Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 480b

Systems Group, Department of Computer Science, ETH Zurich

Exploring the use of WebAssembly for isolating Functions in Dandelion

by

Leon Thomm

Supervised by

Prof. Ana Klimovic, Tom Kuchler

February 15, 2024

Acknowledgements

I want to express my gratitude towards my supervisors Prof. Ana Klimovic and Tom Kuchler for fully involving me in the scientific process from day 1. You gave me the freedom I wanted, and the support I needed. Thanks Ana for your cheerful attitude, thanks Tom for your dedication. I knew from the beginning that I was lucky with this opportunity, and my opinion has not changed.

Further credit goes to all my closest friends and companions this past semester, for adding a constant amount of fun to anything, for answering my nerding with more nerding, and for checking in when I forget to talk to people. Thank you Eaint, Yannick, Frederick, Jan, Sandro, Rebecca, Jakob, Lukas, Emil, Ricardo and Noel.

Contents

Abstract **iv**

1 Introduction **1**

2 Background **2**

2.1 Sandboxing 2

2.1.1 Sandboxing Paradigms 2

2.1.2 Memory Safety 2

2.2 WebAssembly 2

2.2.1 Verification of WebAssembly 4

2.3 Serverless Cloud Computing and Dandelion 5

2.3.1 The Dandelion Cloud Platform 6

3 Design & Implementation **7**

3.1 Dandelion Concepts 7

3.2 Wasm 8

3.2.1 Embedding the SDK data in Wasm Memory 8

3.3 Wasmtime Backend 11

3.3.1 Safety Discussion 12

3.3.2 Scalability Discussion 12

3.4 rWasm Backend 12

3.4.1 Memory Optimizations 14

4 Evaluation **16**

4.1 Latency Under Varying Load 17

4.1.1 General Purpose Hardware (AWS) 17

4.1.2 Morello 18

4.2 Unloaded Latency 18

4.3 Peak Throughput Under Varying Payload 20

5 Related Work **21**

6 Future Work **22**

7 Conclusion **23**

Bibliography **24**

Appendix A - Notes on Experiments 26

Abstract

Untrusted code execution is the process of running software that cannot be assumed to execute correctly (bug-freedom) or with good intentions (malware-freedom). Today, untrusted code execution is found in various places, from a web-browser safely running websites on a mobile phone, to a large cloud computing provider running customer workloads on managed hardware and infrastructure. From an operating system perspective, we talk about *sandboxing* or *isolation* of user code. Computers typically solve this through *virtualization*, which one can view as the most uniquely characterizing ability of computers in general. Various different approaches exist to virtualize workloads that cannot be fully trusted, with varying levels of distrust. Sandboxing fully untrusted code requires precise control over the capabilities of the untrusted workload (can it access the network? the file system? can it make syscalls?), minimizing the attack surface for malicious code. This is commonly achieved through all kinds of *virtual machines*. Unfortunately, VMs often pose a tradeoff between security and efficiency. Running fully untrusted workloads inside a virtualized operating system may yield good security and generality, but typically comes at high cost due to inefficient resource utilization.

This work explores the use of WebAssembly to implement extremely lightweigt, general purpose sandboxing, by implementing WebAssembly execution engines for the Dandelion cloud platform. WebAssembly is a binary instruction format supported as compiler target by many programming language compilers today. It aims to allow execution in a highly controlled environment with little performance penalty compared to native machine code. We show two different approaches of integrating WebAssembly into a server-side environment like Dandelion. We discuss the tradeoffs between them, and quantify them in a comprehensive evaluation. We compare the approaches against each other, as well as against other lightweight VM-less sandboxing mechanisms in Dandelion. Within two different testing setups, under varying computational demand, we find that the two Wasm approaches complement each other well. In each of our tests, at least one of the Wasm approaches performs either on-par with, and sometimes significantly better than the fastest alternative. Furthermore, both Wasm approaches are highly independent of the underlying hardware. These findings illustrate the potential for WebAssembly as a general-purpose software sandboxing technology. Finally, some further thoughts and possible directions for future research are discussed.

1 Introduction

WebAssembly (*WebAssembly* n.d.), abbreviated Wasm, is a binary instruction format designed for a low-level virtual machine to run code at near-native speed with strong software sandboxing guarantees. While originally developed for web browsers, these guarantees have motivated integration of Wasm into various server-side systems, including FaaS and edge computing platforms (Varda, 2018; Fastly, 2023; Gadepalli, Peach, *et al.*, 2019), to implement lightweight untrusted code execution.

This work documents the development of multiple Wasm backends for Dandelion, a cloud platform, part of a research project at ETH Zurich, that rethinks the serverless computing programming model and execution system design by strictly separating compute and I/O. This allows for a variety of sandboxing mechanisms beyond traditional VMs to enforce isolation.

The sensible approach of integrating a lightweight Wasm VM is discussed first. However, this approach comes with some drawbacks, both from a security and, as we will see in the evaluation, efficiency standpoint. These drawbacks are highlighted and an alternative approach is implemented and discussed. This second approach will leverage rWasm (Bosamiya, Lim, and Parno, 2022), a Wasm-to-safe-Rust compiler. The Rust compiler (`rustc`) informally guarantees memory safety of Rust code that does not make use of any `unsafe` Rust features. Using rWasm and `rustc` we can compile any Wasm binary to a native binary whose execution is memory safe for the host.

Apart from the security aspects, Wasm is at its core a unified compiler target with broad compiler support today. This work opens the Dandelion platform for user code written in any programming language that compiles to Wasm.

2 Background

2.1 Sandboxing

In the broadest sense, this work is about untrusted code execution. The process of isolating untrusted workloads to prevent malicious or faulty code to cause any damage is called sandboxing. Major applications range from cloud computing, to email clients, to software add-ons and plug-ins[1], to mobile apps, development environments, IoT devices, and even web-browsers. Notice that while one of the key responsibilities of any operating system is *isolation*, it usually does not provide strong sandboxing guarantees for running arbitrary untrusted code.

2.1.1 Sandboxing Paradigms

Wahbe *et al.*, 1993 differentiates *software fault isolation* and *harware fault isolation*, though we will avoid these terms due to slightly ambiguous usage of them today. We consider untrusted code execution from a perspective of generality, where different approaches require varying levels of hardware specialization.

2.1.2 Memory Safety

The techniques applied in this work put a particular focus on *memory safety*, which is a class of memory usage-related safety issues, such as buffer overflows, dangling pointers, use-after-free and double-free errors, etc. We consider a very simple model: a trusted *host* app executes an untrusted *guest* app. The host needs to ensure the guest cannot access any memory not specifically intended for it, no matter what instructions the guest executes. Crucially, this means we do not care about memory safety issues *within* the guest, we only care about shielding the host from the guest. The host cannot make any assumptions about the guest's program behavior beyond what the hardware allows.

Security concerns through side-channel attacks are not specifically part of this work but are addressed to some extent in the overall Dandelion project.

2.2 WebAssembly

WebAssembly (Wasm) is a binary-code format implementing the ISA of an abstract machine (an imaginary computer), designed for fast and sandboxed execution of untrusted code. As such, Wasm mainly serves as a *compiler target*, with board support from source languages including C, C++, Rust, Go, Java, Kotlin, Haskell,

[1]or Add-Ins if you're Microsoft

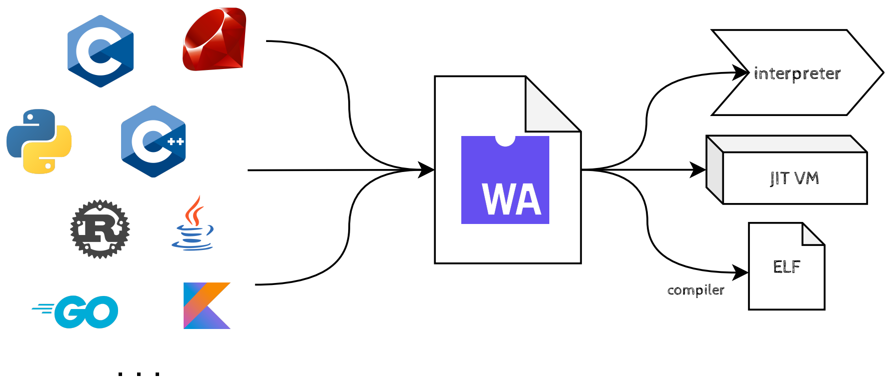


Figure 2.1. Overview of the Wasm compilation workflow and the most important parts of the Wasm machine. Notice the above model implements a Harvard architecture, strictly separating data and instruction memory.

OCaml, Ruby, Python, and more. The machine described by Wasm is rather minimal and designed for lightweight virtual implementation on top of all common architectures[2].

Originally developed for secure and near-native performant code execution in web browsers, today Wasm runs in many places outside of browsers, such as Cloudflare (Varda, 2018), Fastly (Fastly, 2023), and the edge (Gadepalli, Peach, *et al.*, 2019).[3] Figure 2.1 illustrates the typical workflows when using Wasm as a compiler target.

Interfaces. The only way for a Wasm module to communicate with the outside world is through a narrow import/export interface. A Wasm module can export functions that the host can invoke, and import functions provided by the host which the module calls. E.g., in order to print text to stdout, the Wasm module needs to import a function like `fd_write` from the host which would allow it to pass data to that file descriptor.

Memory. A Wasm module hosts one memory segment declared in the Wasm binary. This memory segment is consecutive and a multiple of 64KiB (the Wasm page size).

Structured Control Flow. Control flow is fully structured (no `goto`, no `longjmp`, etc.; only `return` and jumps to enclosing loops). This and the absence of undefined behavior[4] greatly simplifies reasoning and verification processes for Wasm modules.

The safety of the host environment using Wasm relies on two factors:

1. The design of the host API: The host must ensure the exposed interface (the Wasm module imports) is safe and cannot be misused.

2. The integrity of the Wasm implementation: If the implementation of Wasm used by the host (e.g. the Wasm VM) is faulty, security might be compromised.

Constructing and verifying safe APIs is orthogonal to this work. We will not provide any interface to the guest module that could be misused, so we will instead focus on safety of the implementation. Since Wasm describes a *virtual* machine, in order to execute Wasm there must be something implementing it on a real system. There are broadly three approaches:

1. interpreters (e.g. wasm3 (Shymanskyy, 2023), wasmi (Technologies, 2023), WasmRef-Isabelle (Watt *et al.*, 2023))

2. JIT-compilers (e.g. Wasmtime (Alliance, 2023b), WAMR (Alliance, 2023a)), often with partial AOT support

[2] I heard some people are actually discussing hardware support, but this is probably not around the corner.
[3] If you are already confused about the naming: There is a joke that WebAssembly is neither web, nor assembly.
[4] Wasm does allow *non-determinism* in a small number of well-defined cases, but no undefined behavior.

3. full AOT-compilers (e.g. wasm2c (Group, 2023), rWasm (Bosamiya, Lim, and Parno, 2022))

JIT-compilers are the most common approach, they combine fast development with good security. Full AOT-compilers translate a Wasm module to a native binary either directly or through another programming language, removing the need for a VM/runtime. For example, Firefox's RLBox framework sandboxes modules in Firefox by compiling them to Wasm and then transpiling them to C again using wasm2c (Group, 2023), avoiding the overhead of traditional process-based decomposition. (Holley, 2021)

An example of a Wasm binary is given below, in the human-readable WebAssembly Text Format:

```
(module
  (type (;1;) (func (result i32)))              ;; function type definitions
  (type (;2;) (func (param i32 i32) (result i32)))
  (import "env" "memory" (memory (;0;) 2))      ;; imported memory
  (func (;1;) (type 1) (result i32)             ;; function definitions
    (local i32 i32 i32 i32 i32 i64 i64 i64 ...)
    global.get 0
    i32.const 48
    i32.sub
    local.tee 0
    global.set 0
    i32.const -1
    local.set 1
    block  ;; label = @1
      call 7
      i32.const 1
      i32.ne
      br_if 0 (;@1;)
  ...
  (global (;0;) (mut i32) (i32.const 66608))    ;; global variables
  (global (;1;) i32 (i32.const 1024))
  (global (;2;) i32 (i32.const 0))
  (global (;3;) i32 (i32.const 1024))
  ...
  (export "mat_mul" (func 1))                   ;; exports
  (export "_start" (func 2))
  (export "__dandelion_system_data" (global 3))
  ...
  (export "__data_end" (global 7))
  (export "__stack_low" (global 8))
  (export "__stack_high" (global 9))
  (export "__heap_base" (global 11))
  (export "__heap_end" (global 12))
  ;; data segment reserving space for the system_data struct at offset 1024 (global 3)
  (data (;0;) (i32.const 1024) "\00\01\00\00\ff\ff\ff\ff\00\00\00\00\ff\ff\ff\ff\00\00\00\00\ff\ff\ff\
    ff\00\00\00\00\00\00\00\00\00\00\00\00"))
```

Listing 2.1. Example WebAssembly, we will come back to it later.

2.2.1 Verification of WebAssembly

This section provides an overview of the main efforts towards verification of correctness and safety of Wasm modules as of December 2023. Formally verifying semantic correctness of compilers remains highly challenging. Nevertheless, such efforts for Wasm compilers are underway, most notably

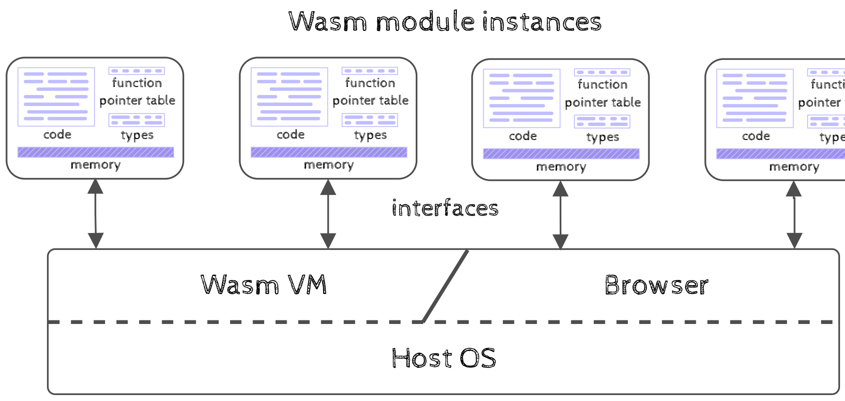


Figure 2.2

- **Wasm interpreter WasmRef-Isabelle** (Watt *et al.*, 2023): A formally verified correct interpreter, which is now used for fuzz testing in the Wasmtime (Alliance, 2023b) continuous integration (CI). The performance is claimed to be comparable to a debug build of the wasmi (Technologies, 2023) interpreter, but significantly higher than the previous OCaml-based reference implementation.

- **Wasm JIT Cranelift**: Efforts are underway to provide partial verification of the Cranelift code lowering routines. Cranelift is a Wasm compiler which several industrial Wasm VMs including Wasmtime (Alliance, 2023b) are based on.

A verified correct implementation of Wasm is safe. However, many safety constraints can effectively be enforced without verifying correctness.

- vWasm (Bosamiya, Lim, and Parno, 2022): A Wasm-to-x86 compiler whose output is formally proven to be safe for the host to execute.

- rWasm (Bosamiya, Lim, and Parno, 2022): rWasm - from the same paper as vWasm - is a Wasm-to-Rust transpiler which leverages the safety guarantees of the Rust compiler to informally prove the resulting binary safe to execute for the host.

- Iris-Wasm (Rao *et al.*, 2023): A verifier based on the Iris (JUNG *et al.*, 2018) framework which proves safety of *interaction between* Wasm modules.

- MSWasm (Michael *et al.*, 2023): An extension of the Wasm standard enforcing module-internal memory safety[5], by extending the concept of memory by *sections* and *handles* to these sections.

rWasm has work-in-progress MSWasm support, which allows for Ahead-of-Time compiling MSWasm binaries to a memory safe native binary (on any Rust target platform) whose internal safety cannot be exploited.

2.3 Serverless Cloud Computing and Dandelion

[5]Notice that module-internal unsafety can still lead to vulnerabilities if unsafe APIs are exploited. For example (Lehmann, Kinder, and Pradel, 2020) demonstrated a cross-site-scripting attack based on module-internal buffer overflows. Incorrect code written in an unsafe language like C can perfectly be compiled to both Wasm and MSWasm, but it cannot be *exploited* in MSWasm as the standard requires the implementation to trap as soon as a violation of a memory safety enforcing rule occurs at runtime.

Cloud computing has continuously evolved over the past two decades. From early Infrastructure-as-a-Service models, to Platform-as-a-Service, to Software-as-a-Service, to modern, highly decentralized computing paradigms such as edge computing.

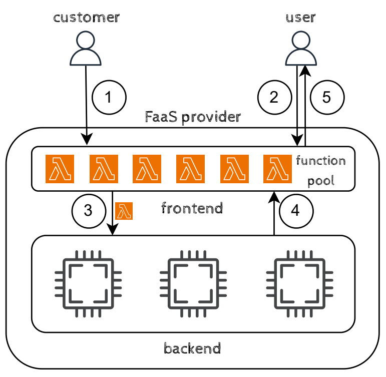


Figure 2.3

Today, the paradigm of *serverless cloud computing* (short *serverless*) is becoming increasingly dominant (explored in "What Serverless Computing Is and Should Become: The Next Phase of Cloud Computing" 2021). Serverless abstracts away the notion of a server for customers. This mainly combines Backend-as-a-Service (*BaaS*; databases, object storage, etc.) and Function-as-a-Service (*FaaS*; server compute). Other than in on-premise services, in serverless computing the cloud provider dynamically manages the whole resource allocation without knowledge by the customer workload. This increases flexibility of resource allocation and simplifies the model for the customer. But it also poses some challenges regarding efficiency and safety.

In FaaS, an application consists of a composition of short-lived "functions" (usually executable binaries). In order to allow seamless transition from on-premise services to serverless, FaaS platforms often fall back to VMs running the user functions. However, the very nature of a short-lived user process gives rise to efficiency concerns when every invocation of a user function leads to setting up (or re-using) a virtual machine.

2.3.1 The Dandelion Cloud Platform

Dandelion is a FaaS platform that rethinks several aspects of serverless cloud computing. It attempts to radically improve resource utilization by clearly separating compute and I/O. A user function performs pure compute, and all I/O (web requests, database access, etc.) is performed by configurable, trusted, platform-native I/O functions. I/O functions and user functions can then be composed into a directed graph representing the data flow of the application, which allows for fine-grained resource optimization.

Figure 2.3 illustrates the basics of a FaaS system. The workflow indicated by the numbers is:

1. The user/client registers a function. The function is usually a binary compiled against some platform SDK.
2. A user/client invokes the function.
3. Once a backend is available, the frontend sends the invoked function with inputs to the backend.
4. After the backend has completed the execution, status and results are returned.
5. The frontend sends status and results back to the user.

In Dandelion, a backend essentially consists of an *engine* and *context* type. The radical simplicity of Dandelion's approach to user functions allows us to create multiple such backends, simply by implementing aforementioned components. This in turn also allows for more effective heterogeneous hardware support, by adding new backends targeting specific hardware (such as a GPU). The following backends exist already:

- CHERI: The CHERI architecture extends existing ISAs by *capabilities* allowing for precise control over the memory access of running applications by the host. While being fast and secure, this requires specialized hardware. Dandelion has a CHERI-backend which leverages capabilities to enforce memory sandboxing of user functions.
- MMU: Dandelion has a backend leveraging Linux process isolation to enforce memory isolation.

3 Design & Implementation

This work is about adding Wasm support to Dandelion, leveraging the security and efficiency aspects of Wasm to implement lightweight isolation. In this model, the user uploads a `.wasm` binary as "function", which will run on one of the Wasm backends.

Implementation Safety, No Correctness, No API. We will focus on safety of the implementation, constructing and verifying safe interfaces is orthogonal to this work. The host (Dandelion) will not provide any interface, i.e. the Wasm modules cannot import any functions and are pure compute.

Using rWasm and Wasmtime. We will use rWasm (Bosamiya, Lim, and Parno, 2022) and Wasmtime (Alliance, 2023b) to implement two different backends running Wasm.

3.1 Dandelion Concepts

We will now take a closer look at how Dandelion implements FaaS, and then motivate how Wasm support can be integrated. In this context, only invocations of already registered functions are interesting to us, since function registration does not depend on the backend.

Frontend and Dispatchers. Figure 3.1 gives an overview over the different components relevant to function invocation in Dandelion. A user request is received by an HTTP frontend and passed to the *dispatcher* which manages available compute resources and delegates requests to *engines*.

Engines and Drivers. Engines perform the actual compute. They receive user functions, prepare function inputs, run the functions, and return results to the dispatcher. Engines must satisfy an internal interface, otherwise little restrictions are put on their implementation. Different engines can make use of different hardware, and apply different sandboxing approaches. An engine instance is bound to a CPU core and can be reused. An engine-specific *driver* is responsible for parsing a user function from the file system that can run on the respective engine type, and setting up the engine itself.

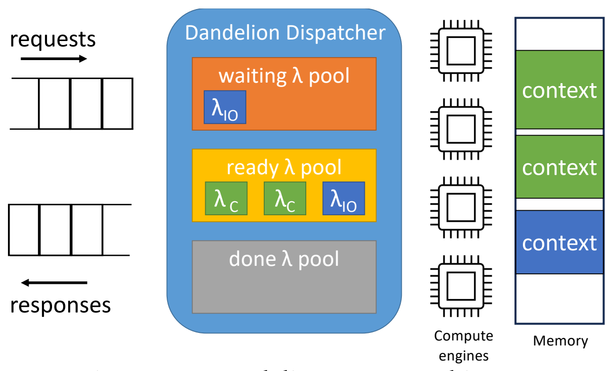


Figure 3.1. Dandelion system architecture.

Contexts. Attached to an engine type is a respective *context type*. A context is an address space of a function. It encapsulates the memory the function can access as one consecutive chunk in Dandelion's address space. Function input buffers, output buffers, as well as the function's allocated heap memory all live in the context. It's the main responsibility of an engine to ensure that a user function never reads or writes any memory which is outside its context.

In this work, when talking about a *backend*, we mean one particular driver-engine-context triple, and possibly further backend-specific compilation workflows.

User Functions. A user function is a binary and attached metadata (the number of inputs, outputs, etc.). The binary has one entry point (the typical C `_start` function) and does not interface with Dandelion or the OS (no syscalls). This restriction is fundamental to Dandelion's approach, user functions are pure compute, all I/O and high-level data flow is described by function *compositions*.

Inputs, Outputs, and the Dandelion SDK. Dandelion defines a data structure holding input buffers, output buffers, and metadata about a function invocation. The input buffers are populated before calling the function, and outputs are extracted after it returned. The user code can access inputs and write outputs through a narrow interface provided by the *SDK* which the user code is compiled against (so it ends up inside the user function binary), and which handles this data structure.

Memory Allocation. Almost any software code today needs some form of dynamic heap management. Recall that function memory is encapsulated in a context. The data structure handed to a user function contains offsets indicating which part of the function's context is free (from code, data, populated input buffers, etc.) and can be used as dynamic heap by the user code. The SDK has a simple heap allocator built-in that makes use of this space.

Legacy applications with `dlibc`. For legacy code compatibility, Dandelion also provides a custom `libc` implementation called `dlibc`. `libc` is a common set of cross-platform interfaces that provide resource management and interaction with the OS. Since Dandelion intentionally does not provide a host OS interface to user functions, most of the functionality of `libc` is not allowed to be used. In those cases, `dlibc` will simply return an error to the user code. For other interfaces, such as the file system, `dlibc` provides a virtualized in-memory implementation. For memory management `dlibc` builds on the memory allocator provided by the SDK. Again, notice that this library will end up within the user function binary, and is therefore not trusted.

3.2 Wasm

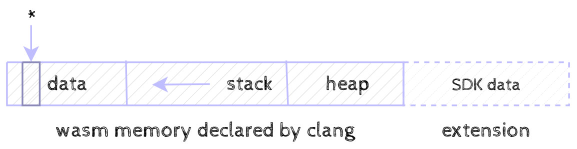
A Dandelion Wasm backend must safely execute Wasm code on the machine Dandelion runs on, which is currently `arm` (with optional `CHERI` extension) or `x86_64`, both running Linux.

Executing Wasm $\triangleq$ simulating a module on the machine described by the Wasm standard.

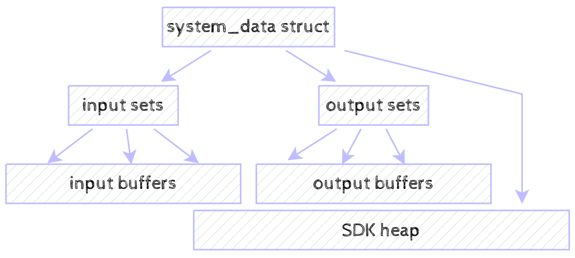
3.2.1 Embedding the SDK data in Wasm Memory

SDK data structure. Figure 3.2 (b) provides an illustration of the data structure that is passed to a user function with populated input buffers, and from which outputs will be extracted after termination. This data must be passed to the Wasm module.

Wasm Memory. The key limitation when passing any data to a Wasm module is that Wasm only has four basic numeric types, so we cannot simply accept the data as a function argument in the entry point. Instead, we need to lay out the data in the Wasm module's memory manually, and then let it know where it is. This, in turn, means the module cannot assume zeroed memory (which would be the default), because we're messing with it prior to



(a) Wasm memory layout.



(b) The data exchanged between Dandelion and the user function (simplified). Each arrow denotes a pointer and associated length value.

Figure 3.2

starting execution of the Wasm code. In Wasm jargon: the Wasm memory is *imported*.[1] For Wasm, our context will essentially hold the Wasm memory of the invoked Wasm binary. The default memory layout of a Wasm binary generated by clang is illustrated in figure 3.2 (a), with the following regions:

- `data` region: holds statically initialized data, similar to `.data` in ELF.
- `stack` region: the Wasm stack, which - since Wasm describes a stack-based machine - holds operands for operations, as well as their results.
- `heap` region: Wasm heap memory, available through instructions such as `i32.load <offset>` (loading memory from the heap onto the stack) and `i32.store <offset>` (analogous opposite).

We can now identify the main challenges for embedding SDK data in the Wasm memory.

1. Where should we put the SDK data, and how do we let the SDK (compiled in the Wasm module) know where it is?
2. Since the SDK data passed heavily relies on pointers, how do we translate those from the address space of Dandelion to the address space of the Wasm module?

Handling Dynamically Sized Buffers. For the first point, the most obvious approach would be to reserve an upper bound of memory for the whole SDK data already in the binary. While theoretically simple, this approach has limits. E.g. in order to change the amount of memory available to the SDK, the user function needs to be re-compiled from scratch. To avoid this, we instead artificially extend the memory we provide to the Wasm module to be larger than what the Wasm binary originally declared, effectively extending the Wasm heap as illustrated in figure 3.2 (a). This is where most of the data (such as buffers and the SDK heap region) go. It gives us precise control over the amount of heap memory available to the function without the need to re-compile the user code.
Communicating `system_data` Location. Now we still need to communicate to the module where exactly it can find this data. Notice for the `system_data` struct we actually can pre-allocate space in the Wasm memory, because it is of fixed size! If `system_data` is expected at a known location in the Wasm memory (see the * in figure 3.2 (a)), we can write the struct there so the Wasm code already knows where to find it, while all dynamically sized buffers referenced in `system_data` are placed in the heap extension. This was technically accomplished by declaring an `extern system_data` symbol in the SDK C code, and compiling user function code with the `--export-undefined` flag, which makes sure the location of this uninitialized struct is exposed in the Wasm

[1] Imported memory is not owned by the module, and hence isn't assumed to be zeroed. This concept can be used to share memory between multiple Wasm module instances.

binary. In the previous Wasm example of listing 2.1 you can see how this looks like in Wasm. Notice we could also supply a pointer to the `system_data` struct in the entry when invoking the user function, but this way we don't need to declare our custom entry point but can rely on the standard C `_start()` function.

Moving data between Address Spaces. For the second issue, we can identify two immediate approaches:

1. Manually changing the pointers (by subtracting the address of the Wasm memory in Dandelion's address space) when handing it over to the user function, and doing the reverse after the function terminated.

2. Constructing the SDK data within the address space of the user function in the first place.

Offsetting pointers when moving data between address spaces would be a general approach. However, implementing this can be tricky as it requires very careful usage of the obtained translated addresses, and changes to the data structure can easily lead to critical memory issues. contexts elegantly solve this problem. One way to look at the concept of a context is as a wrapper around the memory / address space of the untrusted function. The `Context` type provides an interface

- `fn read(offset, read_buf)`

- `fn write(offset, write_buf)`

- `fn get_free_space(size, align) -> Result<offset>`

To write for example the `input_sets` buffer into a context, we can use `get_free_space()` to allocate a suitable buffer in the context and obtain an offset to it, and then we can use the obtained offset in `system_data` as pointer. The opposite process retrieves output buffers from the SDK data in the context, after the function execution finished. This way we never have to mutate pointers and only operate within the function memory, and we can easily ensure that we never accidentally write past its boundaries.

Different Address Space Widths. In contrast to Wasm, which operates on 32 bit addresses, Dandelion operates on the pointer size native to the machine, which on most modern machines is 64 bit. Since the SDK is part of the user function and gets compiled to Wasm, it will expect pointer sizes of 32 bit length. The context interface mentioned above gives us offsets of type `usize`, which translates to `size_t` in C, which might be 64 bit in Dandelion. To ensure Dandelion and the SDK in a particular user function agree on a common pointer size, we rewrote parts of the process constructing the SDK data in a context to generically accept a custom word size.

```
pub mod _32_bit { pub type DandelionSystemData = super::DandelionSystemData<u32, u32>; }
pub mod _64_bit { pub type DandelionSystemData = super::DandelionSystemData<u64, u64>; }

pub struct DandelionSystemData<PtrT: SizedIntTrait, SizeT: SizedIntTrait>
{
    pub exit_code: i32,
    pub heap_begin: PtrT, pub heap_len: SizeT,
    pub input_sets: PtrT, pub input_sets_len: SizeT,
    ...
}
```

Listing 3.1. `system_data` implementation generic on the address width (slightly simplified). Even if Dandelion itself is running on a 64 bit architecture, a Dandelion backend preparing data for a Wasm function operating in 32 bit can simply select `_32_bit::DandelionSystemData`; all the helper functions operating on `DandelionSystemData` can now work with any address width.

3.3 Wasmtime Backend

Dandelion's Wasmtime backend integrates a Wasm VM (Wasmtime in this case) to run the user function. Integrating Wasmtime was straightforward, since Wasmtime provides an excellent API to access exports (such as the `system_data` pointer) and manage memory. Layout information such as the `system_data` pointer is extracted when the function is first loaded (i.e. when it is invoked *cold*) from the file system.

```
// read function from file system
let wasm_module_content = std::fs::read(&function_path);
// parse binary in wasmtime
let precompiled_module = store.engine().precompile_module(&wasm_module_content);
// <-- snip instantiating the module once -->
// extract layout information
let system_data_struct_offset = instance.get_global(&mut store, "__dandelion_system_data");
let wasm_mem_min_pages = memory_ty.minimum() as usize;
let sdk_heap_base = wasm_mem_min_pages * 65536;
let total_mem_size = (wasm_mem_min_pages + SDK_HEAP_PAGES) * 65536;
```

Listing 3.2. High-level process of parsing a user function from a Wasm binary.

The process of invoking the function is similarly straightforward.

```
// instantiate module, with extended memory
let instance = wasmtime::Instance::new(store, &module, &[memory.into()]).unwrap();
// write the system data struct
memory.write(store.as_mut(), sysdata_offset, &sysdata_buffer);
// call entry point
let entry = instance.get_typed_func::<(),()>(store.as_mut(), "_start");
let ret = entry.call(store.as_mut(), ());
```

Listing 3.3. High-level process of instantiating a Wasm function in the Wasmtime backend.

Notice how we are calling `precompile_module()` when loading a function for the first time. This pre-compilation turns the Wasm binary into a representation of the data structure that Wasmtime uses internally to instantiate a module. Wasmtime supports pre-compiling in advance, which reduces the amount of work required when trying to invoke a function (which is always on the critical path). Based on this, I extended the Wasmtime backend to essentially implement two different versions, one which can take an arbitrary Wasm binary like shown above, and another one which expects an already pre-compiled module, with an associated pre-compilation workflow that can be performed ahead of time, when the function is registered.

The main advantages of the Wasmtime backend can be summarized as follows

- Simplicity: Easy to implement and flexible.
- Stability: Battle-tested Wasm VM.
- Development: Wasmtime is well documented, and Wasm extension proposals are usually first experimentally supported in Wasmtime.

However, the cutting-edge development of Wasmtime has some tradeoffs as well. Most notably efficiency and security.

3.3.1 Safety Discussion

Wasmtime is battle-tested and employs highly modern standards for security-critical software, such as being written in a memory-safe language and continuously getting fuzz-tested against a formally verified reference implementation. However, it also tries to be at the frontier of technological development around Wasm, which is not always compatible with the highest security standards. Critical security issues happen rarely, but they happen (*CVE-2023-26489.* 2023) and one might want to narrow down the trusted computing base even further.

3.3.2 Scalability Discussion

The Wasmtime backend has a flaw. As we will see later in the evaluation, while it performs very well on smaller machines, it scales poorly to large servers. Recall that Wasm is a 32 bit architecture. As such, the maximum size of a Wasm memory is $2^{32}/2^{30} = 4$ GiB. Load and store instructions refer to a memory location by providing a 32 bit offset from a 32 bit address, yielding effectively 33 bit accessing into an 8 GiB space. In order to enforce the memory isolation of a Wasm module implied by the Wasm standard, the Wasm implementation (the VM in this case) must ensure all memory access is in bounds, i.e. not beyond the actual size of the Wasm memory, and certainly not beyond 4 GiB. There are a few options to enforce this

- One can manually check every memory access in software. This is generally too slow.
- One can apply bit masking, which effectively applies a modulo to all memory access of the Wasm module. While this is fast, it results in module-internal memory corruption on illegal memory access, while the Wasm standard technically requires the VM to trap instead, preventing continued execution.

A third option, the one most VMs including Wasmtime[2] employ, uses the MMU to implicitly enforce in-bounds access. Here, all Wasm memory is embedded in a region of 8 GiB (everything the module can theoretically address) with the actual Wasm memory at the beginning, and the rest filled with protected *guard pages*. Any out-of-bounds memory access of the Wasm module will land in those guard pages and cause a processor exception, triggering a trap in the VM. This way, the VM doesn't need to manually check all memory access, but will be able to trap once an out-of-bounds memory access occurs.

Internal Fragmentation. The main issue of this approach is that the virtual memory footprint of a single Wasm module instance is now 8 GiB by default, most of which is never used. While this already limits the number of instances we can keep simultaneously[3], it also causes degradation of memory locality.

This is explored in Narayan, Garfinkel, Taram, *et al.*, 2023, and the authors conclude that these limitations are conceptually inherent to Wasm as an SFI technique, and proceed to propose hardware extensions aimed to solve these issues. We don't find this claim to be true for Wasm in general. As we will explore in the following section, there is another option to safeguard Wasm memory without the runtime overhead of manual checks, without the memory overhead of guard pages, without harware extensions and with much more scalable performance characteristics.

3.4 rWasm Backend

To address the shortcomings of the Wasmtime approach discussed above, we added another Wasm backend for Dandelion, based on rWasm. As we will see later, this backend not only scales extremely well, it also provides

[2]Documented at `https://docs.wasmtime.dev/contributing-architecture.html#linear-memory`.

[3]For example, on CPUs providing 48 address bits, this yields a theoretical maximum of $2^{48}/2^{33} = 2^{15} = 32$ K instances.

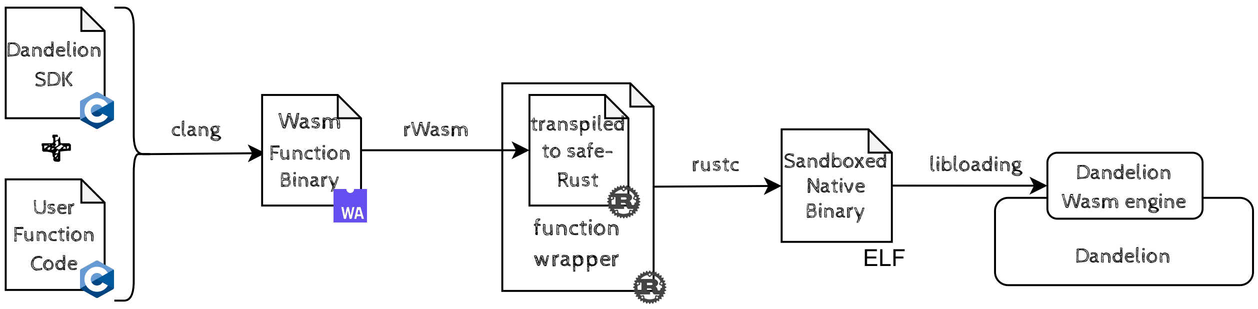


Figure 3.3. Wasm function compilation toolchain.

excellent security guarantees. Since Dandelion provides no functional interface for the user functions, the only way for a user function to cause harm is through memory unsafety. The Rust compiler informally guarantees memory safety of any program written in safe Rust only. For a more detailed discussion on the security aspects please refer to the original rWasm paper (Bosamiya, Lim, and Parno, 2022).

Fork. While rWasm is conceptually highly attractive for a simple execution environment such as Dandelion, it is also somewhat incomplete. Not all Wasm features are supported, the compiler is not bug-free[4], and the last commit on the main branch has been 1.5 years ago. Within this project, we created a fork of rWasm on GitHub, mainly adding new features and more fine-grained control over the compilation, and all mention of rWasm below refers to our fork. The changes are in the process of getting upstreamed into the original project.

Figure 3.3 provides an overview of the compilation workflow to turn user code into a runnable function on the Dandelion Wasm backend.

User Code → Wasm Function Binary. As before, the compilation from user source code to a Wasm binary is performed by the user. We are only using C as source language so far, but as soon as the SDK is ported, any language that compiles to Wasm can be used in a similar way. Using C, this step currently mainly involves a bunch of makefiles.

Wasm Binary → Sandboxed Native Binary. Turning the Wasm binary into a memory safe native binary involves two steps.

1. The Wasm binary is transpiled to Rust using rWasm. This generates a Rust crate which contains only safe Rust code.[5] One major extension I made to rWasm is allowing this generated code to not depend on a memory allocator. Since the user function gets no syscalls, it can also not dynamically request memory. It should use the SDK's memory allocator instead.

2. The crate generated by rWasm is wrapped in another crate. This wrapper is part of the trusted computing base and will be compiled to obtain the final function binary. It mainly serves the following purposes:

    - Introduce necessary unsafety: Ultimately, a user function written in Rust needs some `unsafe` code. For example, some symbols, such as the entry point and the `system_data` pointer, need to be exposed in a predictable way, without the compiler changing their names. In Rust, this is done using the `#[no_mangle]` macro. Now if two binaries are linked together with conflicting un-mangleable declarations, the result will be undefined.

    - Statically expose metadata about the transpiled Wasm module (such as memory size).

[4] Notice that this is not a security concern. rWasm only transpiles Wasm code to safe Rust code. It does not provide any correctness guarantees, and even if the code it generates is broken or doesn't correctly implement the same semantics as the Wasm consumed, the memory safety will be provided by the Rust compiler when compiling the generated Rust code.

[5] While the generated code doesn't make use of unsafe Rust, it still depends on things that do, such as the `core` crate. It is common practice for security critical software written in Rust to assume safety of these components.

- Embed SDK data and run the module: The Rust crate generated by rWasm defines some types which can be instantiated to effectively instantiate the original Wasm module in Rust. The entry point of the wrapper crate receives the SDK data from Dandelion, then instatiates the transpiled Wasm module, embeds the SDK data in its memory, calls the transpiled Wasm module's entry point, and returns the exit code to Dandelion. Since both the wrapper as well as Dandelion are written in Rust, this process is comparatively easy and safe to implement.

The above process is currently automated using some UNIX shell scripts.

Running the Binary. We now have a memory safe native binary which exposes Wasm layout metadata and an entry point which simply receives the SDK data and returns an exit code. The Wasm backend of Dandelion now only needs to dynamically load this binary and call into it. For this, we use the `libloading` Rust library, which wraps the native system loader (calling `dlopen()` in our case). Figure 3.4 shows how this is done in the Dandelion Wasm backend.

```
type WasmEntryPoint = fn(&mut [u8], usize) -> Option<i32>;

match unsafe { self.lib.get::<WasmEntryPoint>(b"run") } {
    Ok(entry_point) => {
        let ret = entry_point(&mut wasm_context.wasm_memory, self.sysdata_offset);
        return match ret {
            Some(_) => Ok(()),
            None => Err(DandelionError::EngineError),
        };
    }
}
```

Listing 3.4. High-level process of instantiating a Wasm function in the Wasm backend in Dandelion. `lib` is the handle to the dynamic library opened by the system loader via the `libloading` crate. Closing the library is elegantly handled by Rust's value dropping system. When no reference to `lib` remains, the library will automatically be closed.

3.4.1 Memory Optimizations

The first implementation yielded extremely bad performance. We are currently always committing 128 MB of heap memory to the user functions (everything not reserved by the original Wasm binary becomes SDK heap), and it turns out to be a bad idea trying to allocate and zero this much memory on the critical path, especially since most functions will never use all this memory.

The solution was to use `mmap()` to allocate the memory for the user functions. `mmap()` was originally introduced to map files in main memory, but since the `MAP_ANONYMOUS` mode was introduced (creating a mapping not backed by any file) `mmap()` is commonly used for general memory allocation today. Calling `mmap()` with `MAP_ANONYMOUS` guarantees two things: we own the amount of memory requested, and it will be zeroed *once we access it*. The OS actually doesn't need to allocate the memory requested when calling `mmap()`, but can do so on demand when accessing it, which is much cheaper.[6]

One pitfall in Rust is that calling `mmap()` directly bypasses Rust's own memory allocator. One of the nice things about Rust is its automatic memory management without garbage colleciton, by automatically freeing memory

[6]Allocating memory lazily like this is referred to as the operating system overcommitting memory. If functions would suddenly access many pages that they technically own but were never actually allocated, they could potentially cause an out-of-memory error in Dandelion, which is in general hard to prevent. Some work remains to be done in Dandelion to ensure that this does not compromise safety.

owned by values when these are *dropped* (go out of scope). When dropping e.g. a reference to the Wasm memory allocated with `mmap()`, Rust will panic since the memory allocator now tries to deallocate memory it has never allocated. The solution is refreshingly simple: create a smart pointer with customized drop behavior, calling `munmap()`.[7]

```
mod wasm_memory_allocation {
    pub struct MmapBox {
        ptr: *mut u8,
        size: usize,
    }
    impl MmapBox {
        pub fn new(size: usize) -> Result<MmapBox, DandelionError> {
            let addr = unsafe { mmap(
                0 as *mut _, size,
                libc::PROT_READ | libc::PROT_WRITE,
                libc::MAP_PRIVATE | libc::MAP_ANONYMOUS,
                -1, 0,
            ) };
            // <-- snip error handling and return -->
        }
    }
    // <-- snip deref semantics -->
    impl Drop for MmapBox {
        fn drop(&mut self) {
            unsafe { munmap(self.ptr as *mut _, self.size); }
        }
    }
}
```

[7]While creating contexts is very fast this way, unmapping memory causes a TLB shootdown. This could potentially be eliminated by shifting more of the memory management performed by the OS into Dandelion.

4 Evaluation

We will now compare the performance of the two Wasm backends against each other, as well as against alternatives. We use two hardware setups, explained below. In the experiments showed here, all functions get 128 MB of memory.

AWS setup. This includes two general purpose AWS EC2 machines, one acting as client and the other running Dandelion. Both machines are placed in the same AWS security group and networking effects should be negligible. We used identical EC2 `c6g.metal` machines with 64 vCPUs and 128GB RAM running Ubuntu.

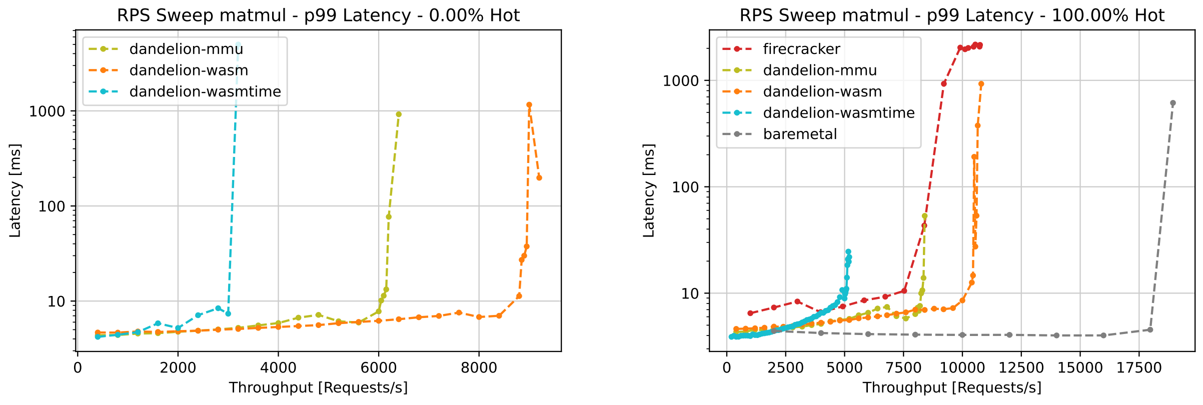
Morello. Dandelion features a CHERI backend which uses CHERI capabilities to enforce sandboxing. This requires hardware support and the only manufactured board currently implementing CHERI is ARM's Morello. We tested on a Morello board running a modified version of Linux.

Terminology. Most benchmarks here are based on a basic matrix multiplication written in C. The server receives requests at a certain rate measured in *request per second* (RPS). *Latency* means the time that was measured on the client machine between sending a particular request and receiving the response. *Unloaded latency* means the latency when the server is completely free and answers single requests, one at a time with long pauses in between. *Throughput* denotes the request density which the server could sustain. This was usually measured by gradually increasing the request rate and carefully determining the point at which the server starts dropping requests due to overload. The *hot* percentage denotes the amount of hot requests. A hot request is a request for a function which the server already has in memory. In contrast, a *cold* request requires the server to read the binary from the file system and set up internal data structures to create contexts for the function. Compared are

- `dandelion-wasm`: Dandelion using the rWasm-based Wasm backend
- `dandelion-wasmtime`: Dandelion using the Wasmtime backend, using pre-compiled binaries by default
- `dandelion-mmu`: Dandelion using the MMU backend
- `dandelion`: Dandelion using the CHERI-based backend (Morello only)
- `firecracker`: Amazon Firecracker
  - this is Amazon's open source FaaS platform
  - AWS Lambda is based on Firecracker, which makes it an attractive baseline
- `baremetal`
  - a version of dandelion that has the user function compiled into the platform
  - it does not run on any backend and does not do any sandboxing
  - it immediately responds to a request as quickly as possible with the result
  - this is the theoretical optimum for any sandboxing backend in Dandelion

4.1 Latency Under Varying Load

4.1.1 General Purpose Hardware (AWS)



(a) Latency of a 128 × 128 matrix multiplication, under different server RPS, when all requests are cold.

(b) Same as left for hot requests.

Figure 4.1. AWS matmul RPS sweep latency, lower is better.

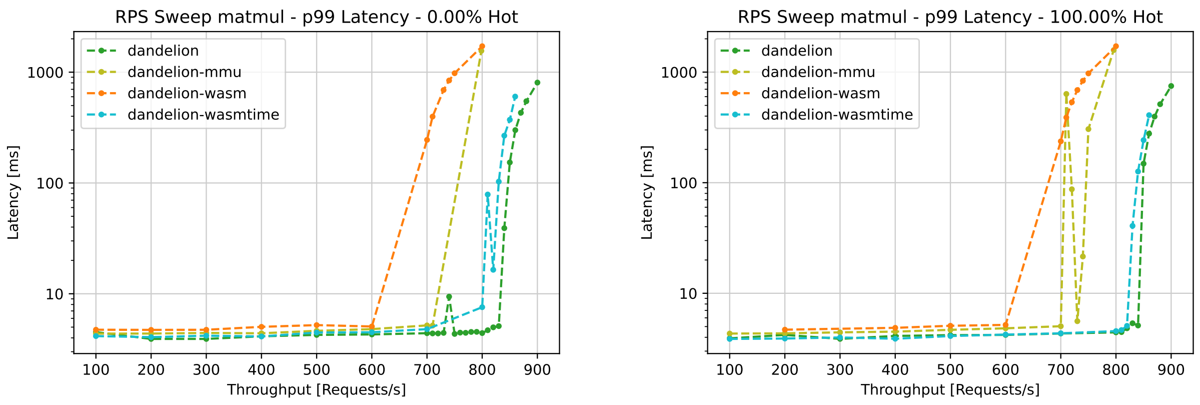
In addition to the general implementation described above we also implemented a list of optimizations, such as using multiple cores to dispatch functions in the server, and eliminating some data copies in Dandelion. Based on these improvements, figure 4.1 compares the architecture-independent backends under different server load (RPS). The space where the latency increases drastically resembles the maximum load the server can effectively sustain.

Cold. The cold experiment shows two interesting relations. While the `dandelion-wasm` backend significantly outperforms even the process-based isolation backend, `dandelion-wasmtime`'s performance is somewhat low.

Hot. Also in the hot experiment, `dandelion-wasm` takes the lead by a significant margin as the only backend clearly beating Firecracker. Firecracker is based on micro-VMs which are created on cold requests, and re-used on hot requests. Therefore, beating Firecracker already under 100% hot load is a particularly strong result. The VM-based approach is expected to quickly reach its limits once cold requests come in, because setting up a VM is expensive. Unfortunately, at this point we don't have very comprehensive Firecracker cold measurements to compare here.

rWasm vs. Wasmtime. Even though `dandelion-wasmtime` shown here uses Wasmtime's pre-compilation feature to pre-compile binaries, it is not competitive with the other backends under high RPS. Dandelion is highly parallel, so we investigated possible synchronization bottlenecks, checked the Wasmtime documentation (version `16.0.0`), and integrated some small optimizations, but we couldn't improve it beyond what is shown here. At 4K RPS we recorded cache and TLB misses of both `dandelion-wasmtime` and `dandelion-wasm` and found that `dandelion-wasmtime` yields about 60% more data TLB load misses. This indicates that the main reason for the quick degradation of `dandelion-wasmtime` is the internal fragmentation described earlier. Notice, however, that `dandelion-wasmtime` actually has the lowest latency for low RPS. We will explore this insight further below.

4.1.2 Morello



(a) cold (b) hot

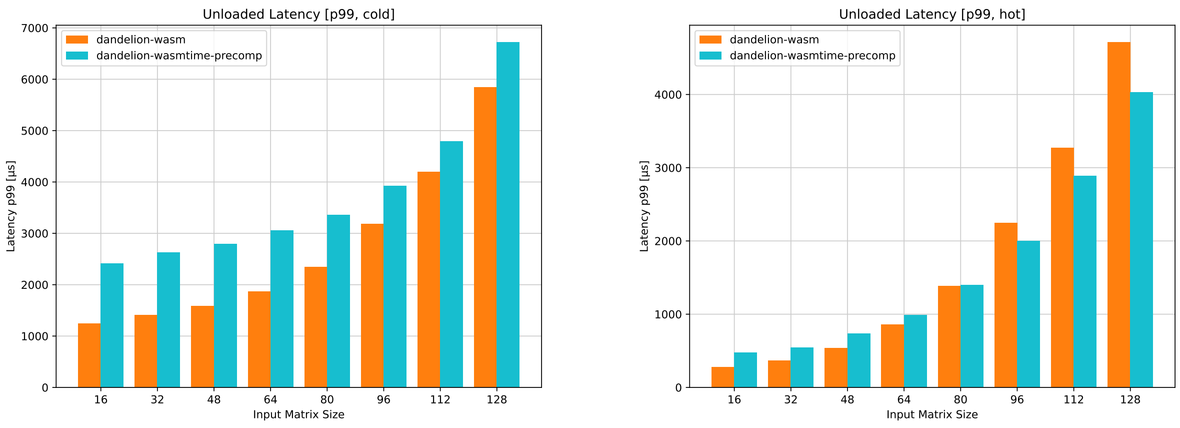
Figure 4.2. Morello matmul RPS sweep latency, lower is better.

Figure 4.2 compares Dandelion backends on Morello, including the CHERI backend which makes use of this hardware. Since this is running on a modified Linux kernel with a modified dynamic loader (with added support for CHERI capabilities), the lower throughput of `dandelion-wasm` relative to the other backends compared to the AWS measurements is likely in parts due to the system loader being slower. This could be improved by extending Dandelion's own optimized loader (which is used by the MMU and CHERI backends) to also handle the binaries obtained from the rWasm compilation toolchain. One the other hand, `dandelion-wasmtime` performs very well here. In particular, the Wasmtime backend is here highly competitive with the CHERI backend which specifically leverages the special hardware for isolation, while Wasmtime does not and can run on a variety of platforms.

The performance spikes you can see are likely an issue of the async runtime, and possibly the client implementation which we found has some nontrivial delay issues.

4.2 Unloaded Latency

We will now take a closer look at how the two Wasm backends compare for *unloaded* latency, i.e. how quickly they respond when nothing else is happening on the server. We now vary the matrix size. This induces differences in computational load (because a larger matrix multiplication is more expensive) as well as in the amount of data to be moved into and out of the contexts. All measurements here were taken in the general purpose AWS setup.



(a) Cold requests with the Wasmtime backend using pre-compiled binaries.

(b) Same as left for hot requests.

Figure 4.3. AWS unloaded latency Wasm vs. Wasmtime (precompiled); lower is better.

Figure 4.3 compares the two backends for hot and cold requests. The higher latency of `dandelion-wasmtime` for very small matrices shows that Wasmtime has more setup overhead, so `dandelion-wasm` is faster. However, `dandelion-wasmtime` catches up as we increase payload, and even overtakes `dandelion-wasm` as we can see in figure 4.1 (b) where, under low RPS, `dandelion-wasmtime` is actually the fastest backend! This means, in this particular case, Wasmtime is faster when actually *running* the Wasm binary than the native binary generated through rWasm executes, and there is a tradeoff determining which backend is better depending on the payload size. Note that this might strongly depend on the particular user source code, but it shows the Wasmtime JIT approach can be faster.

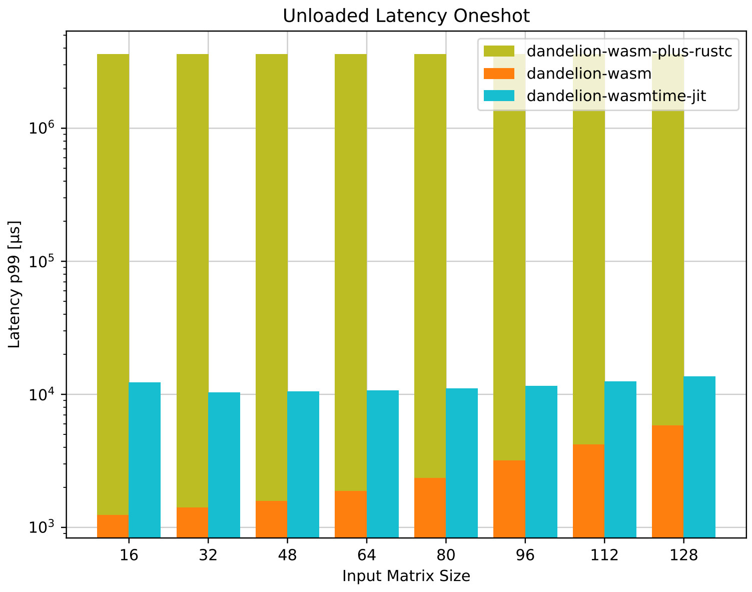


Figure 4.4. AWS unloaded latency oneshot; lower is better.

Oneshot Model. So far we only considered a model where one entity registers a function, and a possibly different entity invokes it. However, for some use-cases the code to be executed isn't known in advance. To also consider this use case, we will call *oneshot* the model in which the goal is a single execution of a function not known before invokation, i.e. any AOT function processing (compilation) is part of the critical path.

Figure 4.4 (note the log scale) compares again `dandelion-wasm` and `dandelion-wasmtime` in the oneshot model. In contrast to the previous plots where `dandelion-wasmtime` uses pre-compilation, `dandelion-wasmtime-jit` does not but tries to both pre-compile and run the Wasm binary as quickly as possible. For `dandelion-wasm`, we added the compilation time required to turn the Wasm binary into a safe native binary through rWasm and rustc. While `dandelion-wasmtime-jit` is slower than `dandelion-wasmtime-precomp`, it dominates `dandelion-wasm` by several orders of magnitude. It is also interesting to note how the runtime of `dandelion-wasmtime-jit` barely increases with increased payload, which indicates that Wasmtime optimizes the function at runtime effectively.

4.3 Peak Throughput Under Varying Payload

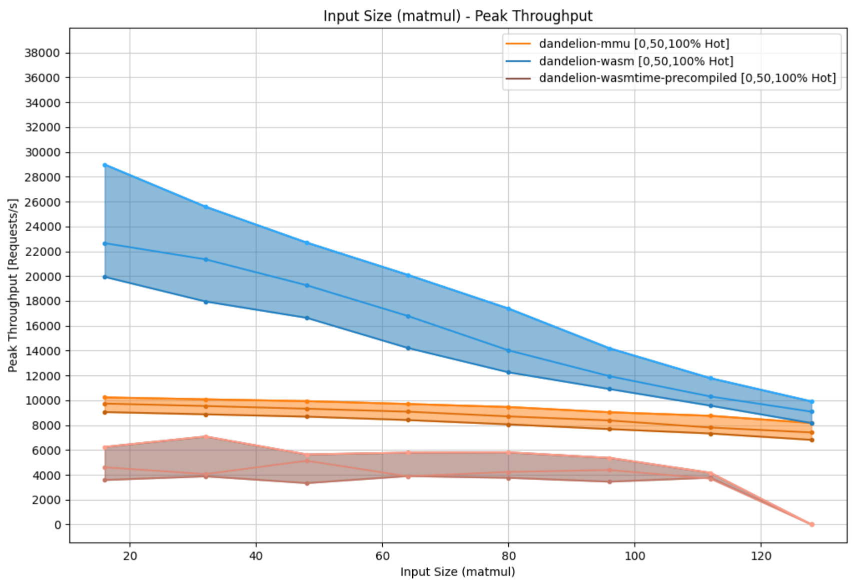


Figure 4.5. Oneshot model.

Figure 4.5 provides further insight into the scalability of the Wasm and MMU backends. For each backend, it shows the range in peak throughput between 100% cold requests and 100% hot requests. Since this measures peak throughput on the AWS setup, the performance of `dandelion-wasmtime` is expected since we already found it doesn't scale very well here. `dandelion-wasm`, however, seems to take significant advantage of smaller payloads and achieves much higher throughput whiel `dandelion-mmu` gets only marginally faster.

5 Related Work

WebAssembly. Much of this work is based on the original paper on rWasm (Bosamiya, Lim, and Parno, 2022). Narayan, Garfinkel, Lerner, *et al.*, 2019 provides an early high-level paper about the potential use of WebAssembly to implement general-purpose SFI. Narayan, Disselkoen, Garfinkel, *et al.*, 2020 and Narayan, Disselkoen, and Stefan, 2021 present RLBox, a C++ framework for retrofitting isolation of third party libraries, used in Firefox. RLBox already has a backend using wasm2c, and could probably be extended by an rWasm backend. However, since Dandelion doesn't interface with the functions other than through the SDK data structure, most of the features of RLBox don't apply to our use case.

Narayan, Garfinkel, Taram, *et al.*, 2023 discusses using Wasm for FaaS, highlights some limitations of purely software-based fault isolation mechanisms, and suggests hardware extensions which can aid sandboxing using Wasm. Given appropriate hardware, this could be the foundation for yet another Dandelion backend leveraging Wasm for sandboxing.

While we do not consider module-internal safety so far, such efforts could be interesting to function authors for some classes of applications. Michael *et al.*, 2023 proposes an extension of the Wasm standard enforcing module-internal memory safety. Since MSWasm support for rWasm is being worked on, this might be added in the future.

Efforts are being made to improve safety of interaction with, or between Wasm modules, such as Rao *et al.*, 2023. We do not consider this direction of research at the moment, because Dandelion purposefully does not implement any runtime interaction with functions.

FaaS. *Sledge* (Gadepalli, McBride, *et al.*, 2020) describes a Wasm-based FaaS system with a focus on edge computing, which has many parallels with this work. They also use a custom compiler to turn Wasm binaries into ELF binaries ahead of time. A notable difference, though, is that they still provide a POSIX interface to the function workloads, which has implications for loading and linking. *FAASM* (Shillaker and Pietzuch, 2020) is yet another FaaS platform that is built around Wasm, with a focus on avoiding data copies between function contexts. Also here, the host provides an extensive interface for I/O which Dandelion purposefully avoids. A comparison between Dandelion and FAASM is already being worked on.

6 Future Work

Some aspects of this project which future research could expand or improve on are:

libc support. There are mainly two approaches for supporting user code that depends on libc on the Wasm backends: 1. provide a libc which compiles to `wasm32`, or 2. compile the user code against WASI-libc and then provide a virtual WASI implementation. dlibc currently doesn't support the `wasm32` target, though future efforts of supporting it are planned.

Extended Loader. Dandelion has a custom ELF loader. The `dandelion-wasm` backend currently uses the native system loader to dynamically load the library. The reason for this is a limitation in rustc inherited from LLVM. When compiling to aarch64, LLVM won't fully support position independent code (PIC), resulting in a binary that still requires some relocations to be performed at load time. Dandelion's loader does not resolve relocations at the moment, though there is no conceptual reason why it can't. Since the decreased performance of `dandelion-wasm` on Morello is likely due to a slower system loader, and since using our own loader will give us more control over the loading process in general, it might be worth to extend Dandelion's loader to support relocations.

Evaluation. More extensive evaluations, with different demands, after the issues mentioned in the evaluation are solved.

7 Conclusion

We introduced the topic of sandboxing and provided two different examples how WebAssembly can be used to implement efficient software sandboxing. We provided solutions for the main technical challenges of integrating those approaches into the Dandelion cloud platform. Assuming no host API, leaving mainly memory safety to worry about, we showed how we can efficiently turn untrusted WebAssembly code into safe native code with great runtime performance using rWasm. We evaluated performance tradeoffs between Dandelion's backends in multiple setups. We found that in each setup, one WebAssembly backend performs very well, illustrating the generality and scalability of these two approaches combined. I hope the ideas and results illustrated in this work will spark further research into leveraging WebAssembly in order to greatly improve safety, and resource efficiency of software.

Bibliography

Alliance, Bytecode (2023a). *WAMR*. https://github.com/bytecodealliance/wasm-micro-runtime.

— (2023b). *wasmtime*. https://wasmtime.dev/.

Bosamiya, Jay, Wen Shih Lim, and Bryan Parno (2022). "Provably-Safe Multilingual Software Sandboxing using WebAssembly". In: *31st USENIX Security Symposium (USENIX Security 22)*.

CVE-2023-26489. (2023). Available from www.cve.org, CVE-ID CVE-2023-26489. URL: https://www.cve.org/CVERecord?id=CVE-2023-26489.

Fastly (2023). *Compute@Edge*. https://docs.fastly.com/products/compute-at-edge.

Gadepalli, Phani Kishore, Sean McBride, *et al.* (2020). "Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge". In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. Delft, Netherlands: Association for Computing Machinery, pp. 265–279. ISBN: 9781450381536. DOI: 10.1145/3423211.3425680. URL: https://doi.org/10.1145/3423211.3425680.

Gadepalli, Phani Kishore, Gregor Peach, *et al.* (2019). "Challenges and Opportunities for Efficient Serverless Computing at the Edge". In: *2019 38th Symposium on Reliable Distributed Systems (SRDS)*.

Group, WebAssembly Community (2023). *WebAssembly Binary Toolkit*. https://github.com/WebAssembly/wabt.

Holley, Bobby (2021). *WebAssembly and Back Again: Fine-Grained Sandboxing in Firefox 95*. https://hacks.mozilla.org/2021/12/webassembly-and-back-again-fine-grained-sandboxing-in-firefox-95/.

JUNG, RALF *et al.* (2018). "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28, e20. DOI: 10.1017/S0956796818000151.

Lehmann, Daniel, Johannes Kinder, and Michael Pradel (Aug. 2020). "Everything Old is New Again: Binary Security of WebAssembly". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, pp. 217–234. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

Michael, Alexandra E. *et al.* (2023). "MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code". In: *Proc. ACM Program. Lang.* 7.POPL. DOI: 10.1145/3571208. URL: https://doi.org/10.1145/3571208.

Narayan, Shravan, Craig Disselkoen, Tal Garfinkel, *et al.* (2020). *Retrofitting Fine Grain Isolation in the Firefox Renderer (Extended Version)*. arXiv: 2003.00572 [cs.CR].

Narayan, Shravan, Craig Disselkoen, and Deian Stefan (2021). "Tutorial: Sandboxing (unsafe) C code with RLBox". In: *2021 IEEE Secure Development Conference (SecDev)*, pp. 11–12. DOI: 10.1109/SecDev51306.2021.00017.

Narayan, Shravan, Tal Garfinkel, Sorin Lerner, *et al.* (2019). *Gobi: WebAssembly as a Practical Path to Library Sandboxing*. arXiv: 1912.02285 [cs.CR].

Narayan, Shravan, Tal Garfinkel, Mohammadkazem Taram, *et al.* (2023). "Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. <conf-loc>, <city>Vancouver</city>, <state>BC</state>, <country>Canada</country>, </conf-loc>: Association for Computing Machinery, pp. 266–281. ISBN: 9781450399180. DOI: 10.1145/3582016.3582023. URL: https://doi.org/10.1145/3582016.3582023.

Rao, Xiaojia *et al.* (2023). "Iris-Wasm: Robust and Modular Verification of WebAssembly Programs". In: *Proc. ACM Program. Lang.* 7.PLDI. DOI: 10.1145/3591265. URL: https://doi.org/10.1145/3591265.

Shillaker, Simon and Peter Pietzuch (July 2020). "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, pp. 419–

433. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shillaker.

Shymanskyy, Volodymyr (2023). *wasm3*. https://github.com/wasm3/wasm3.

Technologies, Parity (2023). *wasmi*. https://github.com/paritytech/wasmi.

Varda, Kenton (2018). *WebAssembly on Cloudflare Workers*. https://blog.cloudflare.com/webassembly-on-cloudflare-workers.

Wahbe, Robert *et al.* (1993). “Efficient Software-Based Fault Isolation”. In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP ’93. Asheville, North Carolina, USA: Association for Computing Machinery, pp. 203–216. ISBN: 0897916328. DOI: 10.1145/168619.168635. URL: https://doi.org/10.1145/168619.168635.

Watt, Conrad *et al.* (2023). “WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly”. In: *Proc. ACM Program. Lang.* 7.PLDI. DOI: 10.1145/3591224. URL: https://doi.org/10.1145/3591224.

WebAssembly (n.d.). Available at https://webassembly.org.

“What Serverless Computing Is and Should Become: The Next Phase of Cloud Computing” (2021). In: URL: https://cacm.acm.org/magazines/2021/5/252179-what-serverless-computing-is-and-should-become/fulltext.

Appendix A - Notes on Experiments

`dandelion-wasmtime-*` and `dandelion-cheri` results were measured at

- experiments commit: `e02bfea73f52618ac1ee4b596860510670045b03`
- using Dandelion commit: `ae4acf6e8d0f172beaf2688daf916c6fdd9d387f`

Other results (`dandelion-wasm`, `dandelion-mmu`, `baremetal`, `firecracker`) were measured at

- experiments commit: `3e6a2cb28c6583b0517a3e26c050188d368a1a91`
- using Dandelion commit `c16efc902b341d72dc35d8a77271b925d50712f8`

The reason for the split is mainly that changes were made to Dandelion while developing the Wasmtime backend, which introduced issues in the already existing Wasm backend. While these changes are not expected to significantly change the measurements for other backends, detailed comparison between Wasmtime backends and others in the sweep plots should be interpreted with caution. It is possible that the Wasmtime backends should perform slightly better in comparison in the future.

Figure 4.4 was recorded with our own experimental closed-loop loader system, using 4096 users (i.e. ≤ 4096 requests can be in flight simultaneously).