π PICCOLO

Deliverable D2.3

Initial Report on Security and Isolation Improvements

WP Leader:	Chris Adeniyi-Jones – Arm Ltd
Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public
Contractual delivery date:	March 31, 2022
Actual delivery date:	March 31, 2022
Suggested readers:	Researchers, developers and technologists interested in edge computing and the convergence of networking and computing.
Version:	1.0
Total number of pages:	30
Keywords:	In-network computing, node

Abstract

This document describes several approaches explored for isolation and security improvements, in the context of edge computing and in-network computing.

Disclaimer

This work was done within the EU CELTIC-NEXT project PICCOLO (Contract No. C2019/2-2). The project is supported in part by the German Federal Ministry of Economic Affairs and Energy (BMWi) and managed by the project agency of the German Aerospace Center (DLR) (under Contract No. 01MT20005A). The project also receives funding awarded by UK Research and Innovation through the Industrial Strategy Challenge Fund. The project is also funded by each Partner.

Impressum

[Full project title] Piccolo: In-network compute for 5G services
[Short project title] Piccolo
[Number and title of work-package] WP2. Piccolo Node
[Number and title of task] T2.3 Isolation and Security
[Document title] D2.3 Initial Report on Security and Isolation Improvements
[Editor] Chris Adeniyi-Jones, Arm
[Work-package leader] Chris Adeniyi-Jones, Arm

Copyright notice

Executive Summary

The Piccolo project defines a distributed computing architecture that features flexible composition and dynamic, constraint-aware function instantiation. In this system, multiple workloads, each comprised of individual function residing on different Node platforms share a common infrastructure, with the general assumption that neither the underlying Node platforms nor the network has to be trusted.

Thus, one key Piccolo feature is the isolation of Piccolo workloads (distributed applications) in sharing such a common infrastructure. *Isolation* refers to security and performance isolation and is a system-wide as well as a Node-local property.

Piccolo has a Node architecture that specifies functions/services of node platforms as well as their APIs to guest functions and the rest of the Piccolo infrastructure. Piccolo systems can leverage different Node implementations, for example featuring different virtualisation and containerisation technologies. This document focuses on the per-Node isolation aspects, however it is also touching upon the system-wide isolation vision for context.

We describe how functions interact with the Node Agent service on each Piccolo Node variant and how performance and security isolation is respectively achieved. We also describe open issues and plans for further research.

List of Authors

Organisation	**Author**
Arm Ltd.	Chris Adeniyi-Jones
British Telecommunications plc	Philip Eardley, Andy Reid
Fluentic Networks Ltd.	Ioannis Psaras
InnoRoute GmbH	Andreas Foglar, Marian Ulbricht
Robert Bosch GmbH	Dennis Grewe, Naresh Nayak
Sensing Feeling	Dan Browning, Jag Minhas, Chris Stevens
Stritzinger GmbH	Mirjam Friesen, Sascha Kattelmann, Peer Stritzinger, Stefan Timm
Technical University Munich	Raphael Hetzel, Nitinder Mohan, Jörg Ott
University of Applied Science Emden/Leer	Dirk Kutscher, Laura al Wardani, T M Rayhan Gias

Table of Contents

List of Figures

Abbreviations

API	Application Programming Interface
CAN	Controller Area Network
CPU	Central Processing Unit
DNN	Deep Neural Network
EE	Execution Environment
ICMP	Internet Control Message Protocol
ICN	Information-Centric Networking
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission (standards body)
I/O	Input / Output
IoT	Internet Of Things
ISA	Instruction Set Architecture
KVM	Kernel-based Virtual Machine
MQTT	(originally Message Queueing Telemetry Transport)
NDN	Named-Data Networking
NFD	NDN Forwarding Daemon
OBD	On-board Diagnostics
OS	Operating System
PKI	Public Key Infrastructure
PLC	Programmable Logic Controller
PoC	Proof of Concept (demonstrator)
QEMU	Quick EMUlator (software)
RPi4	RaspberryPi4
SaaS	Software as a Service
SGX	Software Guard Extensions (Intel)
SoC	System on a Chip
TCB	Trusted Computing Base
UDP	User Datagram Protocol
VM	Virtual Machine
VPE	Visual Processing Engine

YAML	YAML Ain’t Markup Language

1 Introduction

This document describes the work that has been done within the project looking at approaches for improving the isolation and security aspects of a Piccolo Node and a Piccolo System consisting of Piccolo Nodes that are connected and can communicate with each other.

The vision of the Piccolo Project is that "compute" will become integrated into the network and storage fabric: Every network node will provide secure processing and storage for third party applications and network functions. In this context, from a security and isolation perspective, the key challenge is to enable multi-tenancy: for multiple users, the ability to support computation distributed across the nodes, alongside guarantees that one tenant cannot interfere with the operation of another tenant. (Earlier deliverables discuss this, amongst other design goals and requirements for a Piccolo system [1] [2] [3].)

Multi-tenancy breaks down into several requirements which can be categorised as follows:

- Access control - authentication of users and their applications requesting access to Piccolo resources;
- Data security - ensuring that users and applications can only access and update their own data;
- Anonymity - ensuring that users/applications cannot identify other users/applications
- Isolation - constraining users and applications to use only their allocated resources.

The Piccolo node architecture specification is flexible and does not mandate any particular underlying hardware/software platform (as discussed in Piccolo Deliverable report D2.1 [2]). It is therefore appropriate to consider several approaches to meeting the above requirements.

Section 2 discusses the following approaches:

- achieving a reasonable degree of isolation by using micro-actors and language virtual machines (Section 2.3), virtualisation (Docker containers Section 2.4 or kata-containers, Section 2.5), or language specific sandboxing (Section 2.2);
- relying on trusted computing (realms, Section 2.4, or Arm's TrustZone technology, Section 2.3);
- requiring applications to sign and authenticate all data packets, within a data oriented approach (Section 2.4);
- using access control and data privacy in our Behavioural risk monitoring Proof of Concept (demonstrator) (PoC).

The technology selection for a particular Piccolo node and Piccolo system depends on: the sensitivity of the workload, the acceptability of under- or over- utilisation of resources, the type of deployment (for example, private vs public network), and so on. The actual Piccolo functionality running on each node is controlled by its agent.

Section 3 discusses planned work within the timescale of the project, as well as work that we believe would be worthwhile but is beyond our scope.

In summary, we consider security and isolation to be related concepts that when implemented together provide confidence that the system resources can be allocated amongst multiple authenticated parties to perform compute with strong enforcement to prevent interference between the applications and information leakage.

2 Challenges and advances

This section discusses several ways of meeting the security and isolation challenges. It includes work done as part of a proof-of-concept implementation based on Piccolo concepts and also work done using a number of distributed computing systems. The sub-sections discuss:

- for our Behaviour risk monitoring Proof of Concept demo (PoC), the approach chosen;
- techniques suitable for GRiSP and Erlang;
- work on micro-actors;
- an approach to decentralised data processing called IceFlow;
- some experiments using kata-containers for virtualisation.

2.1 Behavioural risk monitoring PoC

The aim of this (PoC) is to instantiate a real-time data processing pipeline which embraces the concept of a Piccolo trusted in-network compute node, and manifest as a Working Demonstrator of future potentially exploitable capability applied to behavioural risk management in the automotive sector.

The goal is to perform real-time distributed, processing of sensor data from multiple sources in a road vehicle, in particular:

- Telematics data sourced from a vehicle's Controller Area Network (CAN) bus via the On-board Diagnostics (OBD) OBD-II port
- Edge-based computer-vision processing of potential hazards internal and external to the vehicle from imaging sources (cameras) on board the vehicle.

The exploitation goal is to perform 'sensor-fusion' style analytics of the data points from these sources in order to derive a real-time and continuous measure of 'behavioural risk' associated with a vehicle during each journey.

The research goal is to redistribute the classic 'edge to cloud' type computing architectures that represent the current state of the art with these types of sensor-fusion applications into one that employs in-network computing nodes that offer enhanced security/privacy, extensible feature-functionality and enhanced computing cost-efficiency.

In the first phase of the Proof of Concept (PoC) project activity, the components selected to construct the end-to-end processing pipeline are described in detail in Deliverable D1.2 of the project ("Application Design Development Report), and are summarised below:

- Eclipse Kuksa - for extraction of appropriate telemetry from the vehicle CAN
- A modified form of Sensing Feeling's redistributed Visual Processing Engine (VPE) originally designed for 5G MEC deployments
- Genivi IoT Event Analytics (IoTEA) - for transformation of vehicle telemetry into data points meaningful for behavioural risk measurement
- Bosch and Sensing Feeling's cloud-based analytics platforms for final analytics, behavioural risk computation and reporting

The Piccolo node system concept was selected to be implemented on Fluentic's Trusted In Network Computing (TINC) prototype, which aimed to deliver the following in-network computing features:

- Resource allocation through dynamic auctions
- Secure Processing (using Intel Software Guard Extensions (Intel) (SGX) enclaves)
- Function-as-a-Service (using Information-Centric Networking (ICN)/Named-Data Networking (NDN) plugin)
- Task Prioritisation and Outsourcing

Resourcing challenges at Fluentic in late 2021 resulted in TINC prototype development ceasing, and the replacement of the in-network computing node with substitutes that focus on Access Control and Anonymity instead of strong Data Security and Isolation.

In this revised approach, Sensing Feeling's edge-based Visual Processing Engine (VPE) is decomposed into two parts to deliver the PoC's Vision Processing pipeline:

- The VPE elements that handle image frame extraction and preparation - remains at the edge
- The Deep Neural Network (DNN) Model Server - introduced into an in-network compute host (Piccolo)

Access Control security between the VPE and the Model Server is implemented using authenticated and encrypted Remote Procedure Call (RPC) technology implemented using gRPC. The gRPC exchange provides the ability to off-load the compute-expensive (but stateless) task of processing image frames through the target DNNs available in the Model Server. The results of the processing are sent back to the VPE as part of this exchange, which continues to interact with the remote cloud-based Analytics Engine in the usual way, using the IoT core, based on the (originally Message Queueing Telemetry Transport) (MQTT) message broker architecture.

Figure 1 illustrates the data flows implemented in this approach.

For the PoC, image frames will be processed in the in-network Model Server in real-time, whilst the

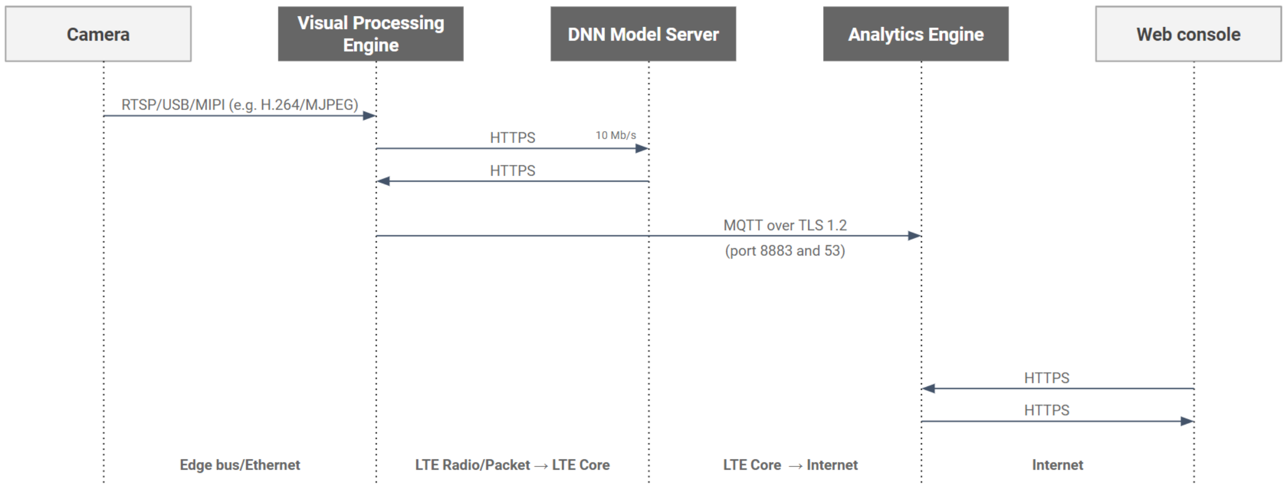

Figure 1: Behavioural Risk PoC Vision Processing Pipeline

results of the processing are combined with the telemetry stream from the vehicle at the Analytics Engine. Because the in-network element can conceptually handle the processing of frames from multiple VPEs (and hence multiple camera sources) in a stateless fashion, where images are discarded immediately after processing within the network, the rights to personal data privacy are upheld in the same manner that would be undertaken at the edge. It could be argued that the safeguarding of rights to privacy of any personal data captured at the edge is strengthened by processing the data in the network, as the image processing and deletion tasks for all edge VPEs is taking place in fewer ‘ephemeral’ in-network compute nodes instead of many more widely-distributed edge nodes.

2.2 GRiSP Seawater SaaS Language specific sandboxing

Out of the box the Erlang VM has no support for multi tenancy, either inside one VM or within cluster of Erlang nodes connected with its built-in distribution protocol. Therefore normally one would run separate instances of Erlang VMs in containers or in a virtualisation environment for multi tenancy (Figure 2).

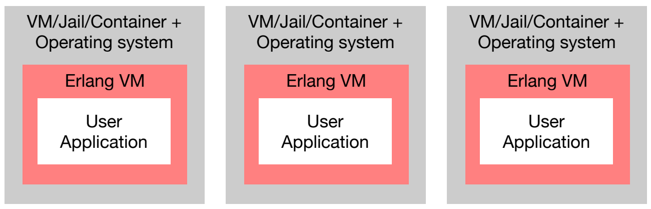
VM/Jail/Container + Operating system
Erlang VM
User Application
VM/Jail/Container + Operating system
Erlang VM
User Application
VM/Jail/Container + Operating system
Erlang VM
User Application

Figure 2: Virtualisation or container based sandboxing

There has been some implementation of Erlang-based sandboxing of Erlang code for teaching purposes[1], based on limited code interpretation. Disadvantages are slow speed of the interpreted code

[1] `https://www.tryerlang.org`

and it took a lot of effort to secure against breaches and is hard to maintain as can be seen in the really old version of Erlang running in Tryerlang (Figure 3).

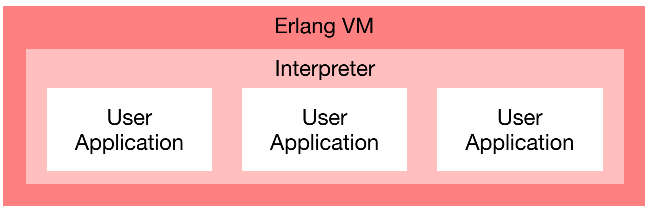

Figure 3: Interpreter based sandboxing

When one implements another language on top of the Erlang VM, running alongside Erlang code, one can achieve some level of sandboxing in the compiler and run-time for the implemented language. By taking care not to allow callouts to general Erlang run-time functions and not allowing user written Erlang code, one can make use of Erlang's language and run-time feature to provide a limited grade of isolation of multiple tenants sharing one Erlang VM.

Erlang's immutable data structures, process isolation and fault tolerance features help greatly in making this kind of language based isolation achievable with a moderate amount of security. One area that needs some extra work is the fairness of computation resource usage. On the language level, Erlang's round-robin scheduling (which behaves like preemptive scheduling) ensures that multiple processes each get their fair share of resources. So, to achieve fairness between tenants sharing one Erlang VM, one needs to make sure that a tenant doesn't simply create numerous processes in order to grab more resource. In addition it will be useful to limit the amount of compute resources a particular process is allowed to consume, and to prevent one user flooding the system with messages - essentially limiting the number of message-causing operations per user and per unit time. Any extra limitations should be handled in the run-time for the programming language sandbox.

Peer Stritzinger GmbH is building a Internet Of Things (IoT) focused Software as a Service (SaaS) which includes our Erlang based implementation of the International Electrotechnical Commission (standards body) (IEC) 61499 standard for distributed Programmable Logic Controller programming. In this run-time the language specific sandboxing techniques described above are available for the free service tier. For a free service tier it is most important to keep the cost down by sharing existing Erlang nodes where the Erlang VM schedulers can cooperate and optimise resource usage. While the same SaaS code-base will also be used in the Smart Factory use case, it will be hosted on premises and multi tenancy is not required.

In the IEC61499 run-time the language element of function-blocks is mapped to Erlang processes. The language element for event based data flow is mapped to Erlang messages. The compiler translates the IEC61499 code into pure functional Erlang callback code, which is called by a language specific run-time written in Erlang (Figure 4). In this Erlang code we have full control over message sending and receiving and can time-out the callback functions which contain the translated user functionality. In addition we can measure the CPU resources consumed by these callback functions. The amount of allowed processes per user can be controlled by the Integrated Development Environment (IDE) by limiting the number of function-blocks. Memory usage can also be controlled easily

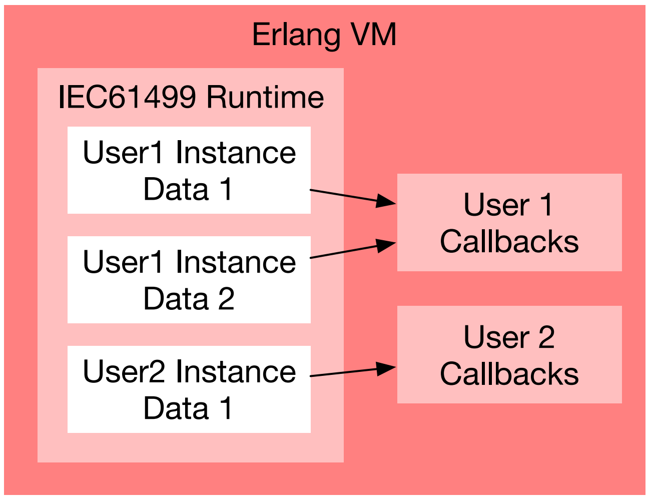

Figure 4: Sandboxing in IEC61499 Erlang implementation

since the language has very static data model which allows to measure memory usage of a function block at compile time.

The free tier on this SaaS demonstrates the functionality of the system to potential customers of paid tiers and possibly for educational purposes. Limited performance is acceptable and provides pressure to go for paid tiers where all of a user's code runs in their own Erlang instance.

Another Erlang VM based language implementations with a similar sandboxing exists for LUA [2] which given the steadily rising popularity of LUA is also something we will explore in the future.

2.3 μActor

μActor [4] is a serverless compute platform developed at the Technical University of Munich that allows running actors on a heterogeneous set of nodes—the same actors can run on small micro-controllers at the far edge of the network and large cloud servers. The actors communicate using content-based messaging: Actors subscribe to messages published to the network by expressing constraints on their content. Further information on μActor can be found in our publication [4].

The actors may originate from untrusted users who can deploy them without requiring approval by the owners of the device. Therefore, they need to be isolated from each other and limited in their access to the system resources. Hence, the security challenge discussed here is the isolation of those serverless actors across a set of heterogeneous devices. In the following, we first describe the isolation properties provided by the use of the actor model and the system's architecture and then describe the chosen method of using language virtual machines to ensure that the system's architectural properties are enforced. Finally, we discuss the possibility of protecting the language virtual machines using ARM's TrustZone technology.

[2]`https://github.com/rvirding/luerl`

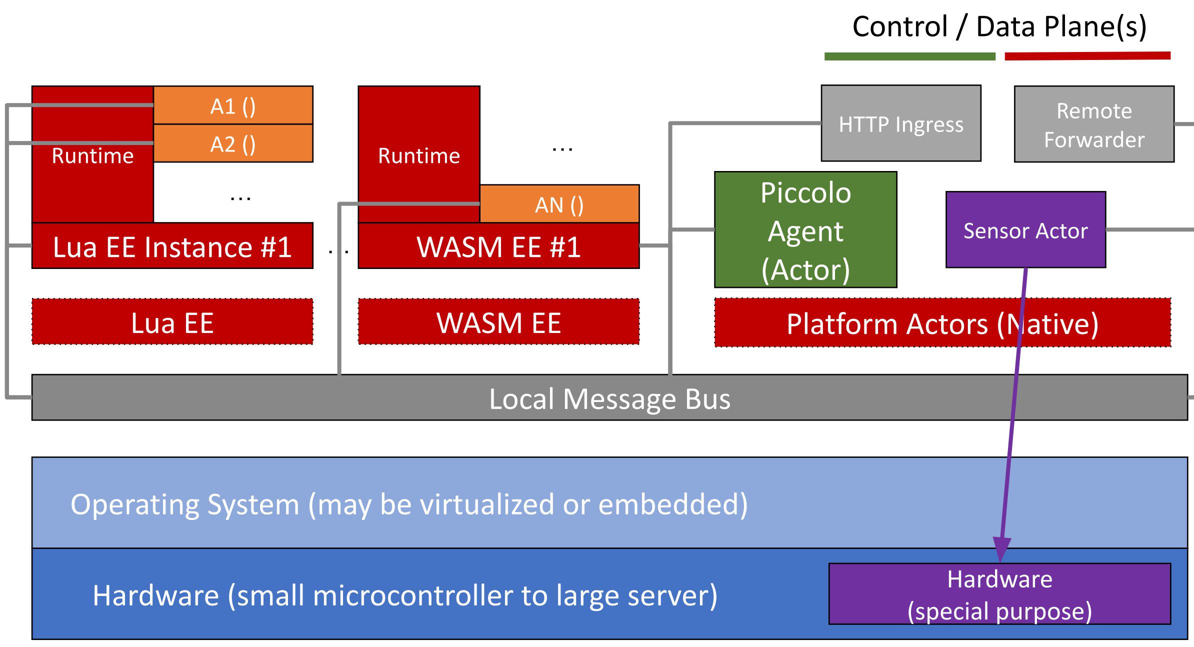

Figure 5: μActor node mapped to the Piccolo node architecture [5]

The actor model can be seen as an isolation mechanism: The actor's state is local, and actors can only interact with each other using messaging [6, 7, 8]. While the theoretical actor model proposes to use unforgeable addresses that an actor can only use if it has previously received it [7, 8], μActor employs a publish/subscribe model as its primary communication abstraction: Actors can subscribe to arbitrary data and can publish messages matching the subscriptions of arbitrary actors. We assume that the publications contain unforgeable publisher information—this can only be guaranteed on the local node and across trusted nodes—and that the receivers filter the messages they react to based on actor-specific criteria. Furthermore, the actors are only active when reacting to a message [7], which is a natural fit for serverless entities: The runtime can activate the actors at a suitable point in time and multiplex a system thread across many actors [4]—the model does not require a direct mapping between actors and threads [8].

Some actors need to interact with their environment, e.g., sensors, actuators, or external services. To enable this, the system could provide APIs to the actors received from the network that allow them to interact with these resources. The system would need to ensure the APIs are only exposed to the actors to the extent desired by the platform, i.e., not every actor should be provided with every Application Programming Interface (API). Similarly, the platform would need to ensure that there is no mutually-conflicting access from different actors [4]. Finally, exposing the APIs increases the attack surface for malicious actors. Instead of providing the actors received from the network with those APIs, μActor reserves the capability to interact with external entities to actors bundled with the platform. Those platform actors are implemented using native code, are assumed to be trusted, and are accessible to the untrusted actors only using messages. They can selectively react to these messages based on their content and origin. This separation between two types of actors assists isolation: The untrusted actors only need facilities to communicate using messaging, and the platform actor can choose to react to those messages using a resource-specific strategy.

The system needs to prevent the actors received from the network from exceeding the capabilities intended by the actor model and the platform. Therefore, the platform needs to isolate their execution and prevent them from interfering with the operation of the platform and other actors, e.g., by directly accessing their state. This requires a mechanism for isolated code execution. As μActor also targets microcontrollers, it can not rely on traditional VMs, unikernels, or containers as those devices do not possess the necessary resources and capabilities. Furthermore, those techniques employ machine-specific executables that are not portable across machines. Therefore, μActor uses language virtual machines such as Lua or V8 as the isolation mechanism for actors.

Those language virtual machines have been used in many recent serverless offerings such as Cloudflare's Workers [9] or Fastly's Compute@Edge [10]. They are the enabler technology to execute and isolate the same actors on small microcontrollers and large cloud servers. As demonstrated in [4], even an ESP32 microcontroller with less than one megabyte of RAM is able to run tens of (minimal) actors—a much larger number can be run on more powerful devices. As shown by Cloudflare[11, 12], the use of language VMs and their isolation functionality also provides benefits on servers that would be able to run containers or VMs: Their use reduces the resource consumption of serverless functions, which allows for higher density on edge servers, and they allow to reduce the cold-start delays.

μActor executes the actors, which are received as source- or bytecode, using those language virtual machines. The actors are only active while processing a message and are expected to complete in a limited amount of time. This provides the platform with full control over the execution and allows reusing a system thread across many actors. The actors are provided with a limited set of functionality that ensures the desired isolation. Some of these language virtual machines provide sandboxing capabilities that allow to run multiple mutually isolated actors within one common instance and therefore reduce their overheads [4], while others can only ensure the isolation of a single actor and would therefore require a separate VM instance for every actor. Each μActor node can run multiple language virtual machines to support different languages or isolation properties. As the actors only interact using messaging, actors can interact with other actors regardless of their implementation language.

Figure 5 shows how μActor maps to the Piccolo node architecture described in [2]: The execution environments for the untrusted actors are shown on the left side and the platform actors are shown on the right side. The language virtual machines span both the Execution Environment (EE) instances as well as the runtimes: They are both used to execute the actors and provide them with functionality such as messaging. As suggested above and demonstrated by the sensor actor, only the platform actors can interact with special-purpose hardware. Furthermore, the platform actors are also used for inter-node communication and external communication interfaces. All actors running on the node are connected to a central messaging bus, which allows them to communicate. Further information on μActor's mapping to the Piccolo node architecture can be found in [5].

The language runtimes protect the platform from malicious actors, but they do not protect the actors from an untrusted node. To support this, the language VMs may be protected using an enclave technology such as ARM's TrustZone. There is initial work exploring the execution of Lua (which is one of the languages used by μActor) inside TrustZone [13], which shows that this combination is feasible but requires modifications to the runtimes as they depend on system APIs not provided by the trusted environment. Additional information and performance measurements can be found in [13]. We plan to integrate this enclave technology into the μActor prototype as part of our future work.

A further area of future work is the security of the messaging mechanism. While the assumption of unforgeable publisher information within a node provides a basic integrity mechanism, integrity should be ensured using cryptographic measures. The current system provides no confidentiality. It is not possible to use end-to-end encryption for entire messages as their content must be accessible during forwarding.

2.4 IceFlow - Decentralized Data Processing

IceFlow [14] – a Dataflow system approach that supports traditional Dataflow with Information-Centric principles as well - can be used as a drop-in replacement for existing Dataflow-based frameworks. Current IceFlow's objectives are: (i) complexity reduction by replacing connection-based overlays in Dataflow systems with named-data communication and corresponding orchestration requirements; (ii) efficient communication by reducing data duplication; and (iii) enabling further performance improvements through more direct communication and caching the data in the network. IceFlow, an NDN-based stream processor, wraps every compute function in the pipeline with an NDN actor that consumes data for the compute function and produces the computed data. That being said, IceFlow operates on top of a distributed architecture connected via NDN. IceFlow's approach towards security starts with using containers for the actors across the infrastructure. Furthermore, it utilizes the NDN way of securing and authenticating every data packet across the network layer.

As IceFlow actors can be initiated from suspicious sources, it is essential to isolate each actor and limit the access of the available resources. We have opted to containerize our DataFlow system into Docker container as docker virtualization platforms have the best as well as near-native performances whereas hypervisors exhibit significant networking and memory overhead, but on Input / Output (I/O) and Central Processing Unit (CPU) bound tasks typically perform on-par with native. Secure containers particularly suffer from overhead in the I/O subsystem, but promising alternatives are being developed. Moreover, unikernels reduce the amount of code, therefore application images are small and reducing attack surface, which increases security. Although it naturally comes at a cost for the reduction in complexity, most often in the form of decreased interoperability with existing applications. Since applications must be compiled to run in a unikernel, possession of this source code is a hard requirement. Moreover, due to the single address space and single process design of unikernels, there is no support for running multiple processes in one unikernel. Separating the applications up into multiple unikernels will induce at least some communication overhead between the different parts [15]. Therefore, we describe how IceFlow actors are segregated from each other by the basic isolation properties provided by Docker.

While IceFlow actors exchange named data among themselves for further processing, it is unarguably crucial to maintain the trustworthiness among them. The Named Data Networking (NDN) architecture builds data authentication into the network layer by requiring all applications to sign and authenticate every data packet. To make this authentication usable, the decision about which keys can sign which data and the procedure of signature verification can be automated using "*Trust Schema*"[16] which consists of a set of rules that describes the connection between the data name and the name of its corresponding signing key. As IceFlow uses NDN for actor's communications, it adopts this security model to ensure that the data communication is more reliable and trustworthy. The trust schema in the Iceflow defines the relationship between the data namespace and the key namespace under the dataflow paradigm. Following the key retrieval pattern mentioned in [16], the key retrieval process in Iceflow starts from the stream produced by the actor up to the producer until it reaches the trust anchor, e.g., for a data item with a name:

$< appX >< producer - fct1 >< stream1 >< data1 >$

The rules to retrieve the corresponding key will extract the key name as:

$< appX >< producer - fct1 >< stream1 >$

Then it will follow the pattern to reach the trust anchor. The trust anchor is considered a pre-authenticated root key of an application. It can be secured using trust models such as Public Key Infrastructure (PKI) or web-of-trust. It is noteworthy that every successful key retrieval patterns always end up at the trust anchor.

This approach enables consumers to discover automatically which keys to use to verify individual data packets. It also provides producers with an automatic decision process about which keys to use to sign data packets and, if keys are missing, how to generate keys while assuring that they are used only within a narrowly defined scope. When receiving the data packets, consumers will also extract the key name from the data name and match it with the rules. When validated, it can start a process to retrieve the corresponding key. Note that keys in NDN are also fetched in the form of data packets. Working on the stream granularity decreases the privilege scope of the keys across the system. So the stream key is used to sign the data packets requested only by the consumers of this particular stream. This decreases the threat effects in case of a key compromise [16].

Designing the namespace is challenging as it needs to be valid and efficient from the application's perspective, the network, and the trust schema. This work defines an NDN trust schema variant targeting the Dataflow paradigm. It preserves the relationship between the key and data names and maintains correct and efficient naming between consumers and producers across the Dataflow pipeline.

In our initial IceFlow prototype, consumers/producers along with predefined compute functions are packaged into containers. These containers are being hosted by Raspbian-OS where a NDN Forwarding Daemon (NFD) Forwarding Daemon) resides. In Iceflow the Piccolo agent, which represents the control plane, is responsible for initiating the dataflow pipeline. Later, the NFD will manage the data plane interactions by routing the interests/data to other containers across the cluster. Figure 6 represents the initial setup of a node of our raspberry pi cluster. A Piccolo agent receives the configuration

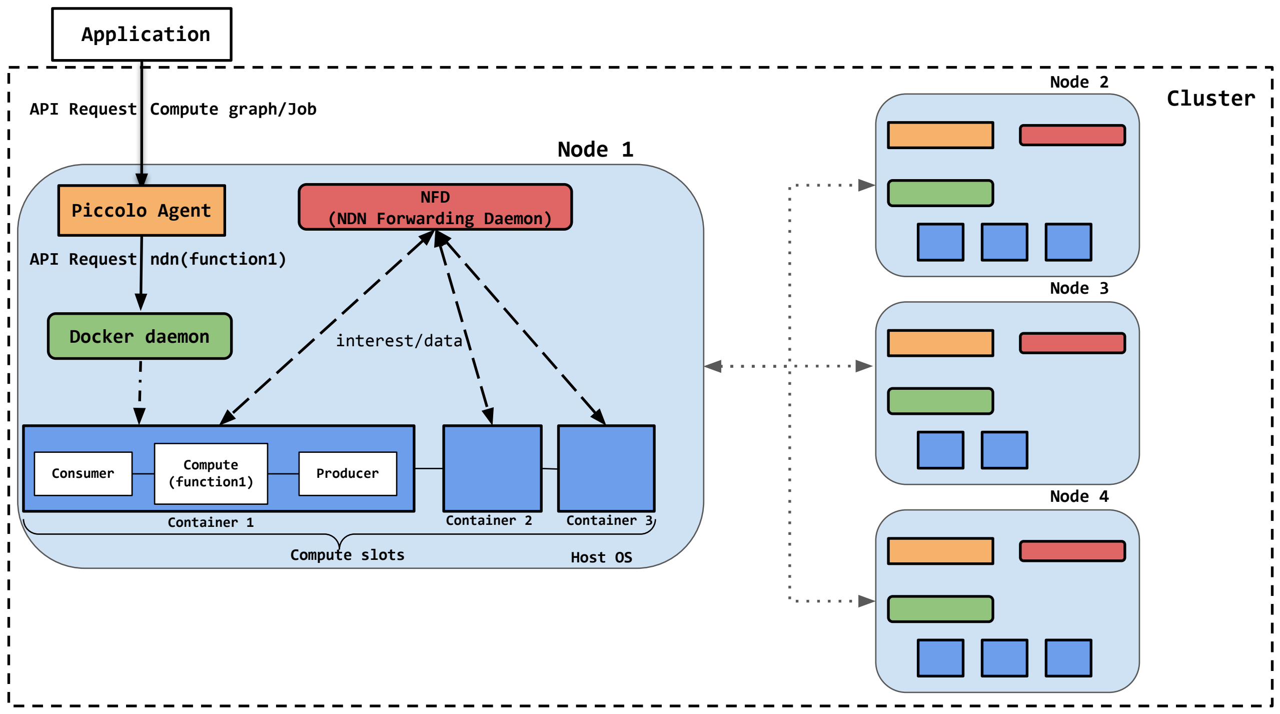

Figure 6: Containerized IceFlow

and the compute graph via agent API from the application. It translates the requests to docker API interactions. While *dockerd* spawns new containers, it creates a set of namespaces and control groups for the containers, which is the first and most straightforward form of process isolation as processes running within a container cannot see processes running in another container, or in the host system.

As there is no mapping to lower layer IP addresses, IceFlow actors use application-relevant names in a suitable namespace. These actors are isolated from each other and their locations as it can change during a distributed program execution. It no longer matters whether the Dataflow system is running in a local network or over the internet. It ensures that other actors do not have the access to containers and only interact through a NDN Forwarding Daemon (NFD). We are using User Datagram Protocol (UDP) interfaces to communicate the NFD's route for interest/data packets in our current protoype.

A container also gets its own network stack, meaning that a container does not get privileged access to the sockets or interfaces of other containers. It is noteworthy that if the host system is setup accordingly, containers can interact with each other through their respective network interfaces, which is not implemented in our current experiment.

In conclusion, security in Iceflow is realized using two mechanisms. First, we use containers to make sure that actors are isolated and we can support multi-tenant third-party applications across the infrastructure. Second, using the defined trust schema model at the network layer, every data packet is signed and later authenticated using the corresponding key. This model also ensures end-to-end data authenticity among the actors.

2.5 Using kata-containers for virtualisation

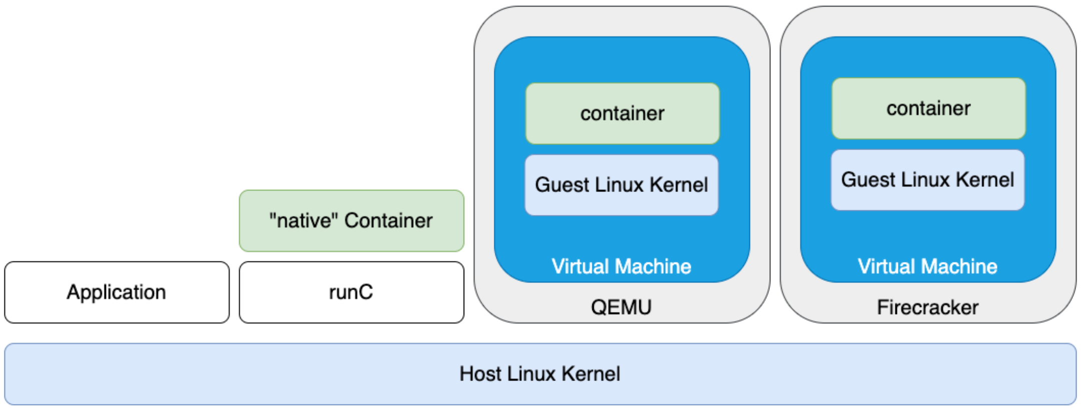

Figure 7: kata-containers

Kata-containers [17] is a container runtime that enables the use of lightweight VMs and containers. It aims to combine the convenience of deploying containers with the security of lightweight VMs without the overhead of managing full virtual machines. Kata-containers extends the isolation of containers by using a separate guest kernel for each container (or group of containers). The containers appear to be executing within their own Virtual Machine and a hypervisor is used to provide efficient isolation of network, I/O and memory using hardware virtualization extensions. The guest kernel is configured to be as small as possible. Kata-containers can be configured to use different hypervisor solutions for creating lightweight VMs as shown in Figure 7. We looked at two hypervisor backends: QEMU/KVM [18] in Section 2.5.1 and AWS firecracker [19] in Section 2.5.2.

The Kata Containers runtime is compatible with the OCI runtime specification [20] and therefore works seamlessly with the Kubernetes Container Runtime Interface (CRI) through the CRI-O and containerd implementations. The container runtime to be used when deploying a particular application can be set in the YAML Ain't Markup Language (YAML) specification for the application. This enables different applications to use different container runtimes depending on differing requirements with respect to isolation. Using kata-containers to isolate workloads in this way makes it much harder for those workloads to compromise the node (via direct attacks on the host kernel for example).

There are however some limitations when using kata-containers as a replacement for runc. The most significant being:

- Host-networking: host network access (using the host IP address and network stack) is not supported. It is not possible to directly access the host networking configuration from within the VM. Networking for the VM (and its associated containers) is configured via the usual Container Networking Interface.
- Host resource sharing: Privileged containers get full access to the guest VM in addition to some host access. The container runs with elevated capabilities within the guest and is granted access to guest devices instead of the host devices. The container may also be granted full access to a subset of host devices.

These limitations would not prevent the use of kata-containers in the context of the Piccolo architecture. If direct access to hardware is required by less trustworthy applications that we want to isolate (using kata-containers) then we would use a trusted proxy container deployed using the standard runc container runtime to provide access-as-a-service to the isolated applications.

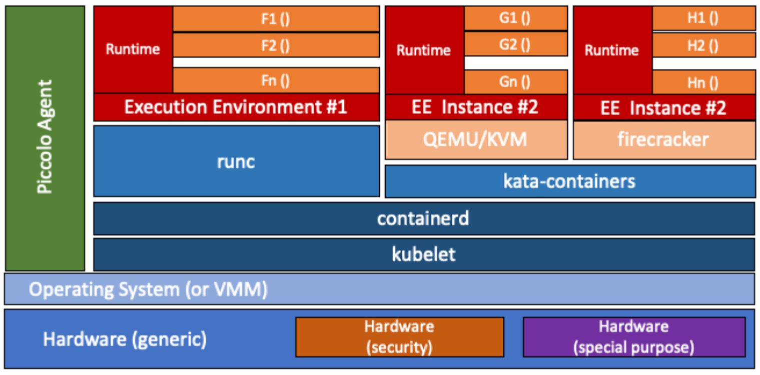

Figure 8: Piccolo integration

Figure 8 shows how kata-containers could be used to enable containers to be deployed using lightweight virtual machines on a Piccolo node. In this example, k3s is being used as the container orchestrator directed by the Piccolo Agent software which controls choice of which container runtime is used for each Piccolo Execution environment.

2.5.1 QEMU/KVM

Quick EMUlator (software) (QEMU) is a generic and open source machine emulator. In our case, we are using QEMU for system emulation where it provides a virtual model of an entire machine (CPU, memory and emulated devices) to run a guest Operating System. Kernel-based Virtual Machine (Kernel-based Virtual Machine (KVM)) is a virtualization module in the Linux kernel that allows the kernel to function as a hypervisor to allow such a guest Operating System to run directly on the host CPU at near-native speed. QEMU also includes functionality to allow "user mode emulation" where processes compiled for one Instruction Set Architecture can be executed on a CPU with a different Instruction Set Architecture. This ability to emulate many different systems and devices coupled with the cross-Instruction Set Architecture functionality makes QEMU more capable than firecracker at the cost of more software complexity.

QEMU uses the vhost-net driver (on the host) to connect to virtio-net (within the guest). This improves networking performance for the Linux guest OS by offloading the handling of network packets from the hypervisor to the host kernel driver. Using vhost-net reduces the number of system calls involved in virtio networking.

Using QEMU it is possible to share host filesystem resources with VM using virtio-fs [21].

	Raspberry Pi 4	Solidrun Honeycomb LX2
CPU	4 x Cortex-A72	16 x Cortex-A72
CPU Frequency	1.5 GHz	2.0 GHz
RAM	4 GB (LPDDR4-3200)	64 GB (DDR4-2600)
Network	1 GigE	1 GigE
Kernel	5.11.0	5.4.47
OS	Ubuntu 20.04	Ubuntu 20.04.02
k3s	v1.21.2+k3s1	
containerd	v1.5.2-0	
kata-runtime	v2.1.1	
QEMU	v5.1.0	
firecracker	v0.23.5	

Table 1: Platform Specifications

2.5.2 AWS Firecracker

Firecracker is a virtual machine monitor (VMM) that also uses the Linux Kernel-based Virtual Machine (KVM) to create and manage lightweight VMs known as microVMs. Firecracker has a minimalist design - it excludes unnecessary devices and guest functionality to reduce the memory footprint and attack surface area of each microVM. Firecracker provides a minimal required device model to the guest operating system (only 5 emulated devices are available: virtio-net, virtio-block, virtio-vsock, serial console, and a minimal keyboard controller used only to stop the microVM).

Firecracker is designed to be as small as possible with fast startup time. It has a reduced attack surface area by including only the minimum required devices. As a consequenc eaccess to host devices such as GPU or other accelerators is not possible from within the guest. Firecracker does not support the use of virtio-fs and only supports block-devices for storage - this precludes the sharing of file systems with the host (as well as requiring the host to provide block-based storage).

Firecracker uses virtio-net but does not support using vhost-net. This may limit its networking performance compared to QEMU.

2.5.3 Experimental setup

We performed experiments using two systems which represent different points in the spectrum of hardware used for Edge Compute: (1) The Solidrun Honeycomb LX2 is a feature-rich mini-ITX development board using NXP Semiconductors QorlQ® Layerscape® LX2160A Processor [22]. (2) The Raspberry Pi4 is a low-cost single board computer built using Broadcom's BCM2711 System on Chip [23]. Table 1 shows the specifications of the platforms.

We installed kata-containers on both platforms along with k3s [24] – a Kubernetes distribution built for IoT and Edge Computing. We built the kata-container components from source as well as building a tightly configured Linux kernel that would be run inside each VM. We installed versions of QEMU and firecracker and configured k3s for deploying containers via kata-containers. Both QEMU and firecracker were configured to use a single vCPU and to use tcfilter to connect the VM to the host network.

2.5.4 Results

We measured the network performance difference between deploying the same container image using the runc container runtime, using kata-containers with QEMU/KVM and using kata-containers using firecracker. We used *iperf3* [25] to measure maximum achievable bandwidth between a container running the iperf3 client and an iperf3 server. Table 2 shows the results. We measured the bandwidth in the following scenarios:

1. From the container to an iperf3 server running on a remote computer connected via a 1 GigE switch (Figure 9)
2. From the container to an iperf3 server running on the test system itself (localhost) (Figure 10)
3. From the container to an iperf3 server running in a container on the test system using the same container-runtime as the client (Figure 11)

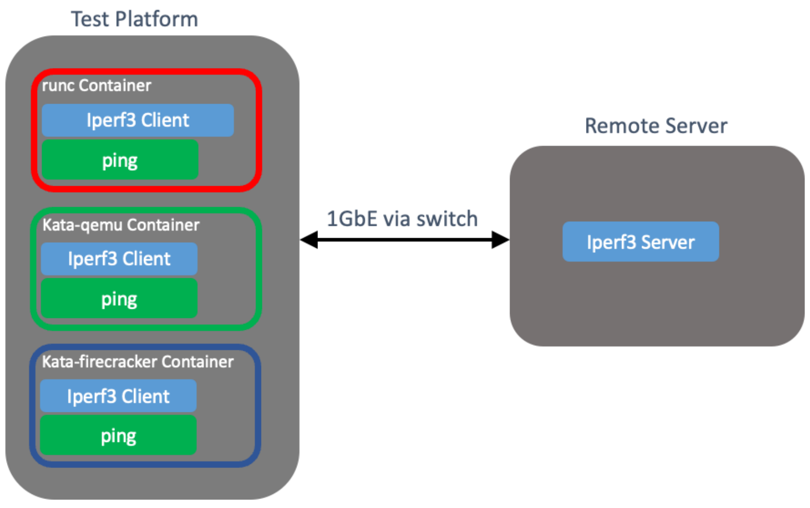

Figure 9: Remote test

The result of network bandwidth tests are shown in Table2. All the configurations were able to get close to saturating the external link (1Gbit/s). In these cases, the extra cost of running a virtual machine with a guest kernel and network stack does not significantly affect the maximum bandwidth.

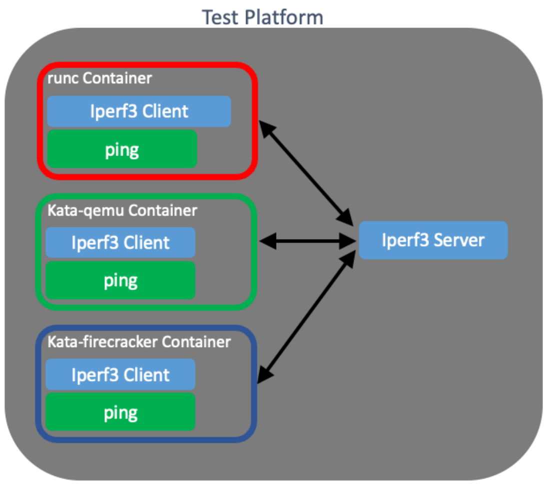

Figure 10: Local test

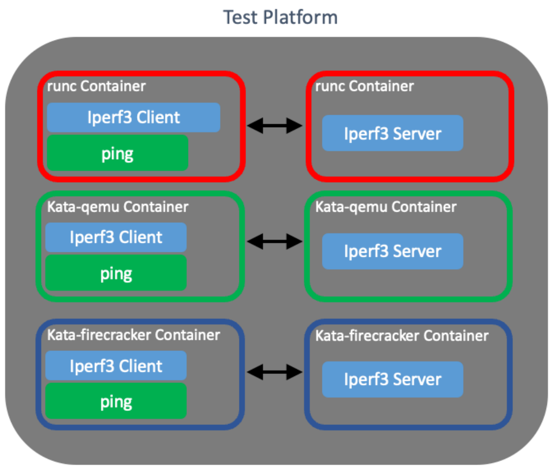

Figure 11: Runtime-runtime test

The CPU and memory performance difference between platforms are very apparent in the local test where the LX2 is able to sustain almost 3x more internal bandwidth than RaspberryPi4 (RPi4) (Raspberry Pi4). The RPi4 is limited by the internal bus in the System on a Chip (SoC) to a much lower maximum memory bandwidth of around 5.5 GB/s (compared to the maximum LPDDR4-3200 DRAM performance of 12.8 GB/s) This explains the difference between the two platforms which is larger than the difference in CPU frequency. The overhead of virtualization is now more noticeable - the maximum performance achieved was 67% of the runc performance using kata-qemu on the LX2 with the kata-firecracker performance limited to around 58% of the kata-qemu performance. We would expect kata-qemu to provide more network bandwidth than kata-firecracker because kata-qemu uses the more efficient vhost-net driver.

On the RPi4 the difference between kata-qemu and kata-firecracker is less pronounced possibly because the performance of both is constrained by the memory-bandwidth limitation.

The container-container tests show similar results to the local tests - the overheads of virtualization, with two guest kernels and networking stacks, are significant. Again the different between kata-qemu and kata-firecracker on the RPi4 is less pronounced than on the LX2.

	Remote		Local		Container	
	LX2	RPi4	LX2	RPi4	LX2	RPi4
runc	933	934	17520	6233	15089	5467
kata-qemu	99%	96%	67%	37%	88%	32%
kata-firecracker	99%	92%	39%	31%	43%	27%

Table 2: iperf3 Bandwidth Mbits/sec (runc). % normalised to runc

We used the *ping* tool [26] to measure network latency and created a container image with the ping tool. Ping uses the Internet Control Message Protocol (ICMP) protocol ECHO_REQUEST datagram and measures and records the round-trip time of the packet and any losses along the way. Table 3 shows the results. We measured the round-trip network latency for the following situations:

- From the container to a remote computer connected to the same 1 GigE switch as the test system
- From the container to the test system itself (localhost)

The result of network latency tests are shown in Table 3. There is a significant increase in latency when using lightweight virtualization. This is due to running a guest kernel and network stack - each outbound ping packet is effectively travelling from guest-userspace -> guest-kernel -> host-kernel network driver and then the reply follows the reverse path. Again kata-firecracker appears to incur a greater overhead than kata-qemu (which uses the more efficient vhost-net driver). Both kata-qemu and kata-firecracker seem to have lower overhead on the RPi4 comapred to the LX2. We have not determined why this is the case. It may be due to RPi4 running a newer kernel (5.11.0) compared to the LX2 (5.4.47).

	Remote		Local	
	LX2	RPi4	LX2	RPi4
runc	0.272	0.552	0.141	0.537
kata-qemu	× 1.86	× 1.38	× 2.29	× 1.15
kata-firecracker	× 2.24	× 1.78	× 3.38	× 1.68

Table 3: ping Mean latency milliseconds (runc). × normalised to runc

Finally, to obtain more application-centered measurements we deployed the Eclipse Mosquitto MQTT message broker MQTT using the three container runtimes. Table 4 shows the results. We used a MQTT benchmarking tool, mqtt-bench [27], running on a remote machine to simulate the MQTT broker. The benchmarking tool simulated 35 clients each sending 20000 messages of size 100 bytes with "exactly once" delivery. We measured the resident set size for Mosquitto when deployed using the three container runtimes.

	Mean Msg Time		Average Bandwidth	
	LX2	RPi4	LX2	RPi4
runc	2.48	7.79	384.6	127.9
kata-qemu	× 1.19	× 1.20	85%	83%
kata-firecracker	× 1.67	× 1.27	62%	79%

Table 4: mqtt-bench Msg time milliseconds (runc) and Average Bandwidth msg/sec (runc). × and % normalised to runc

MQTT test results are shown in Table 4. The average bandwidth when using kata-qemu is relatively close to the raw iperf3 numbers on both platforms. The kata-firecracker results are lower especially on the LX2. The mean message time results show kata-qemu with around a 20% increase on both platforms. kata-firecracker is once again much worse on the LX2.

Table 5 shows the resident set size for Mosquitto using the different container runtimes. Both kata-qemu and kata-firecracker require more memory than runc as they execute a guest Linux kernel. The kernel used for firecracker has been optimised to remove unneeded functionality and this can be seen its smaller Resident Set Size (RSS).

	Memory Usage MB	
	LX2	RPi4
runc	3.6	3.0
kata-qemu	127.0	125.0
kata-firecracker	98.0	86.0

Table 5: Resident Set Size for Mosquitto

2.5.5 Conclusion

The use of kata-containers to provide lightweight virtualization enables more secure deployment of computation in scenarios where multiple applications are executing. These lightweight VMs are not suitable for every application (such as those requiring access to specialised hardware accelerators for example) and we would anticipate that the best solution would be a hybrid model of standard containers plus containers executing within lightweight VMs. Also, as we have seen, the overheads of using lightweight VMs are not insignificant and may be too high for an application that requires maximum performance from the hardware. However, in the context of Piccolo and Edge Computing in general these overheads may still be worthwhile for the benefits of isolation that are achieved when using lightweight virtual machines.

The QEMU and firecracker backends for kata-containers have differing properties that mean neither would be the default choice when looking for a more secure container runtime. QEMU is more flexible and provides more of the standard container runtime features (such as host file-sharing, access to hardware devices). Firecracker is more streamlined and its lower memory footprint would allow more lightweight VMs to be deployed on a machine.

3 Next Steps

Here we briefly outline future work, both that which is planned within the project and potential activity beyond Piccolo's lifetime / scope.

Based on our work on kata-containers for virtualisation (Section 2.5) we identify two new approaches that provide different ways to improve the isolation of an application on a Piccolo node.

The first approach is based on seL4 [28], which is an open-source microkernel and also provides a trusted hypervisor with a small Trusted Computing Base. As well as protecting guests and their applications from each other, this arrangement could protect the guests from a malicious host. IceCap [29] is an example of a virtualization platform with a minimal trusted computing base that aims to provide guests with confidentiality and integrity guarantees. We are making IceCap into a hypervisor backend for kata-containers that we can then compare with QEMU and firecracker.

The second approach is based on features from the Arm Confidential Computing Architecture [30] using realms to provide isolation of applications from each other and the host. The most straightforward application of realms is to isolate each Virtual Machine within its own realm. A more radical approach would be to use realms to isolate userspace processes. Unfortunately it will not be possible to perform this research within the lifetime of the Piccolo project - although some simulation models and associated software components are available today, they are beta-releases and do not implement all the required functionality.

As discussed in section 2.3, we plan to protect the actors using trusted execution environments (such as Arm's TrustZone) and to explore security mechanisms related to the messaging functionality. However, both these planned activities are out of scope for this project. Other potential issues for actor approaches - where no work is planned at the moment - include how to check for correctness across a chain of micro-actors, and how to authenticate that a message has come from a genuine sender actor.

Turning next to Iceflow 2.4, we identify two future approaches that provide ways to improve the performance and security isolation of IceFlow on a Piccolo node.

Initially, every application has multiple trust anchors where the local Piccolo agent in every node is responsible for generating a root key for the actors running on this specific node. In future, another approach would be having one trust anchor per application generated by the Piccolo agent, which instantiates the DataFlow pipeline of the application. This trust anchor would be solely responsible for all the data signing by producers and the data authentication by consumers. These two approaches will affect the namespaces' design, as the trust schema plays an important role here. Our next steps not only include testing these approaches but also deciding suitable namespace design that fits the application, the network, and the trust schema.

For performance isolation, we have adopted the isolation features that are provided by docker containers: every compute function runs in the container (whereas NFD runs on host Operating System (OS), which will also be containerised in future). For efficiency, data communications among IceFlow actors of an application running in different containers on the same node do not need to communicate via NFD; instead direct communication can be used. This approach will not only reduce the communication overhead but also improve the performance of latency critical applications.

Finally, considering Erlang (Section 2.2). Each Erlang node could be run in its own ARM Realm [30] to achieve hardware assisted security isolation. The Erlang nodes could connect securely with Erlang's distribution protocol (it comes with TLS support out of the box). Taken together, one can achieve a very interesting secure, distributed, multi-tenant computing platform. It would allow distributed Erlang applications to be completely isolated from the host operating system and hypervisor as well as from each other.

An interesting future challenge here is adding distributed key management infrastructure for encrypted Erlang distribution which would make large scale secure distributed applications possible.

A couple of other ideas are: to add some firewall-like features between Erlang nodes [31], and to run multiple Erlang VM instances on the seL4 secure micro-kernel, which isolates the Erlang VMs (Kry10 project [32]).

References

[1] Piccolo Project. *Use cases, Application Designs and Technical Requirements*. Deliverable D1.1. 2021.

[2] Piccolo Project. *Piccolo Node Definition*. Deliverable D2.1. 2021.

[3] Piccolo Project. *Architectural Invariants for Distributed Computing*. Deliverable D3.1. 2021.

[4] Raphael Hetzel, Teemu Kärkkäinen, and Jörg Ott. “μActor: Stateful Serverless at the Edge”. In: *Proceedings of the 1st Workshop on Serverless Mobile Networking for 6G Communications*. MobileServerless’21. Association for Computing Machinery, 2021, 1–6.

[5] Piccolo Project. *Initial Report on PoC Implementation of a Piccolo Node*. Deliverable D2.2. 2021.

[6] Carl Hewitt, Peter Bishop, and Richard Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. 1973, 235–245.

[7] Carl Hewitt and Henry G. Baker. “Laws for Communicating Parallel Processes”. In: *Proceedings of the IFIP Congress 1977*. 1977, pp. 987–992.

[8] Carl Hewitt. “Actor Model for Discretionary, Adaptive Concurrency”. In: *Computing Research Repository (CoRR)* abs/1008.1459 (2015). version 38. URL: `http://arxiv.org/abs/1008.1459v38`.

[9] Cloudflare, Inc. *Cloudflare Workers*. URL: `https://workers.cloudflare.com/` (visited on 03/02/2022).

[10] Fastly, Inc. *Fastly Compute@Edge*. URL: `https://www.fastly.com/products/edge-compute/serverless` (visited on 03/02/2022).

[11] Kenton Varda. *Introducing Cloudflare Workers: Run JavaScript Service Workers at the Edge*. 2017. URL: `https://blog.cloudflare.com/introducing-cloudflare-workers/` (visited on 03/02/2022).

[12] Zack Bloom. *Cloud Computing without Containers*. 2018. URL: `https://blog.cloudflare.com/introducing-cloudflare-workers/` (visited on 03/02/2022).

[13] Adrian Steffan. “Running a language interpreter inside the ARM TrustZone: An exploration of dynamic code execution in trusted execution environments”. Bachelor’s Thesis. Technical University of Munich, 2020. URL: `https://adriansteffan.com/pdf/bthesis.pdf`.

[14] Dirk Kutscher, Laura Al Wardani, and T M Rayhan Gias. “Vision: Information-Centric Dataflow: Re-Imagining Reactive Distributed Computing”. In: *Proceedings of the 8th ACM Conference on Information-Centric Networking*. ICN ’21. Paris, France: Association for Computing Machinery, 2021, 52–58. ISBN: 9781450384605. DOI: `10.1145/3460417.3482975`. URL: `https://doi.org/10.1145/3460417.3482975`.

[15] Vincent van Rijn. *A Study of Performance and Security Across the Virtualization Spectrum*. Master’s Thesis. 2021. URL: `http://repository.tudelft.nl/`.

[16] Yingdi Yu et al. “Schematizing trust in named data networking”. In: *proceedings of the 2nd ACM Conference on Information-Centric Networking*. 2015, pp. 177–186.

[17] Kata Containers Community. *Kata containers Description*. 2021. URL: https://katacontainers.io (visited on 03/03/2021).

[18] Alexandru Agache et al. “Firecracker: Lightweight Virtualization for Serverless Applications”. In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[19] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’05. Anaheim, CA: USENIX Association, 2005, p. 41.

[20] *"Open Container Initiative Runtime Specification"*. Jan. 17, 2021. URL: https://github.com/opencontainers/runtime-spec/blob/main/spec.md.

[21] *virtio-fs*. 2022. URL: https://virtio-fs.gitlab.io (visited on 03/18/2022).

[22] Solidrun. *HONEYCOMB LX2 WORKSTATION*. Dec. 14, 2021. URL: https://www.solid-run.com/arm-servers-networking-platforms/honeycomb-workstation/.

[23] Raspberry PI. *Raspberry Pi 4 Tech Specs*. Dec. 14, 2021. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

[24] *Lightweight Kubernetes"*. Jan. 18, 2022. URL: https://k3s.io.

[25] *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. Dec. 14, 2021. URL: https://iperf.fr.

[26] *ping (8) Linux manual page*. Dec. 14, 2021. URL: https://man7.org/linux/man-pages/man8/ping.8.html.

[27] *MQTT Benchmarking Tool*. Dec. 14, 2021. URL: https://github.com/krylovsk/mqtt-benchmark.

[28] Gernot Heiser, Gerwin Klein, and June Andronick. “SeL4 in Australia: From Research to Real-World Trustworthy Systems”. In: 63.4 (Apr. 2020), pp. 72–75. ISSN: 0001-0782 (print), 1557-7317 (electronic). DOI: https://doi.org/10.1145/3378426.

[29] 2022. URL: https://gitlab.com/arm-research/security/icecap/icecap (visited on 03/18/2022).

[30] Arm Ltd. *Arm Confidential Computing Architecture*. URL: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture (visited on 03/04/2022).

[31] potato salad. *RFC: Erlang Dist Security Filtering Prototype*. URL: https://erlangforums.com/t/rfc-erlang-dist-security-filtering-prototype/1002 (visited on 03/31/2022).

[32] Kry10 Ltd. *Kry10 Secure Platform*. URL: https://kry10.com (visited on 03/31/2022).