN to a Contiki border router, which in turn is connected to a workstation where we place the target DNS server.

We measure the outgoing DNS request rate of the app under three different types of policy configurations: 1) no policy, 2) a variety of power consumption policies and 3) a specific policy aimed at preventing DNS requests. We also measure the jitter in the DNS request intervals to gain insight as to how the policies affect the execution of the application. For each policy configuration, we run the experiment 10 times, and extract the mean request rate and the standard deviation.

In Figure 9, we find in the leftmost column that, as expected, the application is unable to send any DNS request at all when it is forbidden to use the DNS service. The three columns in the middle show the results when the application is put under different power consumption policies, with a linear relationship between the allowed power consumption and the number of DNS requests sent. For reference, we also show in the rightmost column the attained request rate when the application is operating under no policy. Hence, each application’s policy should be set so that the application is only allowed to do what it is designed for, thereby reducing the possibility to exploit an application.

Figure 9: Outgoing DNS request rate for a VM application under different VM policies. Velox reduces the request rate as the application is put under stricter policies, effectively preventing attackers from using the application for DDoS attacks.

7. Conclusions

In this paper, we have presented Velox, a virtual machine architecture with the objective to provide a safe, yet resource-efficient execution environment for IoT applications. One of Velox’s main features is a fine-grained resource and security policy framework, through which one can ensure that the applications do not access any system resources beyond what has been provisioned for them. We have also demonstrated Velox’s support for high-level programming languages that can be compiled to a compact bytecode format. Through our experimental evaluation, we have shown that Velox effectively enforces the policies upon IoT applications with low computational and memory overhead. Furthermore, we have compared the cost of virtualization with the execution of corresponding apps written in C, and shown that the energy and CPU overhead is in the range of 1-5% for I/O-driven IoT applications. Hence, we find that Velox provides a safe execution environment—while providing low overhead—for a large class of IoT applications.

Acknowledgments

This work was financed by VINNOVA, the Swedish Agency for Innovation Systems; and by the distributed environment E-care@Home, funded by the Swedish Knowledge Foundation.

- [1] Andersen, M., Fierro, G., Culler, D., 2017. Enabling Synergy in IoT: Platform to Service and Beyond. *Journal of Network and Computer Applications* 81, 96–110.
- [2] Aslam, F., Fennell, L., Schindelhauer, C., Thiemann, P., Ernst, G., Haussmann, E., Rührup, S., Uzmi, Z., 2010. Optimized java binary and virtual

- machine for tiny motes. In: Proceedings of Distributed Computing in Sensor Systems (DCOSS). Santa Barbara, CA, USA.
- [3] Bocchino, S., Fedor, S., Petracca, M., Feb. 2015. PyFUNS: A Python Framework for Ubiquitous Networked Sensors. In: Proceedings of the European Conference on Wireless Sensor Networks (EWSN). Porto, Portugal.
 - [4] Bormann, C., Ersue, M., Keranen, A., May 2014. RFC 7228: Terminology for Constrained-Node Networks.
 - [5] Brouwers, N., Langendoen, K., Corke, P., 2009. Darjeeling, a feature-rich vm for the resource poor. In: Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys). Berkeley, CA, USA.
 - [6] Davidson, D., Moench, B., Ristenpart, T., Jha, S., 2013. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In: Proceedings of the 22nd USENIX Security Symposium. Washington, D.C., USA.
 - [7] Docker, Inc., 2016. Docker. Web page, visited 2017-02-07. URL <http://www.docker.com/>
 - [8] Dunkels, A., Dec. 2011. The ContikiMAC Radio Duty Cycling Protocol. Tech. Rep. T2011:13, Swedish Institute of Computer Science.
 - [9] Dunkels, A., Eriksson, J., Finne, N., Tsiftes, N., Mar. 2011. Powertrace: Network-level power profiling for low-power wireless networks. Tech. Rep. T2011:05, Swedish Institute of Computer Science.
 - [10] Dunkels, A., Grönvall, B., Voigt, T., Nov. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the IEEE Workshop on Embedded Networked Sensor Systems (IEEE Emnets). Tampa, Florida, USA.
 - [11] Dunkels, A., Österlind, F., Tsiftes, N., He, Z., Jun. 2007. Software-based on-line energy estimation for sensor nodes. In: Proceedings of the IEEE Workshop on Embedded Networked Sensor Systems (IEEE Emnets). Cork, Ireland.
 - [12] Dunkels, A., Schmidt, O., Voigt, T., Ali, M., Nov. 2006. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In: Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys). Boulder, Colorado, USA.
 - [13] Eriksson, J., Österlind, F., Finne, N., Tsiftes, N., Dunkels, A., Voigt, T., Sauter, R., Marrón, P. J., Mar. 2009. Cooja/mspsim: Interoperability testing for wireless sensor networks. In: SIMUTools 2009. Rome, Italy.

- [14] Fok, C., Roman, G., Lu, C., Jun. 2012. Servilla: a flexible service provisioning middleware for heterogeneous sensor networks. *Science of Computer Programming, Special Issue on Best Papers of Coordination'09* 77, 663–684.
- [15] Fonseca, R., Dutta, P., Levis, P., Stoica, I., 2008. Quanto: Tracking energy in networked embedded systems. In: *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA, USA.
- [16] Gu, L., Stankovic, J., Nov. 2006. t-kernel: providing reliable OS support to wireless sensor networks. In: *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*. Boulder, CO, USA.
- [17] Hall, D., 2011. Python-on-a-chip. Web page, visited 2017-02-07.
URL <https://code.google.com/archive/p/python-on-a-chip/>
- [18] Hoffmann, H., 2015. JouleGuard: Energy Guarantees for Approximate Applications. In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, USA.
- [19] Kan, M., 2016. An IoT botnet was partly behind Friday's massive DDOS attack. *ComputerWorld*, Nov 8, 2016.
URL <http://www.computerworld.com/article/3134062/>
- [20] Kelsey, R., Clinger, W., J. Rees (Editors), et al., 1998. Fifth Revised Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices* 33 (9), 26–76.
- [21] Lachenmann, A., Marrón, P., Minder, D., Rothermel, K., 2007. Meeting lifetime goals with energy levels. In: *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*. Sydney, Australia.
- [22] Levis, P., Culler, D., Oct. 2002. Maté: A tiny virtual machine for sensor networks. In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, CA, USA.
- [23] Levis, P., Gay, D., Culler, D., May 2005. Active sensor networks. In: *Proceedings of the USENIX Symposium on Networked Systems Design & Implementation (NSDI)*. Boston, MA, USA.
- [24] Liang, C.-J. M., Karlsson, B. F., Lane, N. D., Zhao, F., Zhang, J., Pan, Z., Li, Z., Yu, Y., 2015. SIFT: Building an Internet of Safe Things. In: *Proceedings of the International Conference on Information Processing in Sensor Networks (ACM/IEEE IPSN)*. Seattle, WA, USA.

- [25] Lindholm, T., Yellin, F., Bracha, G., Buckley, A., 2013. The Java Virtual Machine Specification, Java SE 7 Edition.
URL <https://docs.oracle.com/javase/specs/jvms/se7/html/>
- [26] Lorincz, K., Chen, B., Waterman, J., Werner-Allen, G., Welsh, M., 2008. Resource aware programming in the Pixie OS. In: Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys). Raleigh, NC, USA, pp. 211–224.
- [27] Open Mobile Alliance, 2016. LWM2M Specification 1.0.
URL <http://technical.openmobilealliance.org/Technical/technicalinformation/release-program/current-releases>
- [28] Provos, N., 2003. Improving host security with system call policies. In: Proceedings of the 12th USENIX Security Symposium. Washington DC, USA.
- [29] Roy, A., Rumble, S., Stutsman, R., Levis, P., Mazières, D., Zeldovich, N., 2011. Energy Management in Mobile Devices with the Cinder Operating System. In: Proceedings of the ACM European Conference on Computer Systems (ACM EuroSys). Salzburg, Austria.
- [30] Saltzer, J. H., Schroeder, M. D., 1975. The protection of information in computer systems. Proceedings of the IEEE 63 (9), 1278–1308.
- [31] Sivieri, A., Mottola, L., Cugola, G., 2012. Drop the Phone and Talk to the Physical World: Programming the Internet of Things with Erlang. In: International Workshop on Software Engineering for Sensor Network Applications (Sesena). Zürich, Switzerland.
- [32] Tsiftes, N., Dunkels, A., Nov. 2011. A database in every sensor. In: Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys). Seattle, WA, USA.
- [33] Zeng, H., Ellis, C., Lebeck, A., Vahdat, A., 2002. ECOSystem: Managing Energy as a First Class Operating System Resource. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). San Jose, CA, USA.
- [34] Zeng, H., Ellis, C., Lebeck, A., Vahdat, A., 2003. Currentcy: A Unifying Abstraction for Expressing Energy Management Policies. In: Proceedings of the USENIX Annual Technical Conference. San Antonio, TX, USA.