WEB ASSEMBLY OPERATING SYSTEM FOR IOT DEVICES.



**UNIVERSITY OF NAIROBI**

**WEB ASSEMBLY BASED OPERATING SYSTEM FOR IOT DEVICES**

By James Kiarie

Registration Number: P15/ 1667/2019

Supervised by Dr. Wanjiku Ng’ang’a

November 2022

This project proposal has been submitted in partial fulfillment of the requirements of the Bachelor of Science in Computer Science as offered by the University of Nairobi.

# Acknowledgements

Dr. Wanjiku, for kind guidance.

Phillip Opperman, writer of “Writing an OS is Rust” blog. The blog singlehandedly made OS development to stop looking like a craft reserved for wizards.

Stephen Marz, writer of the “The Adventures of OS” blog. Took the OS development tutorials beyond just writing “hello world”.

Lin Clark, for simplifying complex webassembly jargon using mortal language... and simple cartoon illustrations.

The writers of “Wasmachine: Bring IOT up to speed with Web Assembly OS” research paper. This was the bedrock of this project.

All the Rust, wasm and riscv teams for creating an awesome well\_thought\_out tech.

# Introduction

It is a dream of many a computer scientist to control matter. To make all matter all around us programmable.

One step towards this direction is through embedded programming. There was a time when embedded programming was simple, all you had to was to read the data sheet of a piece of hardware, abstract that data sheet using data structures and finally manipulate the exposed registers using MMIO programming. Deployment was simple too, sealing off the programming jack pin was enough. Maintaining the embedded software was not a common occurence.

But now, things have changed. Things have become complex. Both the hardware and software are more complicated. For example, reading the hardware data sheet is not enough, you have to understand the ISA, the corresponding hardwired security implementations and additional compatible circuit extensions. Embedded software now deals with network connectivity... Bringing in a whole set of cybersecurity modules and cloud integration drama.

Now the IoT architecture roughly looks like this [1] :

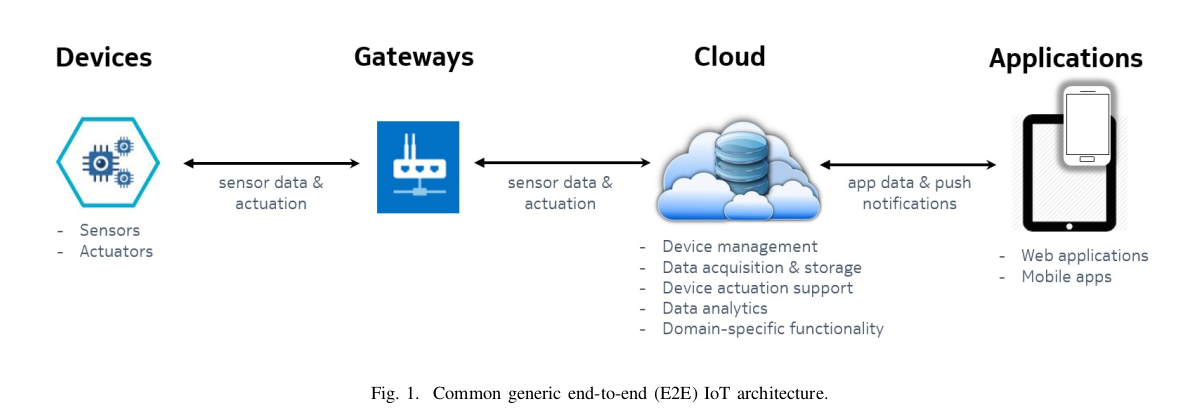
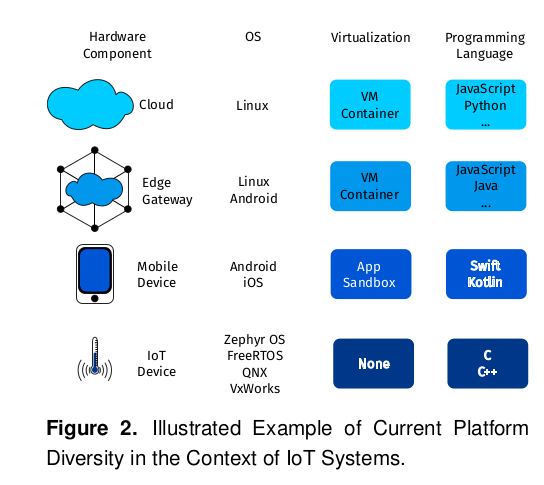


Figure 1 Image borrowed from Taivalsaari, On the development of IoT systems [1]

As a result of the above IOT architecture, building an average IOT system requires the development team to use a lot of unassociated technologies. For example; Assembly for the hardware, C/Rust for embedded programming, distributed programming for building the immediate network infrastructure, docker and kurbenetes for implementing microservices over the cloud, web languages to build a website that acts as a remote interface to the embedded devices... And probably kotlin for a mobile app that interacts with the embedded devices.

Here is a rough illustration of the different technologies used [2] :



This complexity can be summarized to three causes ; heterogeneity of the devices used, inconsistency of the communication protocols used between the heterogenous devices and the necessity to use specialized tools and programming languages.

Below are the solutions as proposed in the paper by Mikkonen [2].

To solve the problem of heterogeneity of devices, Mikkonen proposes a universal API for that describes the abstraction and interaction with known generic devices. For example, the API should specify how to abstract and interact with cameras or heat sensors.

To solve the inconsistency of communication protocols used between heterogenous devices, Mikkonen proposes a standard set of communication protocols to be specified for each known heterogeneous interaction. The paper was in support for the use of existing web protocols for simplicity and easy adoption.

The above two problems have been partially solved by initiatives such as Web of Things Standards (WoT) and Open Communication Foundation (OCF).

The third problem ; having too many tools and programming languages involved in development, causes two kinds of complexity. One, It makes it hard for a small team to properly master and keep up with the tools involved. Two, It makes it hard to migrate code from one node to another within this distributed network of IOT devices. Code migration portability is important because of many obvious reasons like code reusability or code maintainability. However, the major reason why code migration portability is important is because current IOT favors edge computing (fog computing). In edge computing, it is expected that at some point, the edge device may run the algorithms that are usually ran in the cloud (for example a data processing algorithm). If the data processing algorithm found in the cloud node is written in python, does that mean that the developer has to re-write that algorithm in C/Rust just for it to run on the edge device?

This code portability problem can be solved through two methods : One, finding a programming language or tool that gets explicitly and solely used through out the project OR find a way to package the programs written in different languages into a portable package.

Here is an image showing only one programming language being used versus many languages:

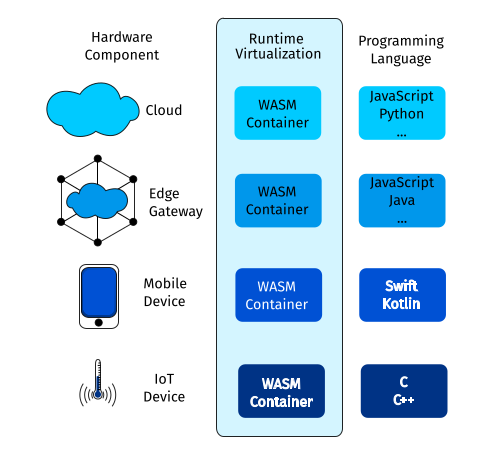
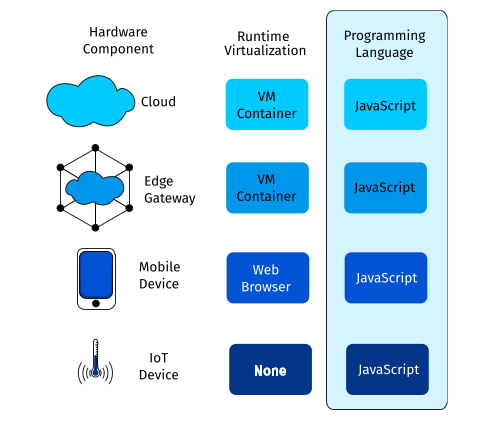


Figure 2 Forced to use 1 language (Left)

Figure 3 Multiple languages used but the pckaging is constant (Right)

Solution one is unfavorable because each language or tool has its purpose, It is not okay to use a single language to solve every problem.

Solution two is considerably favorable ; developers use the languages they are comfortable in but all the programs get compiled to web assembly binary. This solves the code migration problem by making the resultant programs portable and wrapped up using APIs. The portability is achieved by packaging the programs as containers or setting up compatible virtual machines on top of the incompatible execution environments.However, solution two does not reduce the number of technologies being used simultaneously.

Solution two is majorly implemented through the use of container technologies like Docker. There have been propositions and early implementations of using web assembly containers instead of mainstream solutions like Docker. This is because web assembly containers are comparably smaller in size and the corresponding wasm runtimes also have a small memory occupancy.

But if we look at the comparison objectively, Docker has the same properties as wasm functionality-wise. The true strength of wasm lies in the implementation of a universal host system API ; WASI (Web Assembly System Interface).

With WASI, we can package for any host system, be it an operating system, an App, a new kind of AGI... As long as the systems API is loosely defined. This means that WASI will be relevant probably many years to come even as new tech streams in... Just like how POSIX remained relevant for a long time. WASI is just like POSIX, but it is meant for more than just operating systems.

It is better to choose tools whose design and purpose will last for a long time... Hopefully.

## Problem Definition

Human Problem :

Developers find it hard to execute wasm containers on embedded devices at near native speed.

Developers treat development like a craft. A developer has to master their trusted technologies before moving on to solve creative problems. It is like taming a couple of trusted swords before moving on to war. In the path of building largescale IoT solutions, the trusted swords can be : Rust, Riscv and Wasm.

However, the wasm ecosystem is immature BUT quickly growing. Most of the available wasm tools are browser-focused, only a few projects focus on building wasm tools for the IOT space.

One tool that would be beneficial to have is a embedded system operating system that safely executes wasm containers on top of riscv CPUs at near-native speed.

At the moment, developers who need such a tool have resorted to two solutions ;

On one hand, developers embed the wasm runtimes in their bare-metal program executables. Let us call this method Method\_A. This method is not favorable because it means that the developer will not be able to run multiple programs on the same riscv CPU. Having an OS enables one to separate a process from the CPU thus enabling running of multiple programs. Furthermore, this process is unsuitable for modern IOT development because it makes remote code modification impossible.

On another hand, developers resort to running the wasm runtime on top of another operating system like Tock or Linux. Let us call this method Method\_B. This method allows swapping of programs and remote code modification. You get to enjoy the capability security system of wasm and use the extensible WASI interface. However this method is unfavorable because the layers of abstraction sacrifice out the execution speed of the wasm programs. The wasm programs run ontop of a wasm runtime that runs ontop of an OS.

### Technical Problem

Running wasm containers on top of a runtime that runs ontop of an OS is slow because the execution experiences two kinds of latencies ; System-call-normalization latency and context-switching latency.

The System-call-normalization latency is caused by the normalization process. The normalization process is the mechanism of converting a WASI function call to have the same function signature as the host kernel’s system calls. Normalization happens when there is a difference in the function prototypes of the WASI API syscalls and the Native Syscalls.

System-call-normalization happens during the compilation of .wasm files to the corresponding native binary files. When an AOT compiler is used, the normalization latency does not affect the execution speed of the binary file at runtime. If a JIT compiler or an interpreter is used, the System-call-normalization latency will affect the execution speed of the binary file at runtime.

Context switching occurs when the wasm executable calls for a system call function via WASI. This is because the CPU stops executing the wasm runtime process and instead switches to the kernel process, just to execute a system call function.

### Solution

The solution to the mentioned human problem is to build an embedded operating system that runs wasm containers at near-native speed.

The solution to the mentioned technical problems is to eliminate or reduce the intensity of the two latencies responsible for impeding the execution speed of wasm containers.

#### Handling the System-call-normalization latency

The system-call-normalization latency can be completely eliminated by making the kernel export its native system calls using the prototypes used in the WASI API [3], this means that the webassebly runtime will not need to normalize functions.

#### Handling the Context-switching Latency

It is possible to completely eliminate the context switching expense by making the wasm modules part of the kernel code. Such that the CPU will only execute one process through out its up-time. But this solution requires the developer to compile the wasm modules together with the kernel code. This means that whenever the wasm app gets updated, the developer has to recompile the entire kernel. This is highly unmaintainable. This solution is suitable when performance is more important than maintainability.

Completely eliminating context switching comes at the cost of maintainability, for this reason, it becomes more viable to focus on reducing the cost of context switching instead of ompletely eliminating it.

Context switching from a user-level process to a kernel-level process is more expensive than context switching from a kernel-level process to another kernel-level process. This is because switching from user-level process to a kernel-level process requires more overhead : The CPU has to switch modes, re-map the Memory Management page tables and switch the execution stack. On the other hand, switching from one kernel-level process to another kernel-level does not require the CPU to switch modes and re-map the page tables.

#### Chosen Solution

Based on the above discussion on the different methods of solving the latencies, the project went with building a kernel that :

1. Has the wasm runtime embedded in it, as part of the kernel code
2. Uses an Ahead-of-time compiler to compile the wasm modules
3. Exports its system calls using WASI function prototypes
4. Runs the loaded wasm containers in kernel mode.

### Objectives

For the project to be complete, the following research and implementation objectives had to be completed.

#### Research Objectives

1. Understand the different kernel designs and implementations
2. Understand the implementation aspects of wasm containers and wasm runtimes
3. Understand the IOT development infrastructure and figure out the problems faced in IOT development.
4. Understand the Docker technology and its downsides
5. Understand the relevance and usage of the Riscv CPU.
6. Research on how to integrate WASI system calls to the kernel

#### System Objectives

1. Build a minimal kernel that supports user programs
2. Extend the minimal kernel to embed a web assembly runtime
3. Modify the extended kernel to export its system calls using WASI function prototypes

### Project Justification

Web Assembly programs execute slowly in embedded devices when compared to native programs. Native programs have good performance but with poor portability and security control. This project aims to implement an OS that makes web assembly programs run at near-native speed in embedded devices while still taking advantage of the capability security system of wasm.

The resultant operating system will be a proof of concept that wasm-based operating systems will be the default operating systems in devices used in the IOT infrastructure. The OS acts as a proof of concept of the wasmachine proposed in the paper [4] with the aim achieving an isomorphic IOT architecture [2].

### Scope

The resultant kernel will only run on top of a Qemu-emulated Riscv CPU. The kernel will not be able to run on other ISAs.

The resultant kernel will not have complicated but necessary features such as networking capabilities or sound and graphics. The kernel will only be limited to the functionalities specified in the System Analysis Chapter.

# Literature Review

The literature review focuses on the technologies used withing the IOT infrastructure and how they fit in the project. The conclusion section of the literature review outlines the summary of the literature review and how it affected the design and implementation of the kernel.

## The IOT development structure

IoT development requires a range of software technologies, from mobile development to cloud and analytics. The common end-to-end architecture for IoT systems consists of edge devices, gateways, cloud, and applications.

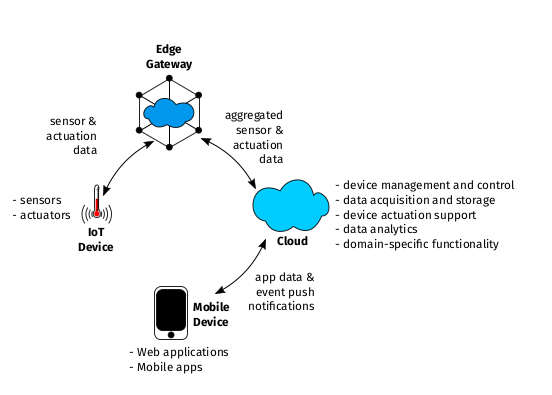


Figure 4 IOT architecture

IoT systems encompass embedded development, meaning that at the edge, the developer has to deal with a resource constrained environment. Embedded devices typically have low processing power, low memory and limited energy supply.

The EdgeGateway connects the edge devices to a local network or the internet and consequently a cloud platform. Edge devices or gateways are intermediary devices located closer to the devices/things, often at the edge of the network. They collect, preprocess, and filter data from connected devices, reducing latency and network traffic by performing local data processing and analysis. Edge devices may also provide additional functionalities like protocol translation and security features.

Cloud services form the backend infrastructure that stores, processes, and analyzes the data collected from IoT devices. Cloud platforms offer scalability, data storage, computing resources, and various services like data analytics, machine learning, and application development. They enable centralized management, real-time monitoring, and remote control of connected devices.

The Web and Mobile Apps provide a way to interact with both the cloud and edge devices, they provide a form of user-interface to the backend. IoT app development is dominated by mobile and web development frameworks more than desktop frameworks like Tauri.

Under this architecture, the backend services are packaged as independent micro-services; this helps in making the backend software more modular. The micro-services get wrapped as containers. A popular container choice is Docker. To mange the containers, developers employ a container orchestration software such as Kurbenetes.

The bigger dream of IOT development is to build a programmable world, ubiquitous technology. To achieve this vision, fog computing and edge computing was introduced. These two technologies have no clear definition, they roughly recommend that computing tasks should be transferable across many nodes. The main point being that the cloud should not be the only place where core processes are executed. This proposition is achievable because the processing power, storage and energy usage found in embedded environment has improved; we now have advanced CPU chips and high-volume storage components in the embedded chips.

The challenge that arises is that the programming languages and tools used to write and execute the data processing applications found the cloud might be not be supported by the runtime found in the edge device. As a result the software portability required by the fog computing proposal becomes hard to implement. As mentioned in the report introdution chapter, the solution to this is to : define communication standards, API standards and use an isomorphic wrapper(container).

## Web Assembly

Web assembly is an intermediate language for a virtual stack CPU. This intermediate language can be represented in assembly form or binary form.

Abstraction is an important concept in Computer Science, if a problem has too many details or multiple implementations... Just abstract it in simple wrappers and let the dirty implementations stay under that wrapper . For example, in the past, there were so many Operating systems cropping up. Every kernel developer came up with unique ways of implementing system calls. To simplify things, developers agreed to create a common System call abstraction, POSIX. Under this abstraction, they defined system call function signatures. From there on, developers had to use those agreed upon signatures but underneath those prototypes, they had the freedom to implement the functions their own way.

Back to webassembly. Webassembly employs the same abstraction technique. It abstracts an execution environment. Core webassembly abstracts a CPU while (Wasm+WASI) abstracts a CPU + the runtime running ontop of that CPU.

Being that webassembly abstracts CPUs, the wasm bytecode can be compiled to suit specific CPU ISAs. This makes wasm bytecode to be Architecture-independet.

Using the WASI API, web assembly abstracts any underlying system, be it an operating system or an ethereum system. As long as you use the WASI prototypes in your wasm bytecode, that code can be compiled for that specific implementation of the target system. This makes wasm platform-independent.

All high level programming languages need to get compiled to their target architecture. Using the LLVM compiler, a developer can compile any high level language to target the wasm virtual machine. This will in turn generate a .wat or .wasm file. A .wat file stands for ((webAssembly Text ). This .wat file contains human readable bytecode. The .wasm file contains machine code meant for the wasm virtual CPU.

The webassembly file is known to me compact and memory-efficient. This is greatly attributed to the fact that the bytecode targets a stack-based system thereby reducing the number of arguments required for each assembly instruction