CGO-less Foreign Function Interface with WebAssembly

GopherCon 2022 - Takeshi Yoneda at Tetrate

Foreign Function Interface(FFI)

Foreign Function Interface(FFI)

*“A foreign function interface (FFI) is a mechanism by which a program written in **one programming language can call routines** or make use of services **written in another**.” – wikipedia*

```
func main () {



}
```

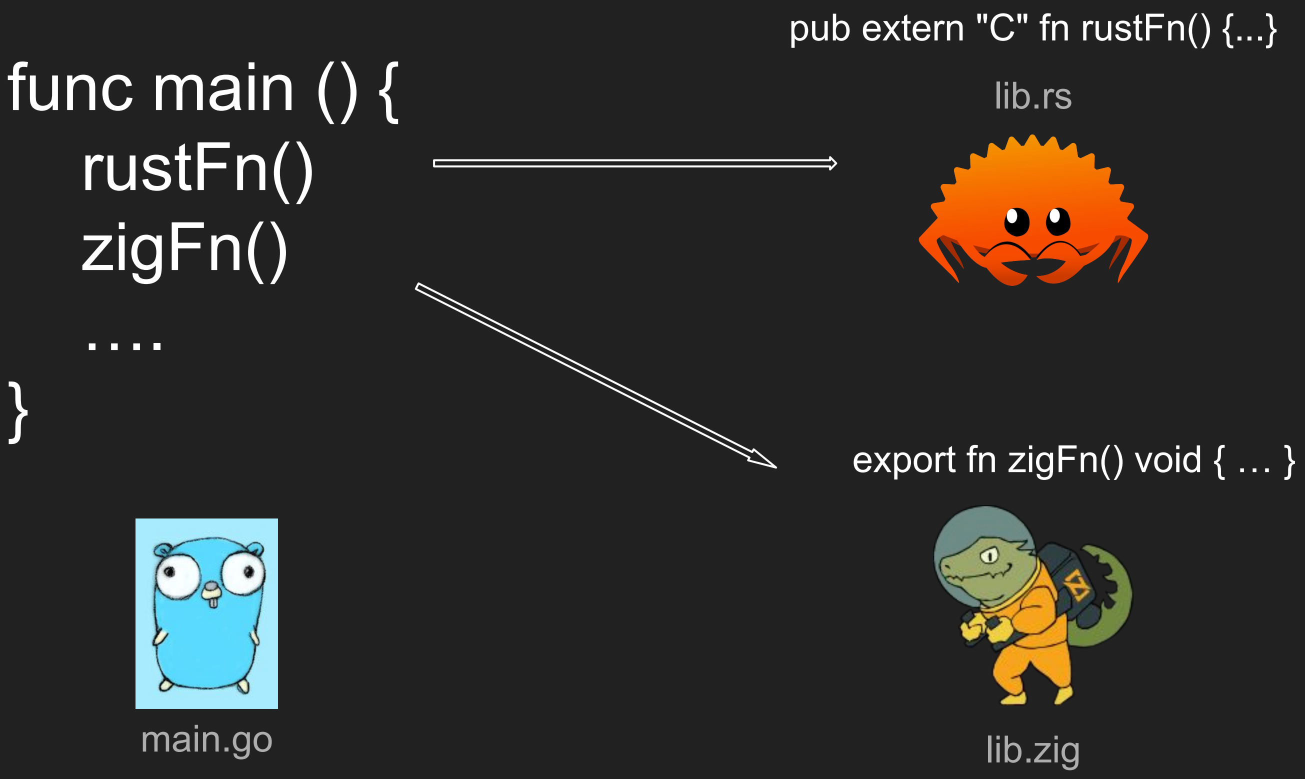
main.go

```
func main () {
    rustFn()


}
```

main.go

```
pub extern "C" fn rustFn() {...}
```

lib.rs



```
func main () {
    rustFn()
    zigFn()
    ....
}
```

main.go

```
pub extern "C" fn rustFn() {...}
```

lib.rs

```
export fn zigFn() void { … }
```

lib.zig

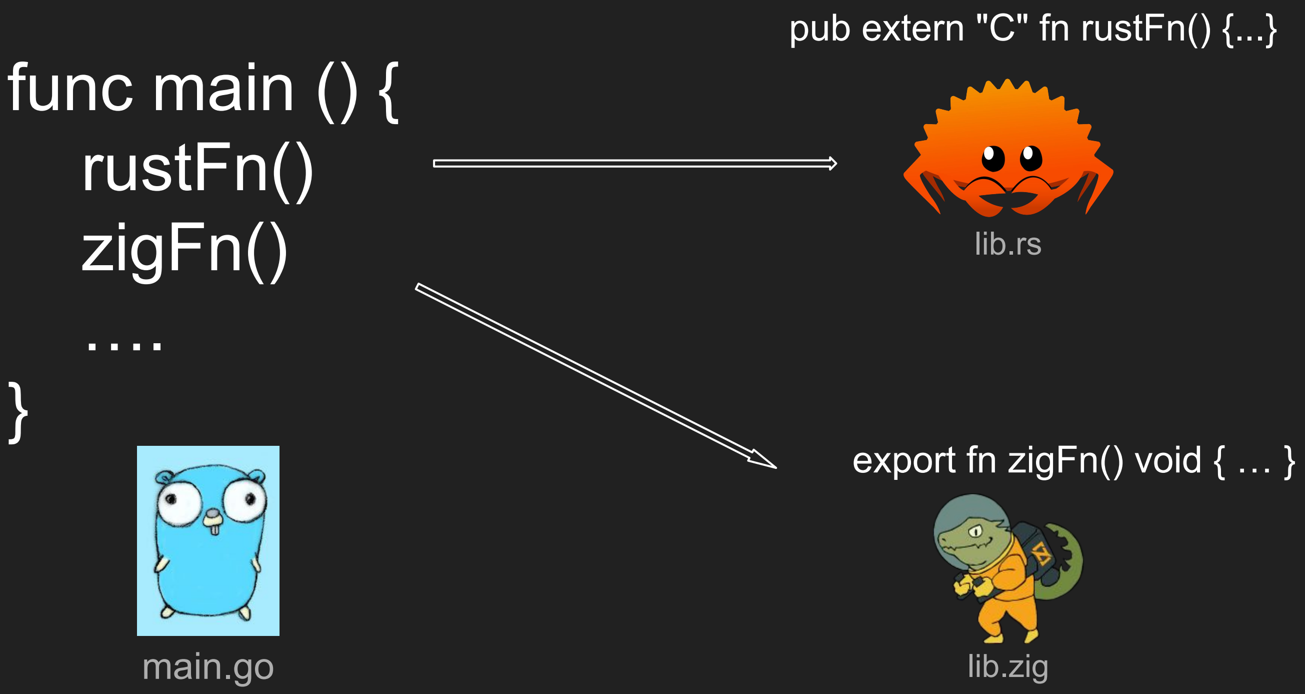
When/Why do we want FFI?

When/Why do we want FFI?

- Reusing softwares in other languages
  - Don’t want to rewrite 100k loc in C

When/Why do we want FFI?

- Reusing softwares in other languages
  - Don’t want to rewrite 100k loc in C
- Plugin System via FFI - Polyglot!
  - Allow users to extend your app in any language



```
func main () {
    rustFn()
    zigFn()
    ....
}
```

main.go

```
pub extern "C" fn rustFn() {...}
```

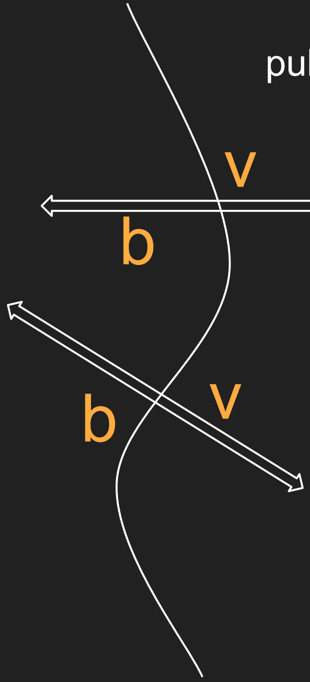
lib.rs

```
export fn zigFn() void { … }
```

lib.zig

```
func main () {
    b := rustFn(v)
    b = zigFn(v)
    ....
}
```

main.go



```
pub extern "C" fn rustFn(v: u32) -> bool {...}
```

lib.rs

```
export fn zigFn(v: u32) bool { … }
```

lib.zig

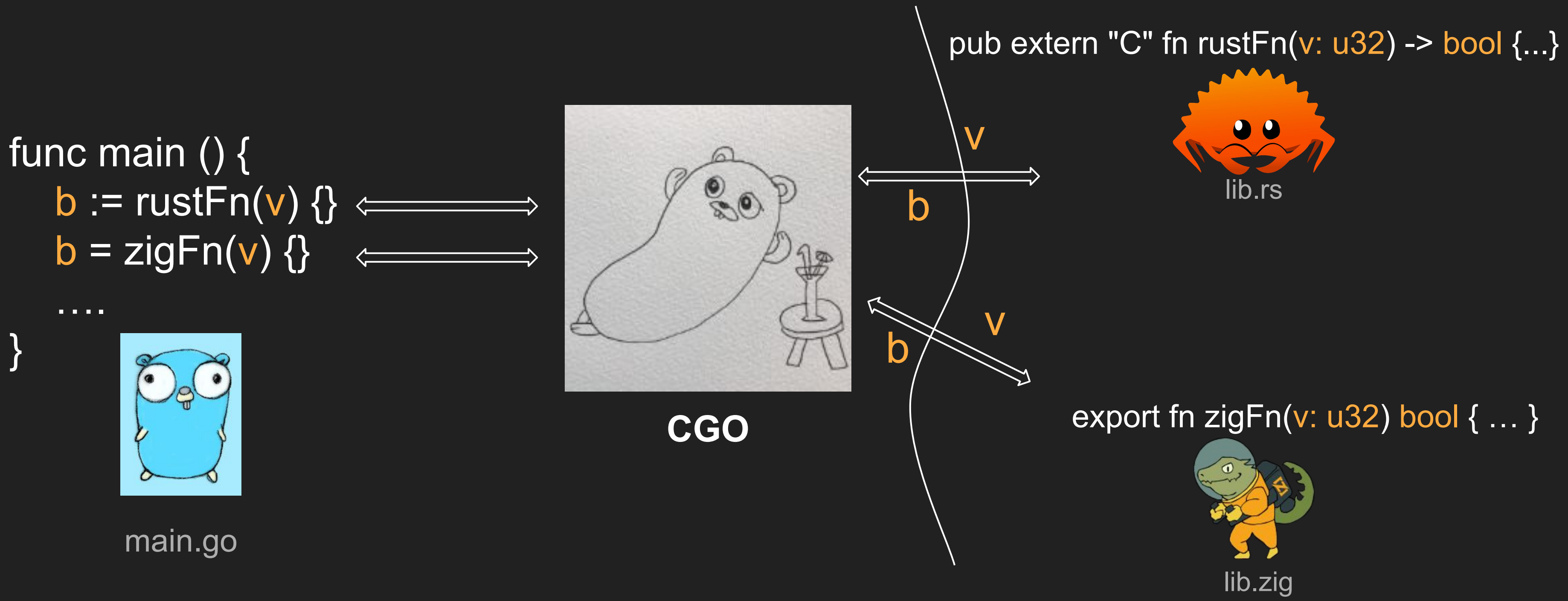
What’s the protocol between Go and another lang?

How Go runtime behaves beyond Go world?

What's the proto nd another lang?

How Go runti nd Go world?

CGO



```
func main () {
    b := rustFn(v) {}
    b = zigFn(v) {}
    ....
}
```

main.go

CGO

```
pub extern "C" fn rustFn(v: u32) -> bool {...}
```

lib.rs

```
export fn zigFn(v: u32) bool { … }
```

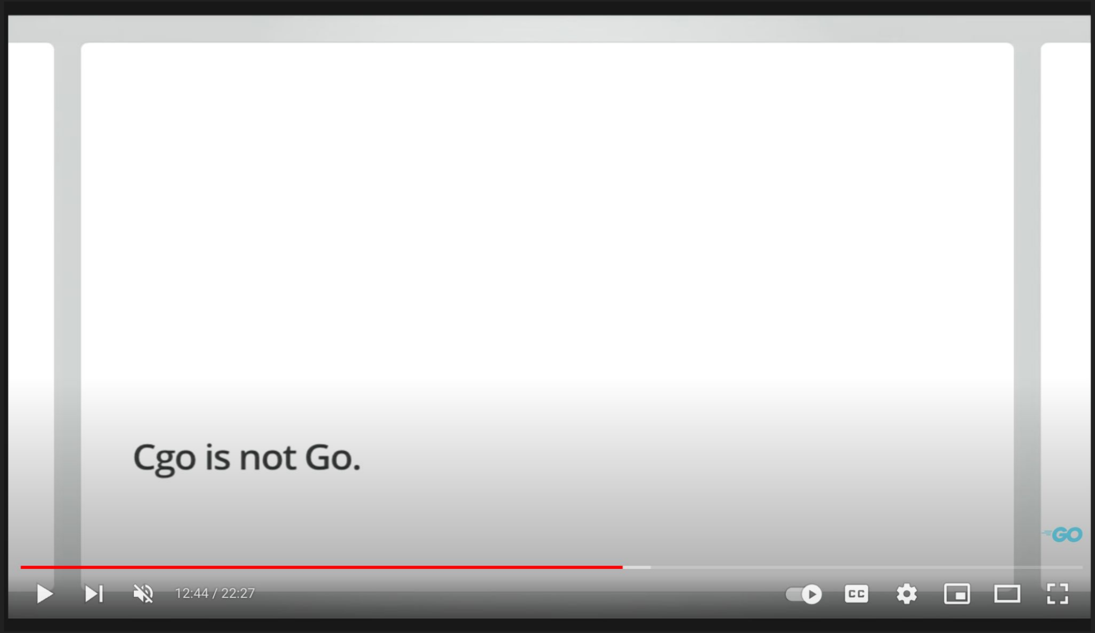
lib.zig

FFI can be done with CGO. The problem solved?

FFI can be done with CGO. The problem solved?

No.

“CGO is not Go”



Cgo is not Go.

Gopherfest 2015 | Go Proverbs with Rob Pike https://youtu.be/PAAkCSZUG1c

CGO troubles

- Dynamic vs Static binary: portability issue
- Cross compilation
- CGO is slow
- Security

CGO troubles

- Dynamic vs Static binary: portability issue
- Cross compilation
- CGO is slow
- Security

```
package main

func main() {  println("hello") }
```

```
package main

func main() {  println("hello") }
```

```
$ go build main.go
```

```
package main

func main() {  println("hello") }
```

```
$ go build main.go
$ ldd main
```

```
package main

func main() {  println("hello") }
```

```
$ go build main.go
$ ldd main
        not a dynamic executable
```

```
package main

import "C"

func main() {  println("hello") }
```

```
package main

import "C"

func main() {  println("hello") }
```

```
$ go build main.go
```

```go
package main

import "C"

func main() {  println("hello") }
```

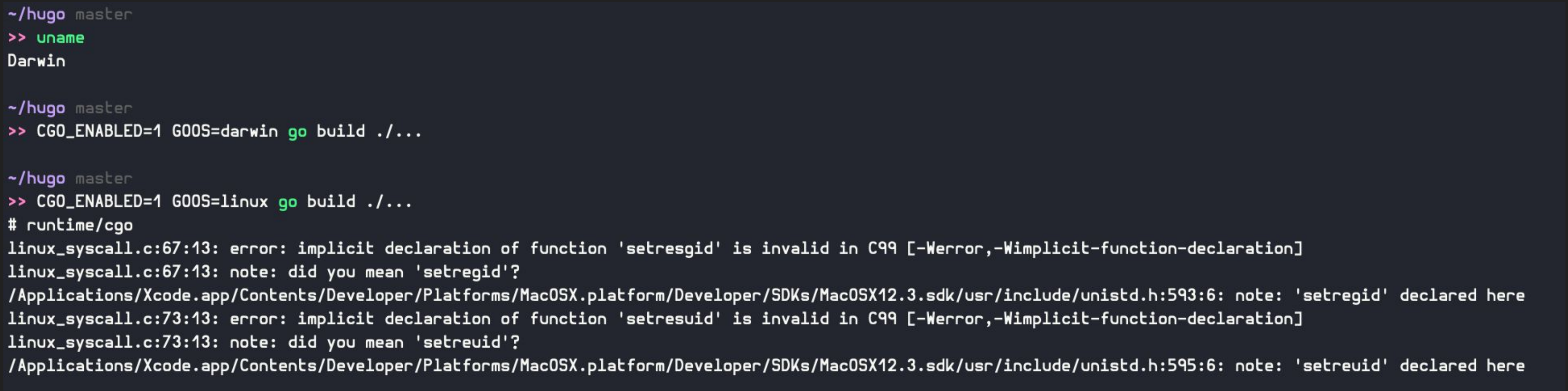
```
$ go build main.go
$ ldd main
```

```go
package main

import "C"

func main() {  println("hello") }
```

```
$ go build main.go
$ ldd main
        linux-vdso.so.1 (0x00007ffd59db2000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fad32c3f000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fad32a4d000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fad32c83000)
```

CGO troubles

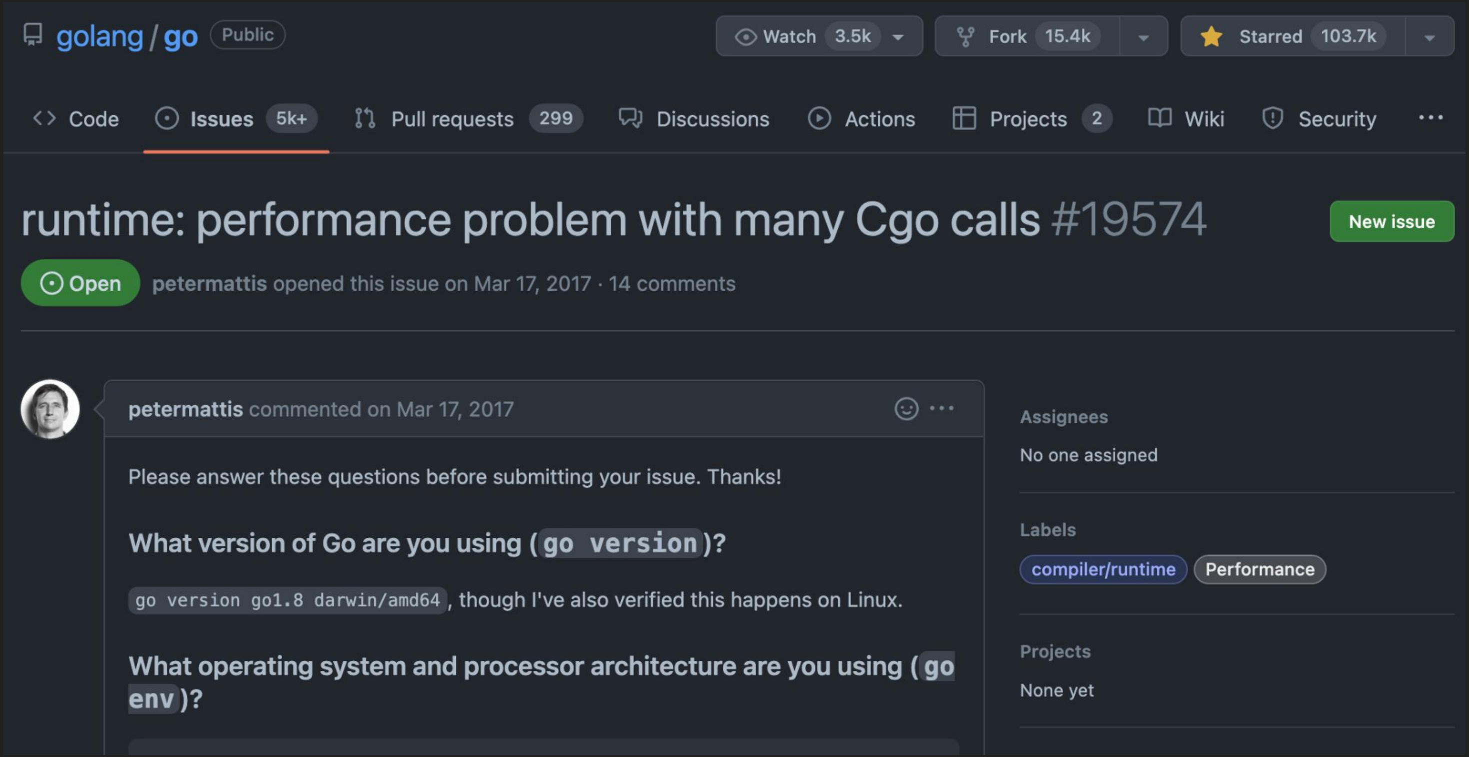
- Dynamic vs Static binary
- Cross compilation
- CGO is slow
- Security



```
~/hugo master
>> uname
Darwin

~/hugo master
>> CGO_ENABLED=1 GOOS=darwin go build ./...

~/hugo master
>> CGO_ENABLED=1 GOOS=linux go build ./...
# runtime/cgo
linux_syscall.c:67:13: error: implicit declaration of function 'setresgid' is invalid in C99 [-Werror,-Wimplicit-function-declaration]
linux_syscall.c:67:13: note: did you mean 'setregid'?
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX12.3.sdk/usr/include/unistd.h:593:6: note: 'setregid' declared here
linux_syscall.c:73:13: error: implicit declaration of function 'setresuid' is invalid in C99 [-Werror,-Wimplicit-function-declaration]
linux_syscall.c:73:13: note: did you mean 'setreuid'?
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX12.3.sdk/usr/include/unistd.h:595:6: note: 'setreuid' declared here
```

CGO troubles

- Dynamic vs Static binary
- Cross compilation
- CGO is slow
- Security



runtime: performance problem with many Cgo calls #19574

Open petermattis opened this issue on Mar 17, 2017 · 14 comments

petermattis commented on Mar 17, 2017

Please answer these questions before submitting your issue. Thanks!

What version of Go are you using (`go version`)?

`go version go1.8 darwin/amd64`, though I've also verified this happens on Linux.

What operating system and processor architecture are you using (`go env`)?

Assignees

No one assigned

Labels

compiler/runtime Performance

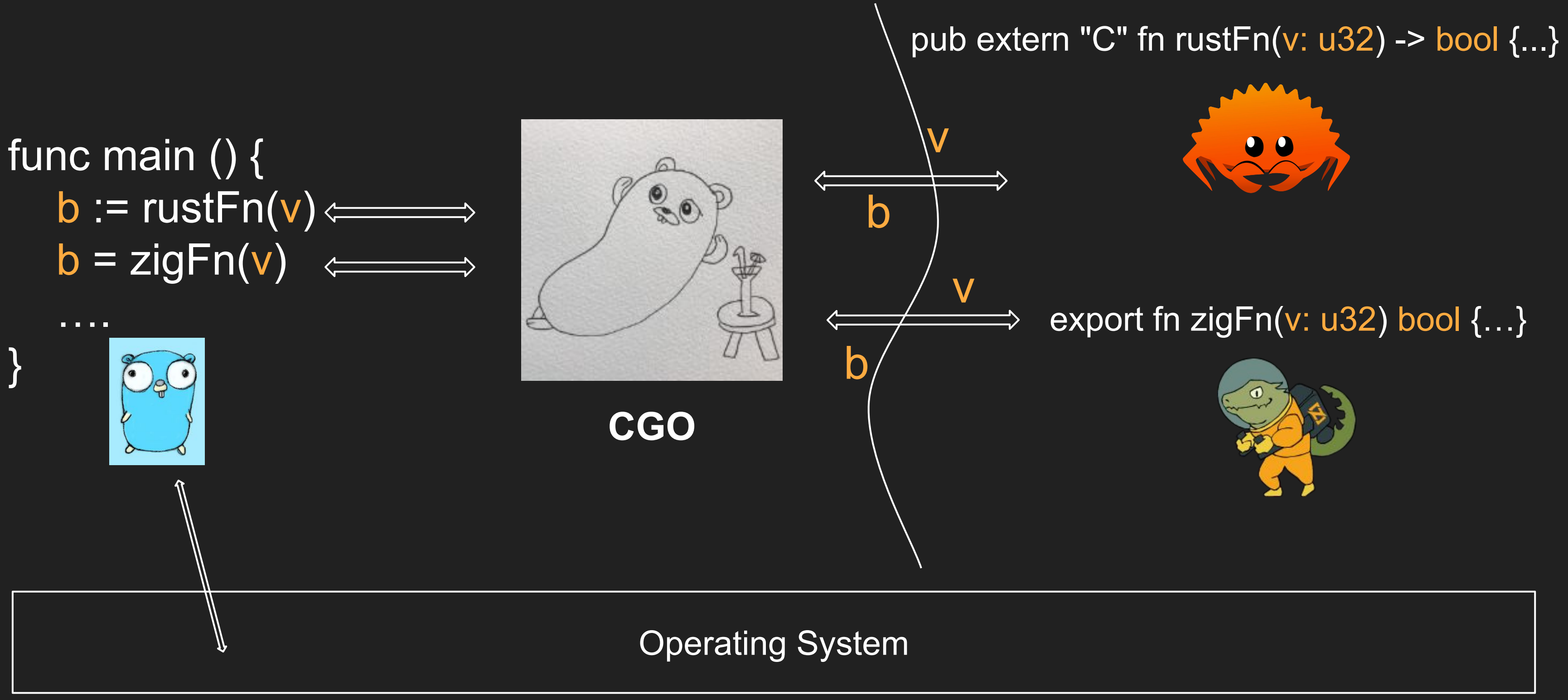
Projects

None yet

https://github.com/golang/go/issues/19574

CGO troubles

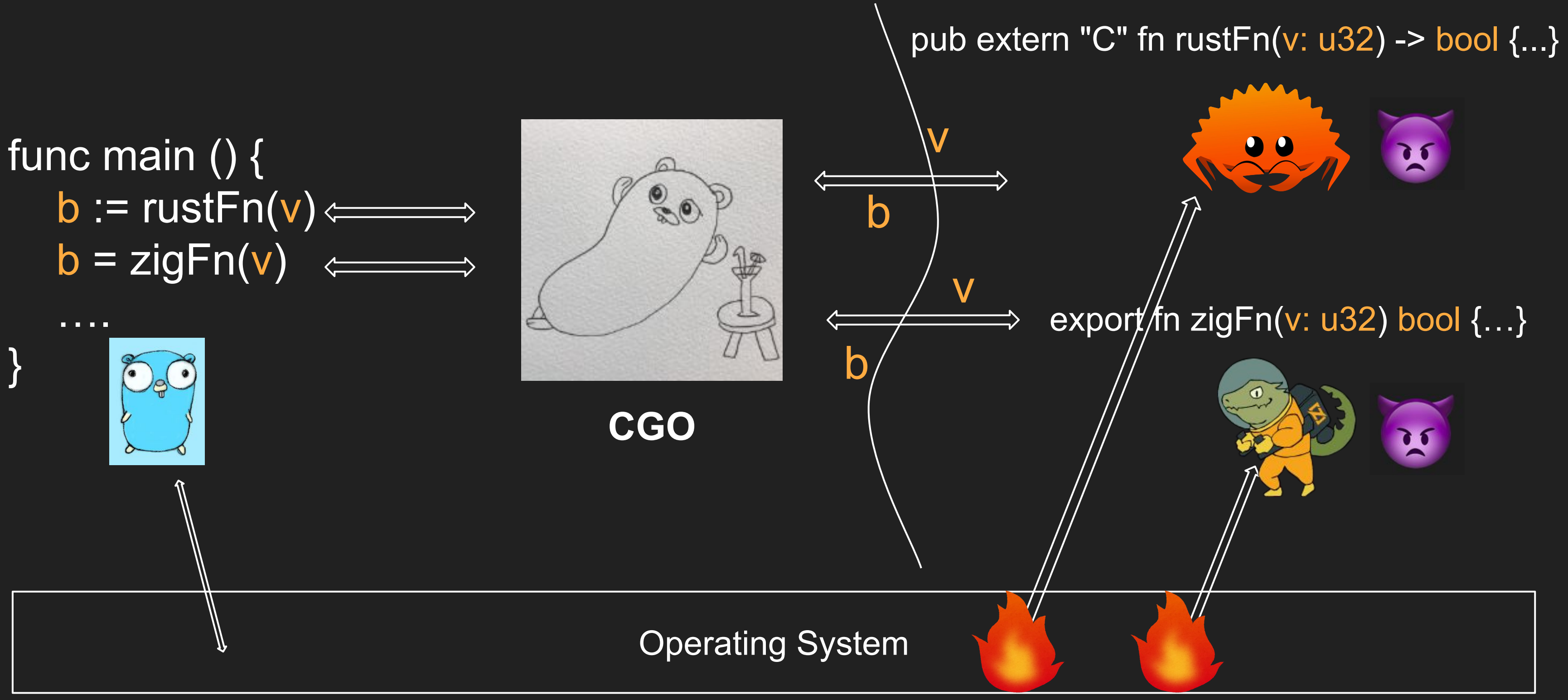
- Dynamic vs Static binary
- Cross compilation
- CGO is slow
- Security



```
func main () {
    b := rustFn(v)
    b = zigFn(v)
    ....
}
```

```
pub extern "C" fn rustFn(v: u32) -> bool {...}
```

```
export fn zigFn(v: u32) bool {…}
```



```
func main () {
    b := rustFn(v)
    b = zigFn(v)
    ....
}
```

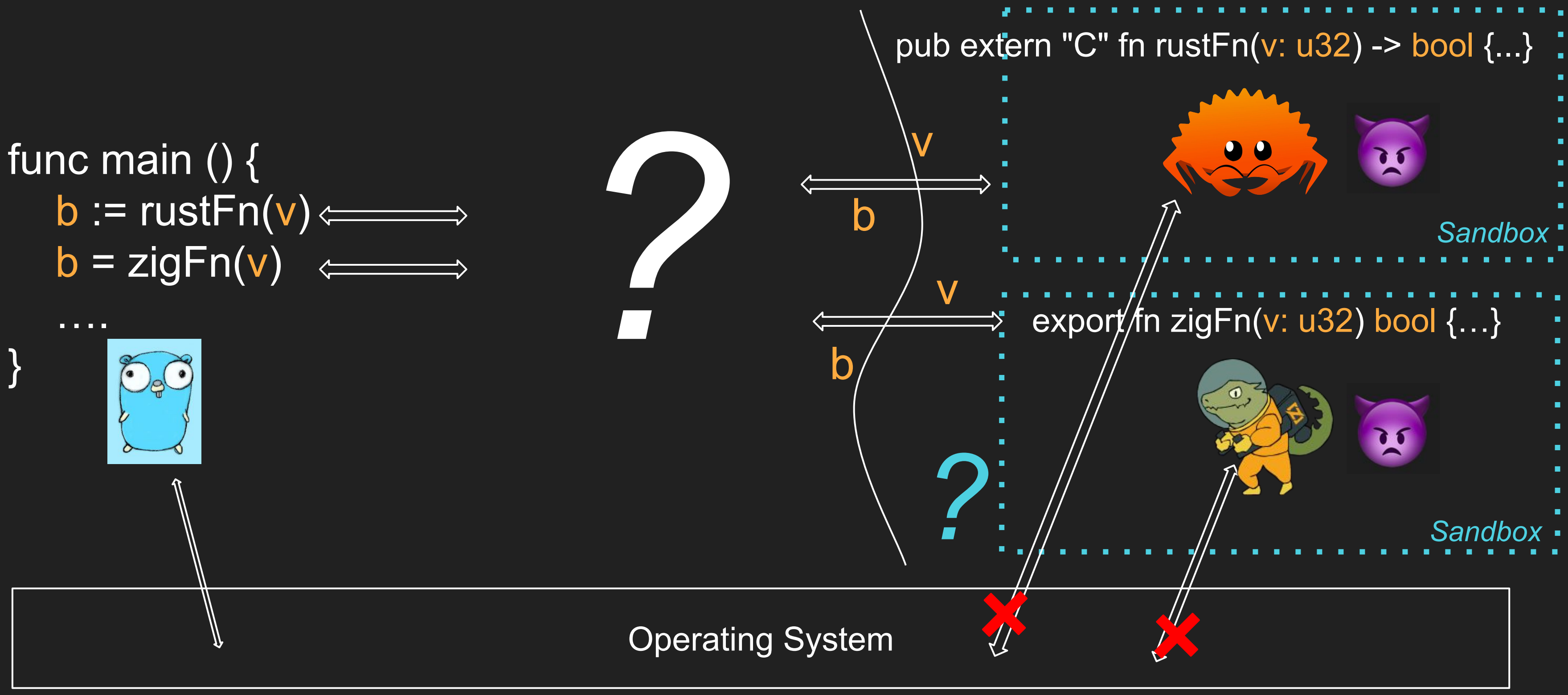
CGO

pub extern "C" fn rustFn(v: u32) -> bool {...}

export fn zigFn(v: u32) bool {…}

Operating System

We need sandox…



```
func main () {
    b := rustFn(v)
    b = zigFn(v)
    ....
}
```

```
pub extern "C" fn rustFn(v: u32) -> bool {...}
```

```
export fn zigFn(v: u32) bool {…}
```

WebAssembly (Wasm)

WebAssembly (Wasm)

- Binary instruction format for a stack-based virtual machine (VM)
- Polyglot
- Security-oriented design
  - Memory guard
  - Deny system calls by default

Wait, isn’t WebAssembly for the web?

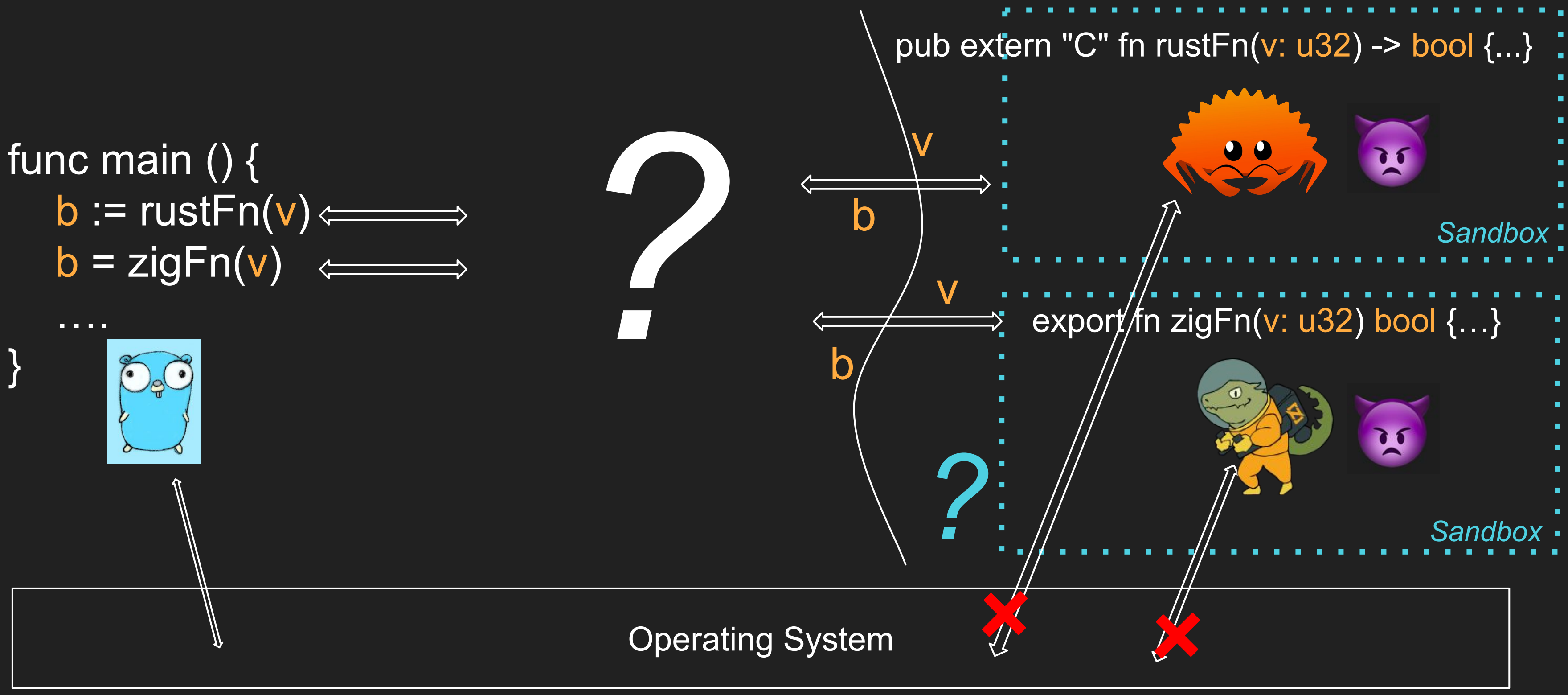
Wasm is not only for the browsers

- Core spec is decoupled from the web concept
- Embeddable in any application with VM implementation

Non-Web examples

envoy

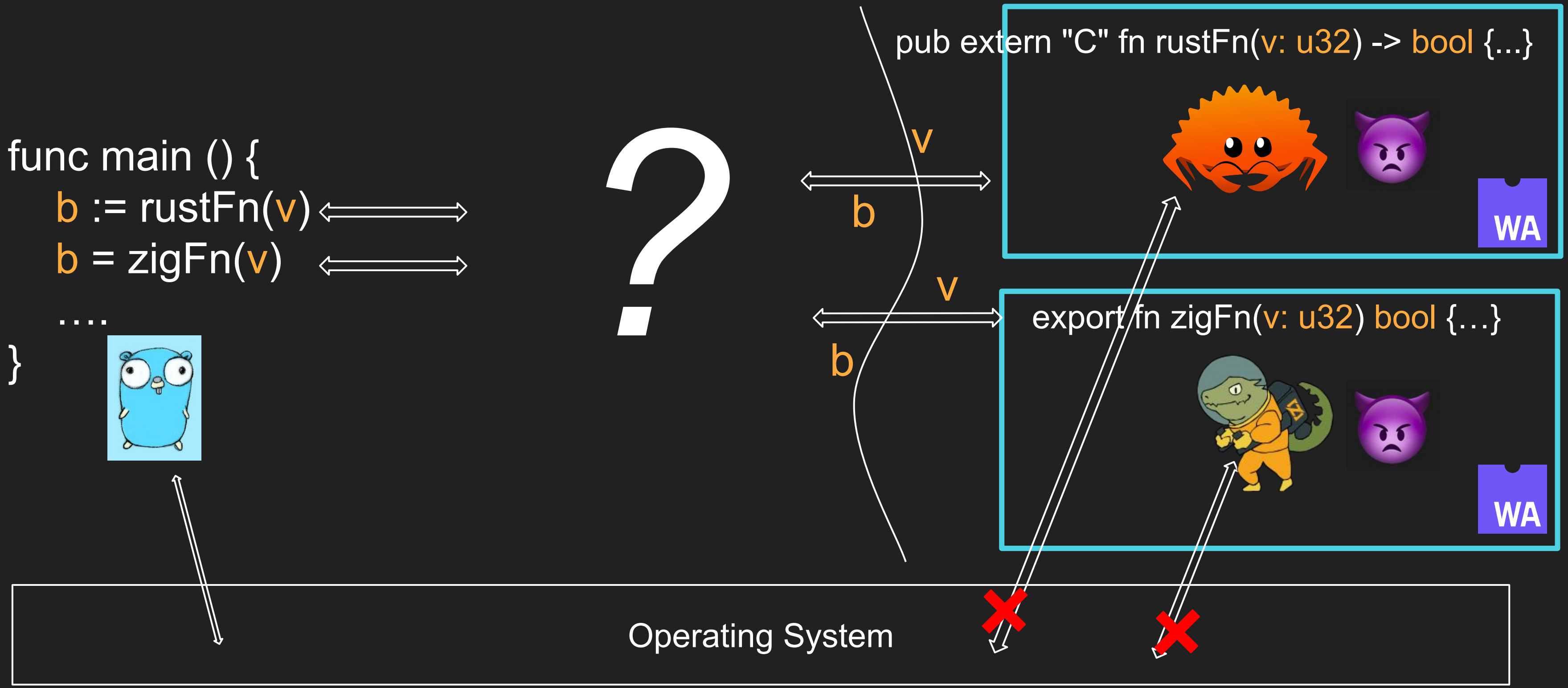
KRUSTLET



```
func main () {
    b := rustFn(v)
    b = zigFn(v)
    ....
}
```

```
pub extern "C" fn rustFn(v: u32) -> bool {...}
```

```
export fn zigFn(v: u32) bool {…}
```



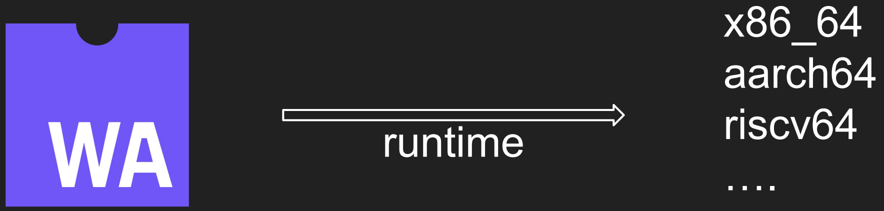
```
func main () {
    b := rustFn(v)
    b = zigFn(v)
    ....
}
```

```
pub extern "C" fn rustFn(v: u32) -> bool {...}
```

```
export fn zigFn(v: u32) bool {...}
```

How to run Wasm binary inside Go?

Wasm needs a VM runtime!



runtime

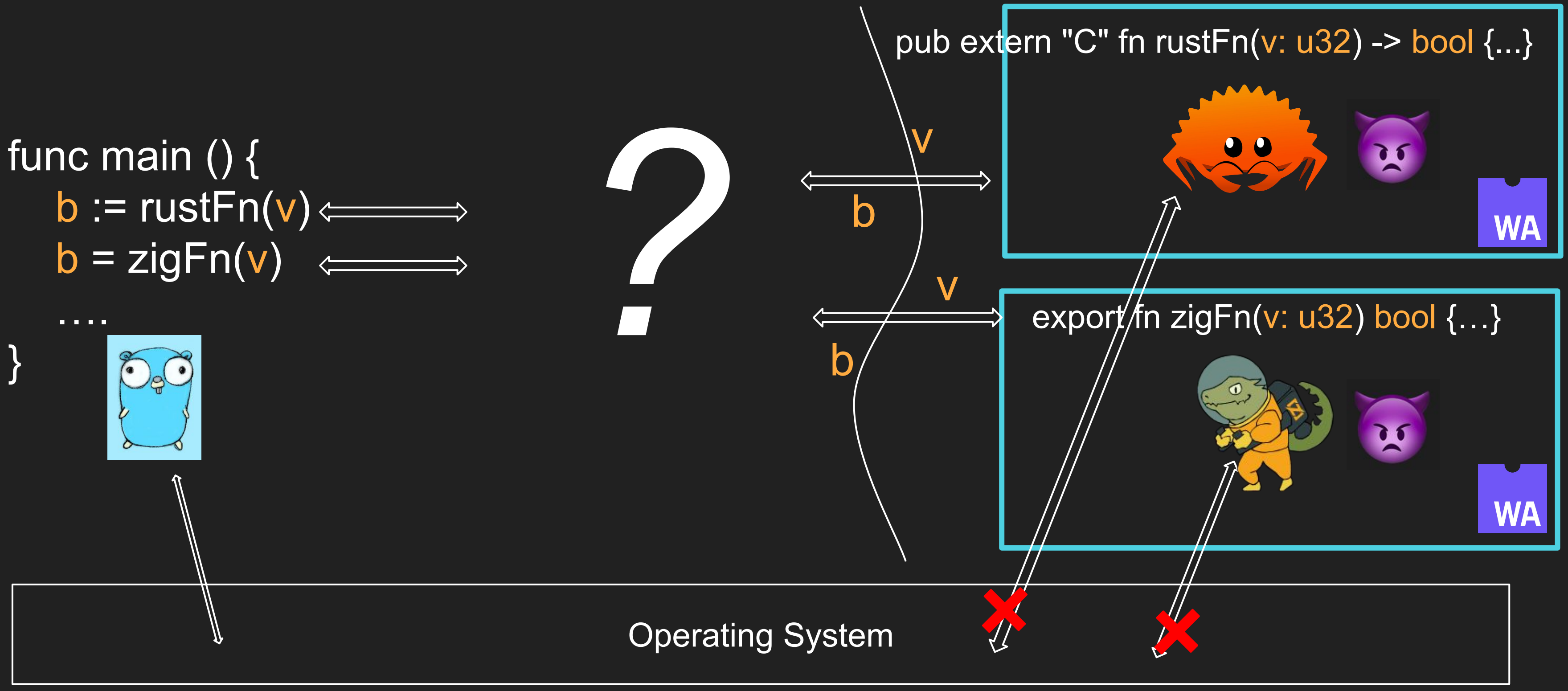
x86_64
aarch64
riscv64
….

wazero.io

the zero dependency WebAssembly runtime for Go developers

What is wazero?

- Started out as my hobby project: now sponsored by Tetrate
- The Wasm runtime with zero dependency
- Written in pure Go, no CGO!



```go
func main () {
    b := rustFn(v)
    b = zigFn(v)
    ....
}
```

?

v

b

```rust
pub extern "C" fn rustFn(v: u32) -> bool {...}
```

WA

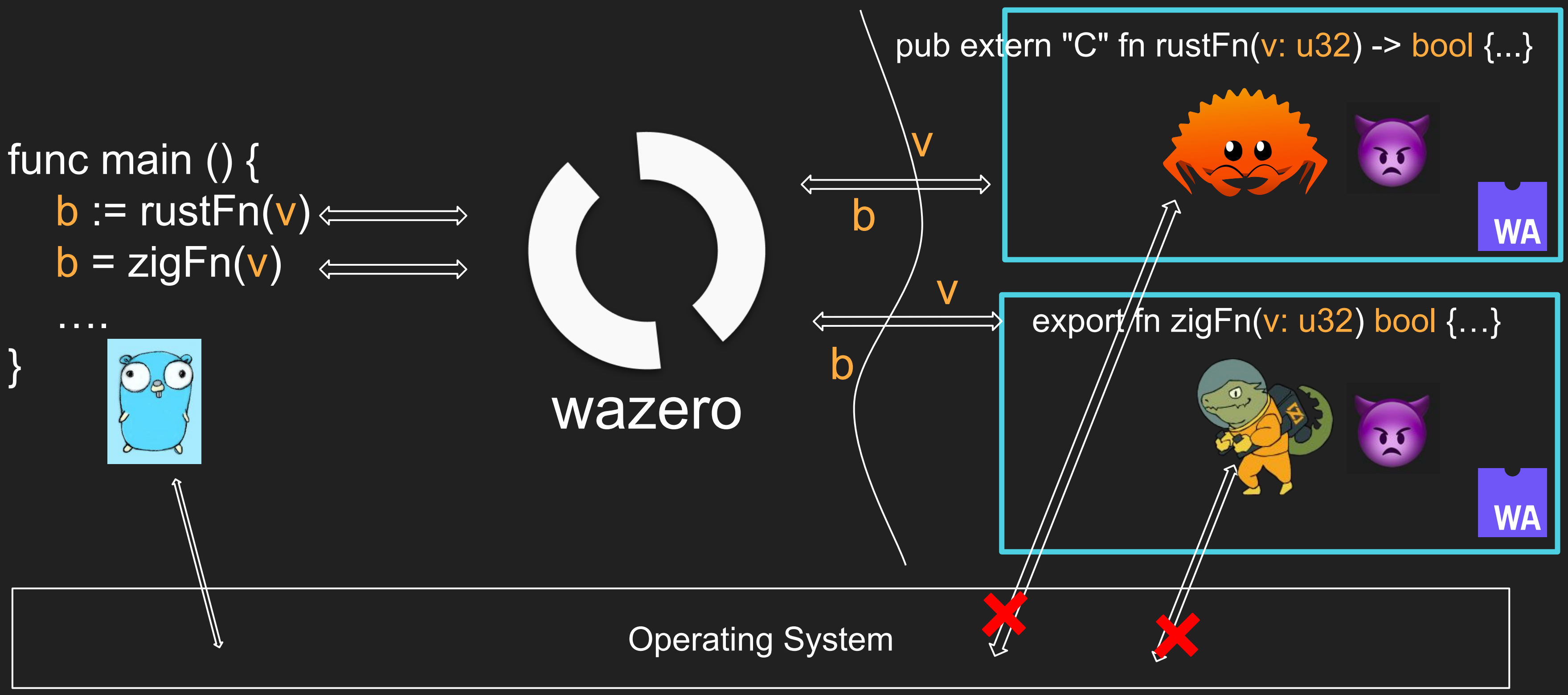
v

b

```zig
export fn zigFn(v: u32) bool {…}
```

WA

Operating System



```
func main () {
    b := rustFn(v)
    b = zigFn(v)
    ….
}
```

```
pub extern "C" fn rustFn(v: u32) -> bool {...}
```

```
export fn zigFn(v: u32) bool {…}
```

=> CGO-less Foreign Function Interface

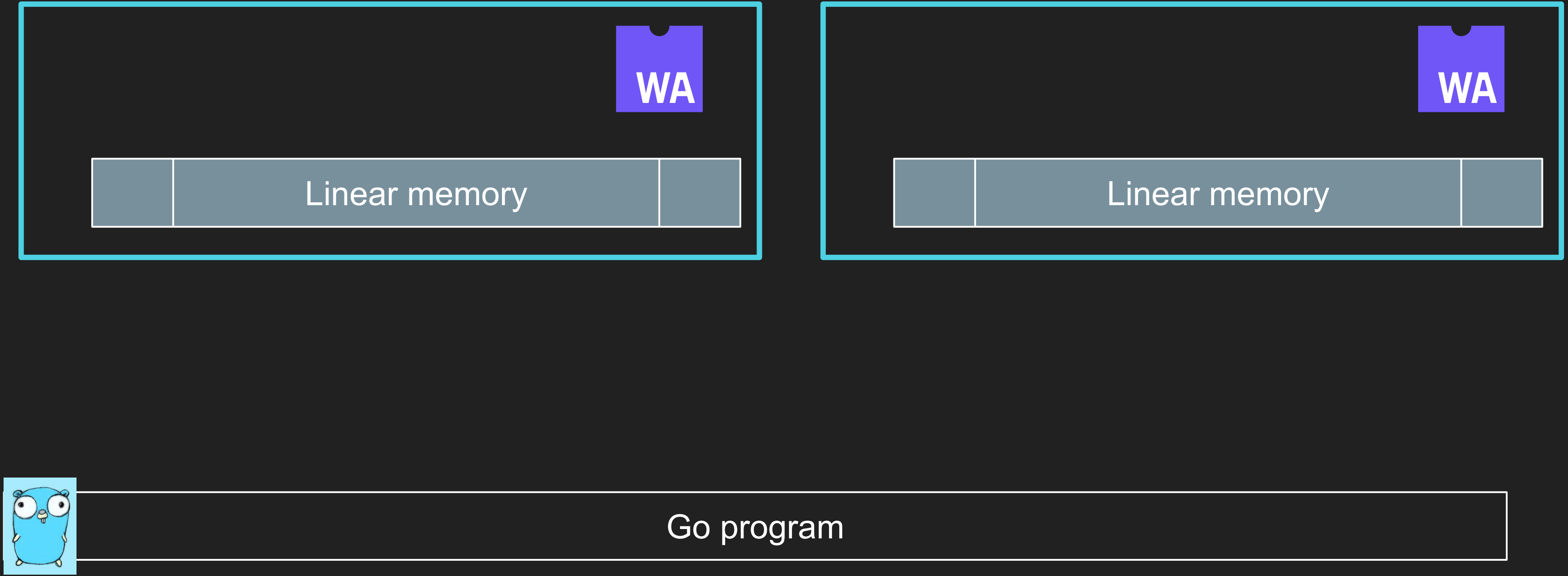
FFI with wazero vs CGO

- No CGO
  - Static binary, cross compilation, etc
- Zero dependency
  - E.g. third party toolchains
- Compile once, run everywhere
- Sandbox environment
  - Memory isolation
  - Deny “system calls” by default

How it works: memory isotation

```
// Instantiate a Zig Wasm binary.
zig := r.InstantiateModuleFromBinary(...)
```

```
// Instantiate a Rust Wasm binary.
rust := r.InstantiateModuleFromBinary(...)
```



WA

Linear memory

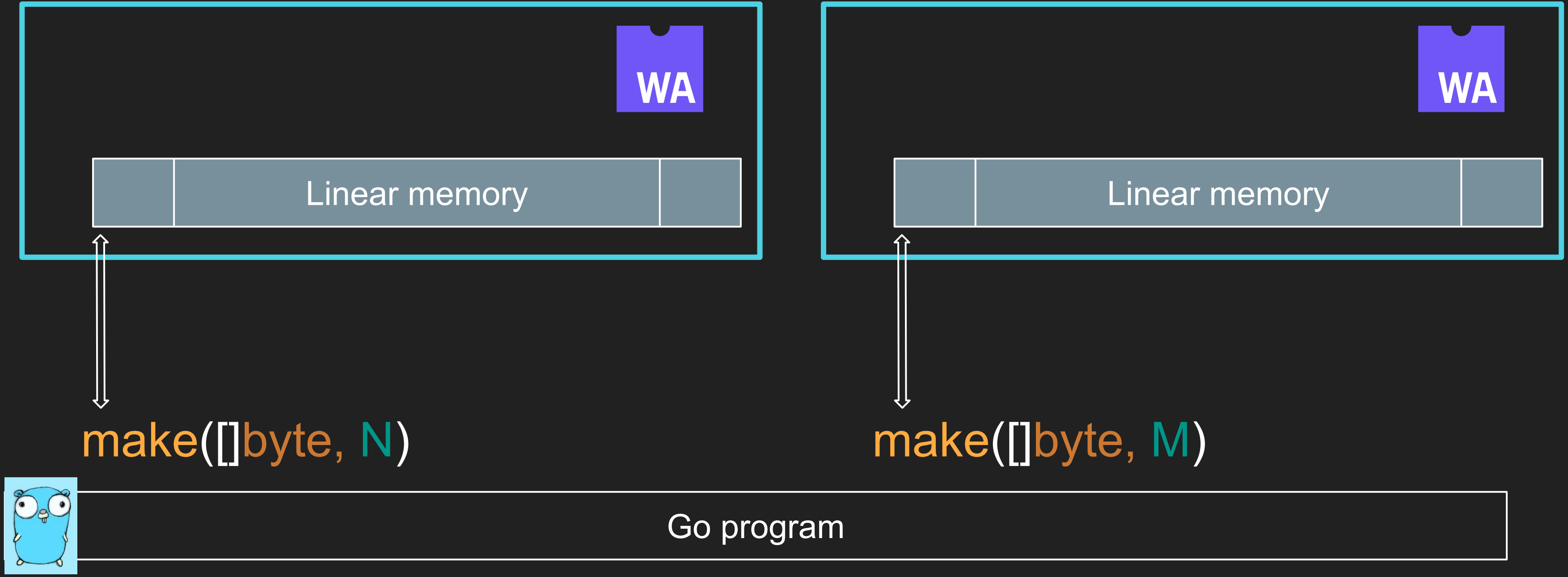
WA

Linear memory

Go program

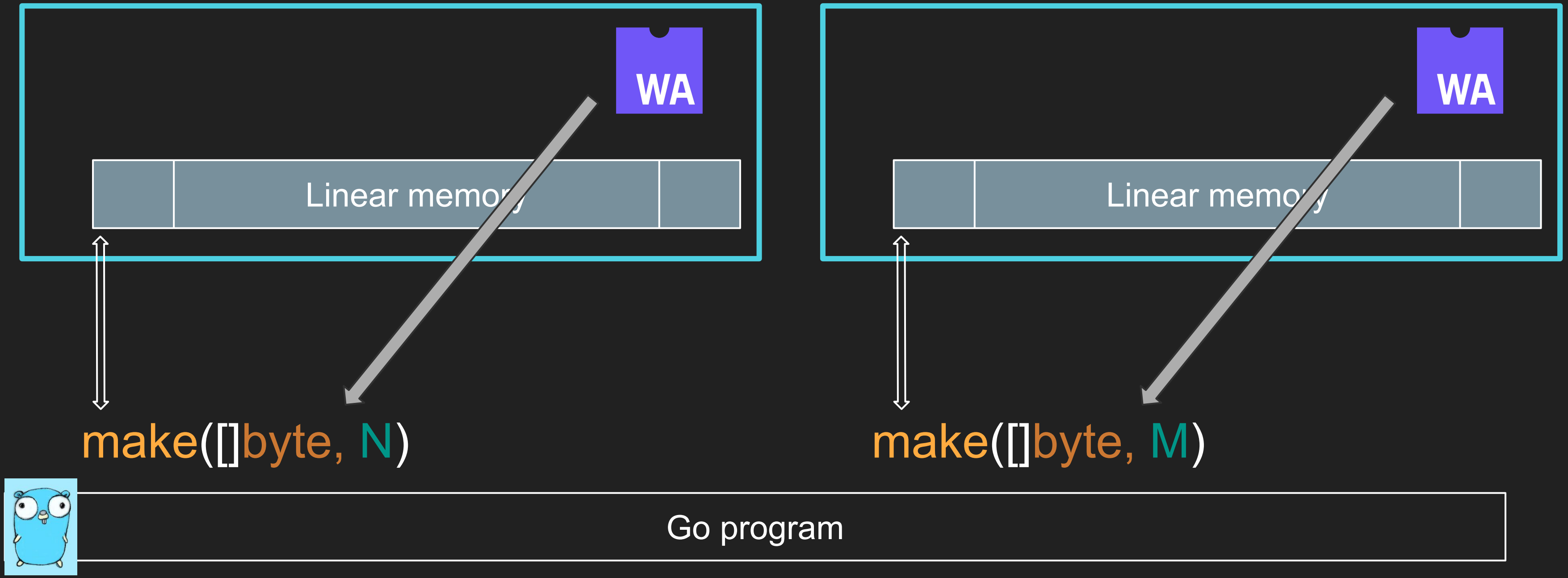
```
// Instantiate a Zig Wasm binary.
zig := r.InstantiateModuleFromBinary(...)
```

```
// Instantiate a Rust Wasm binary.
rust := r.InstantiateModuleFromBinary(...)
```



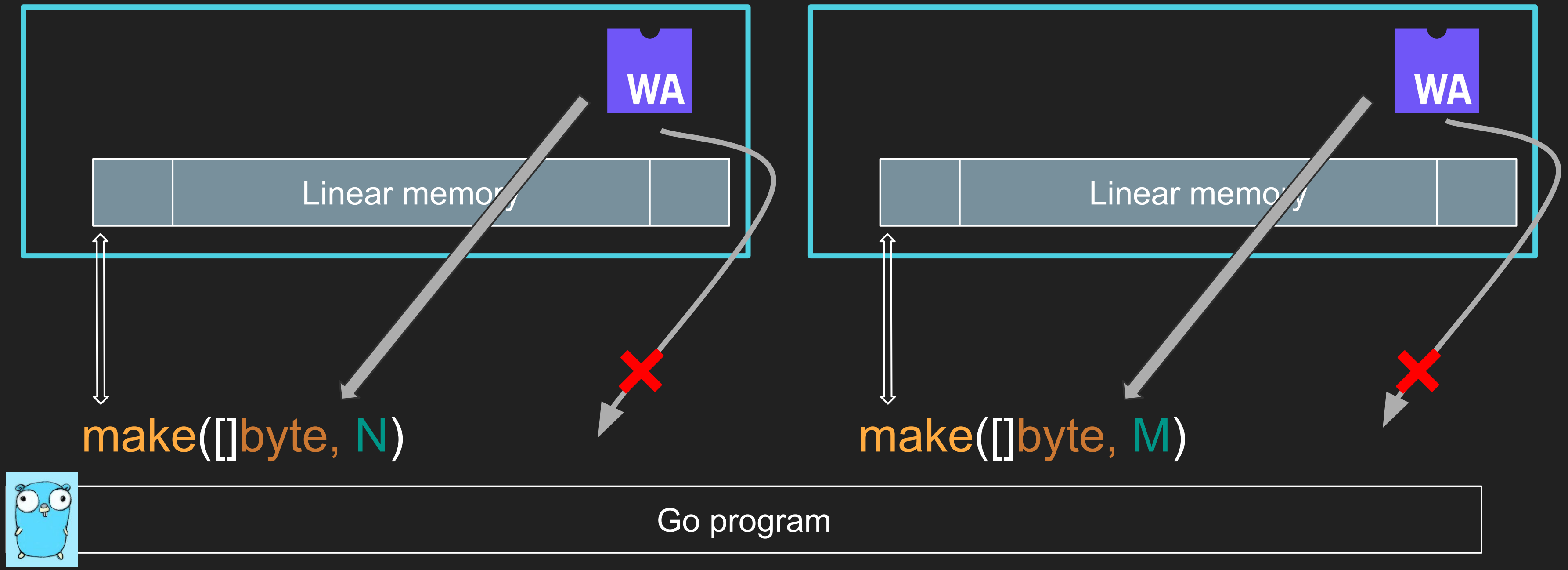
```
// Instantiate a Zig Wasm binary.
zig := r.InstantiateModuleFromBinary(...)
```

```
// Instantiate a Rust Wasm binary.
rust := r.InstantiateModuleFromBinary(...)
```



```
// Instantiate a Zig Wasm binary.
zig := r.InstantiateModuleFromBinary(...)
```

```
// Instantiate a Rust Wasm binary.
rust := r.InstantiateModuleFromBinary(...)
```



WA

Linear memory

WA

Linear memory

make([]byte, N)

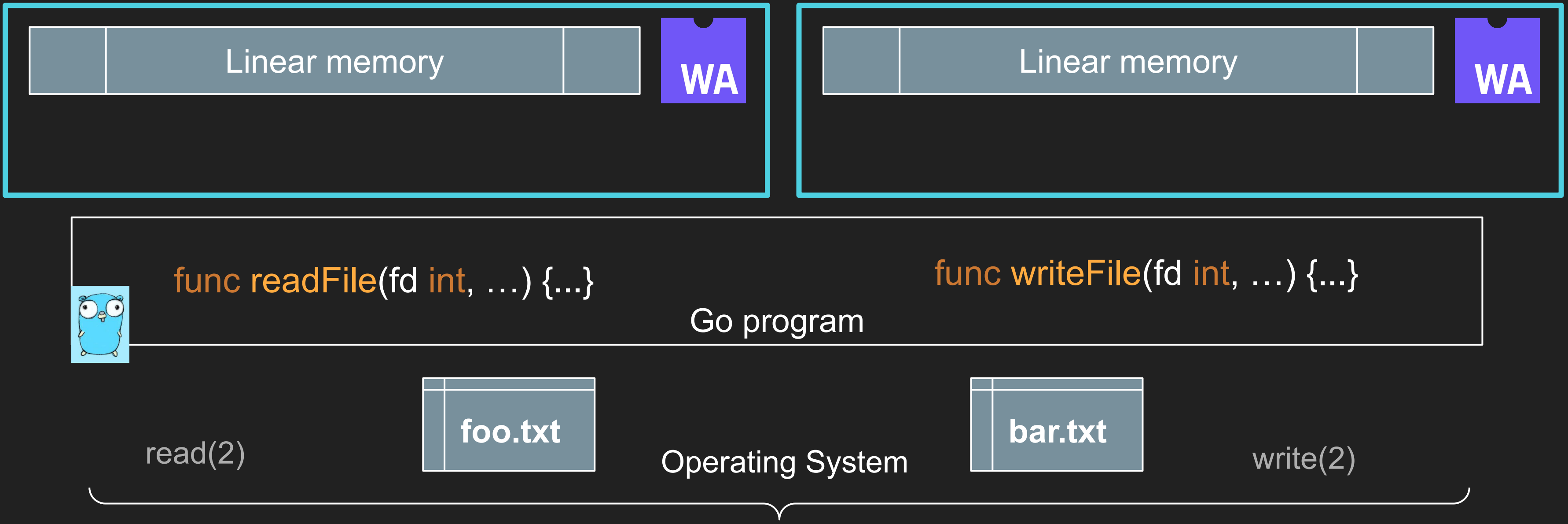
make([]byte, M)

Go program

How it works: system call isolation

```
// Instantiate a Zig Wasm binary.
zig := r.InstantiateModuleFromBinary(...)
```

```
// Instantiate a Rust Wasm binary.
rust := r.InstantiateModuleFromBinary(...)
```

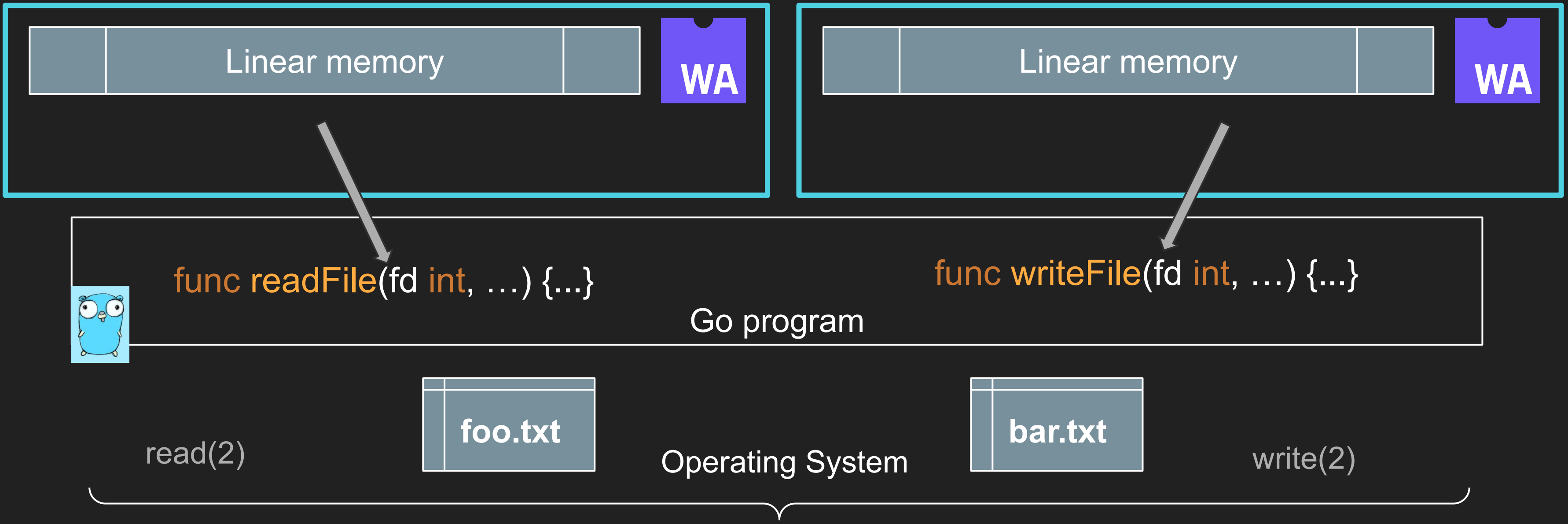


```
func readFile(fd int, …) {…}
```

```
func writeFile(fd int, …) {…}
```

```
// Instantiate a Zig Wasm binary.
zig := r.InstantiateModuleFromBinary(...)
```

```
// Instantiate a Rust Wasm binary.
rust := r.InstantiateModuleFromBinary(...)
```



Linear memory

WA

Linear memory

WA

func readFile(fd int, …) {...}

func writeFile(fd int, …) {...}

Go program

read(2)

foo.txt

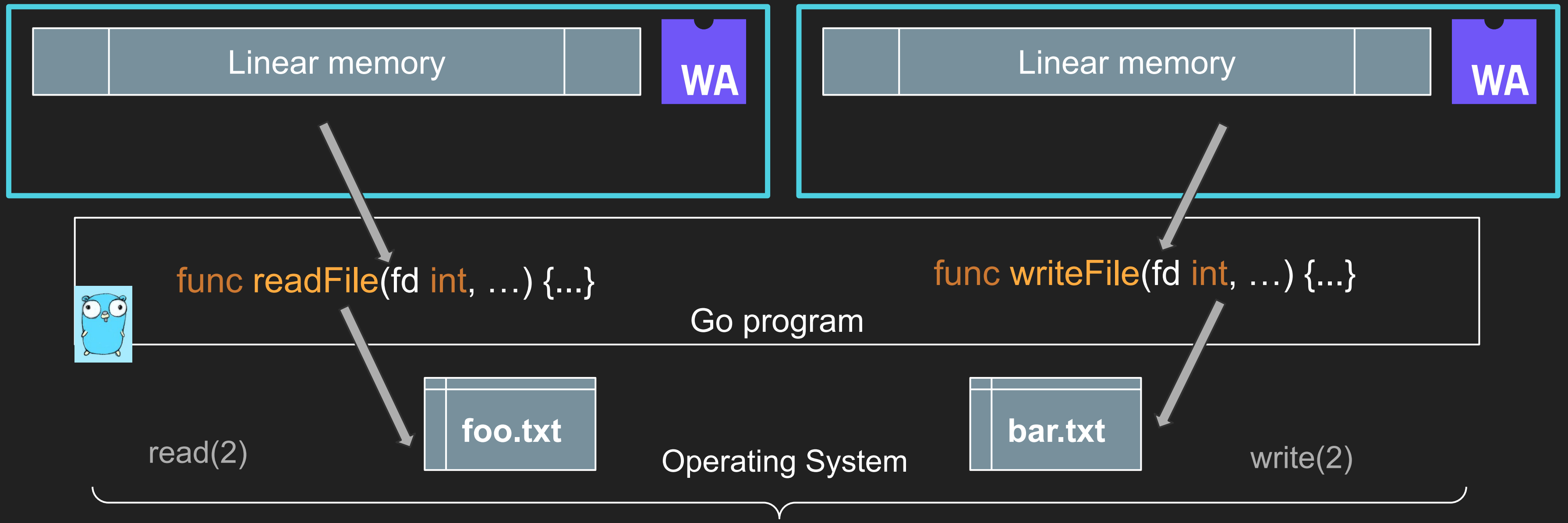
Operating System

bar.txt

write(2)

```
// Instantiate a Zig Wasm binary.
zig := r.InstantiateModuleFromBinary(...)
```

```
// Instantiate a Rust Wasm binary.
rust := r.InstantiateModuleFromBinary(...)
```

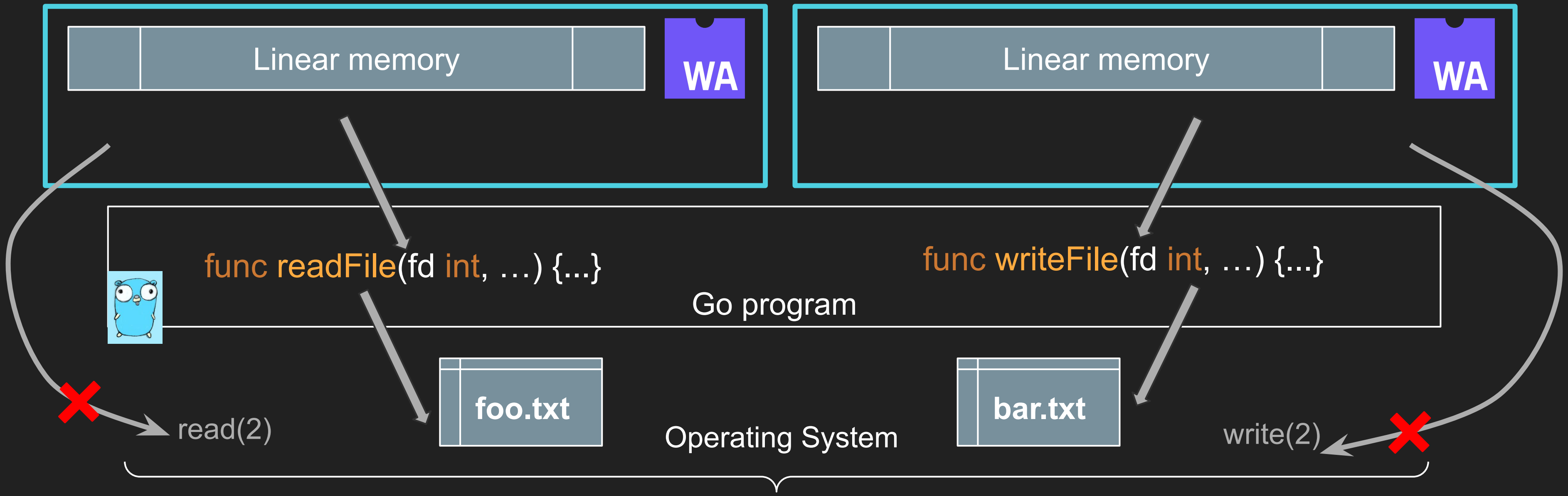


```
func readFile(fd int, …) {...}
```

```
func writeFile(fd int, …) {...}
```

```
// Instantiate a Zig Wasm binary.
zig := r.InstantiateModuleFromBinary(...)
```

```
// Instantiate a Rust Wasm binary.
rust := r.InstantiateModuleFromBinary(...)
```



Linear memory

WA

Linear memory

WA

```
func readFile(fd int, …) {...}
```

```
func writeFile(fd int, …) {...}
```

Go program

foo.txt

bar.txt

read(2)

write(2)

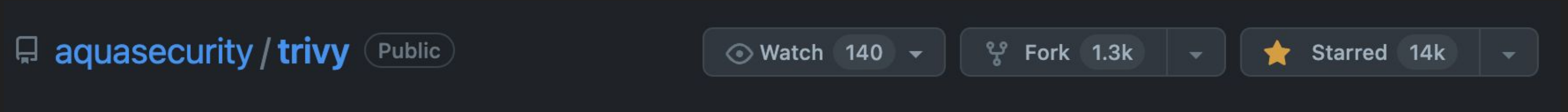
Operating System

System Calls = Go functions

Memory = []byte{...}

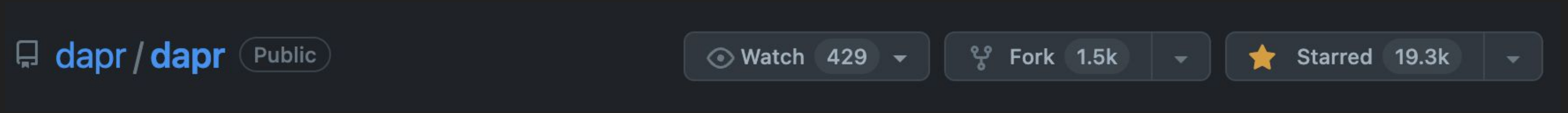
```
// Create a new WebAssembly Runtime.
r := wazero.NewRuntime(ctx)
// Instantiate a Rust Wasm binary.
rust, _ := r.InstantiateModuleFromBinary(ctx, rustBinary)
// Instantiate a Zig Wasm binary.
zig, _ := r.InstantiateModuleFromBinary(ctx, zigBinary)

// Call functions exported by Wasm modules.
... := rust.ExportedFunction("rustFn").Call(ctx, ...)
... := zig.ExportedFunction("zigFn").Call(ctx, ...)
```

Example projects!



aquasecurity / trivy Public

- Trivy: vulnerability scanner
- Can extend scanning logics with Wasm, powered by wazero



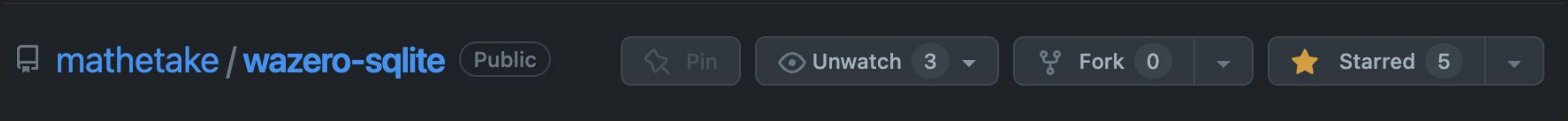
dapr / dapr Public

Watch 429

Fork 1.5k

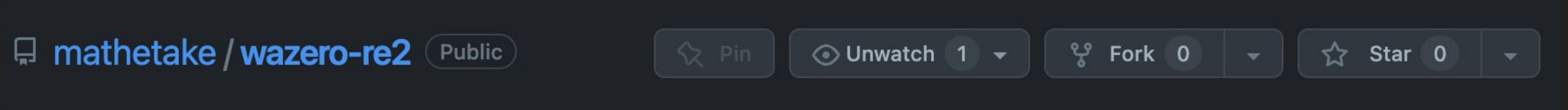
Starred 19.3k

- dapr: portable, serverless application platform
- Can add HTTP middleware in Wasm powered by wazero



mathetake / wazero-sqlite

- Running a Wasm-compiled SQLite inside Go, without CGO
- Possible implementation of CGO-less and sandboxed SQL Driver.



mathetake / wazero-re2

- re2: a fast regular expression engine in C++
- Running Wasm-compiled re2, without CGO
- In some cases, faster that regexp package in the Go std library!

Cons of FFI with wazero vs CGO

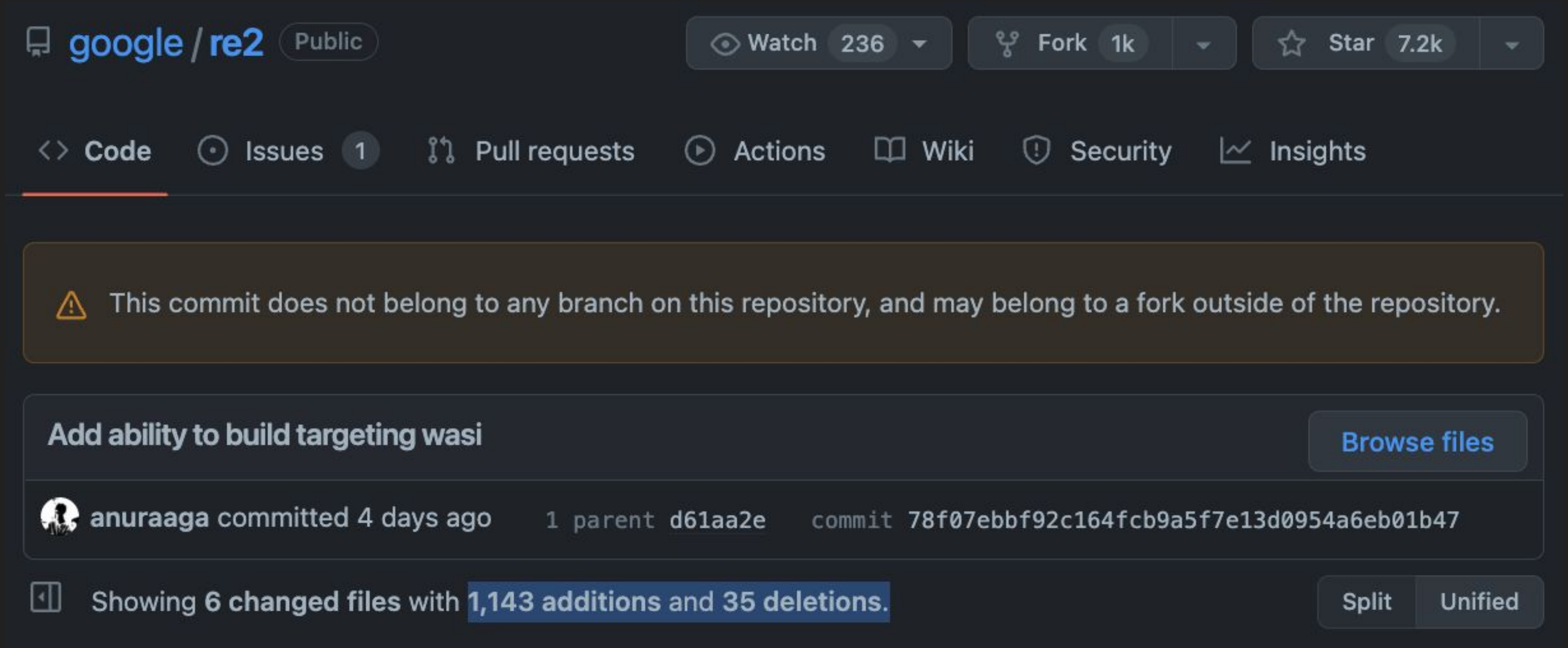
- Performance degradation
  - Wasm == Virtualization
  - Depends on runtime implementation
- Needs to compile your FFI to Wasm
  - Premature ecosystem
  - Refactor in a Wasm-friendly way

Cons of FFI with wazero vs CGO

- Performance degradation
  - Wasm == Virtualization
  - Depends on runtime implementation
- Needs to compile your FFI to Wasm
  - Premature ecosystem
  - Refactor in a Wasm-friendly way

Cons of FFI with wazero vs CGO

- Performance degradation
  - Wasm == Virtualization
  - Depends on runtime implementation
- Needs to compile your FFI to Wasm
  - Premature ecosystem
  - Refactor in a Wasm-friendly way



This commit does not belong to any branch on this repository, and may belong to a fork outside of the repository.

Add ability to build targeting wasi

anuraaga committed 4 days ago 1 parent d61aa2e commit 78f07ebbf92c164fcb9a5f7e13d0954a6eb01b47

Showing 6 changed files with 1,143 additions and 35 deletions.

wazero deep dive…

Q. How correct is the implementation?

Q. How correct is the implementation?

A. 100% compatible with Wasm spec (1.0&2.0)

Q. How is wazero tested?

Q. How is wazero tested?

A. Specification tests & random binary fuzzing

Q. How is the VM implemented?

Q. How is the VM implemented?

A. Two modes: interpreter and AOT compiler

Interpreter mode

- Runs on any platform (GOOS/GOARCH)
- Fast startup time
- Slow execution

```
wazero.NewRuntimeConfigInterpreter()
```

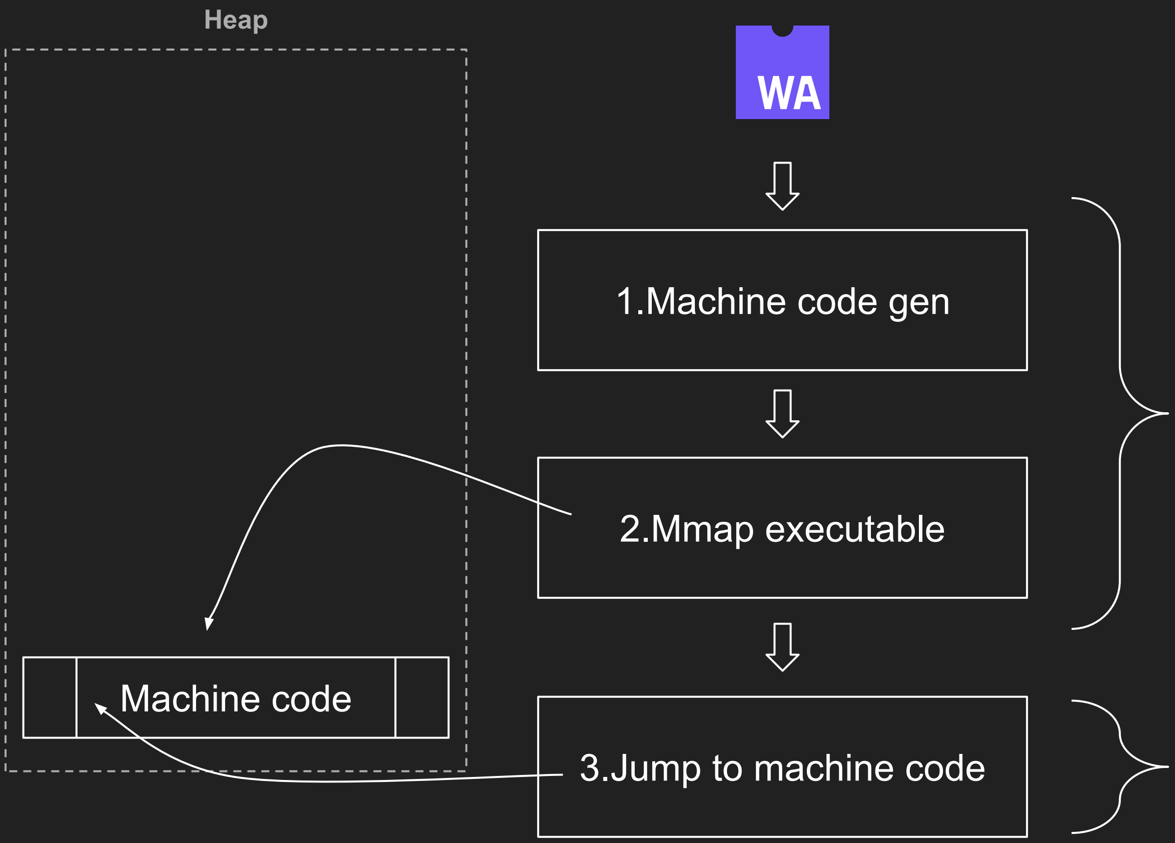
Ahead-Of-Time (AOT) compiler mode

- Runs on {amd64,arm64} x {linux,darwin,windows,freebsd,etc}
- Slow startup time
  - AOT = compile Wasm binary into native machine code before execution
- Fast execution (10x+ faster than interpreter)

```
wazero.NewRuntimeConfigCompiler()
```

How AOT compiler works

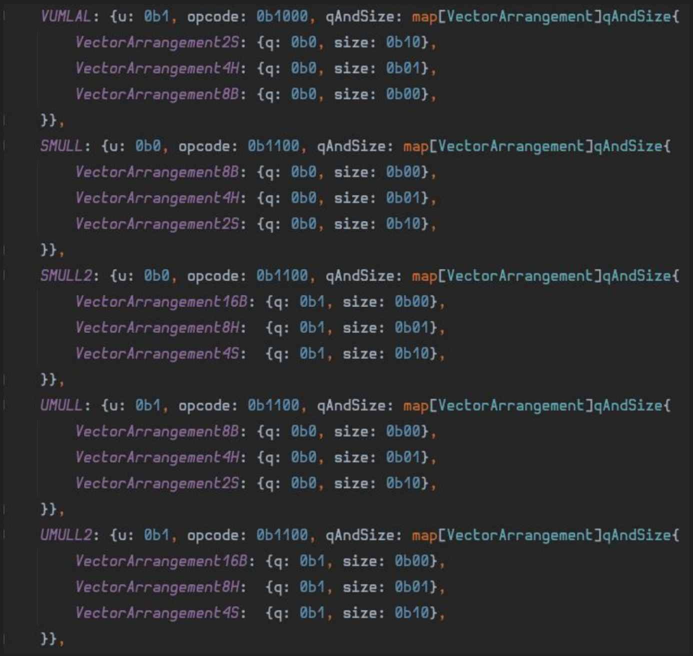
1. Creates native machine code semantically equivalent to Wasm binary
2. mmap the machine code []byte as executable
3. Jumps into the “executable” []byte via a Go Assembly function



```
vm, err := r.InstantiateModuleFromBinary(
    ctx, wasmBinary,
)
```

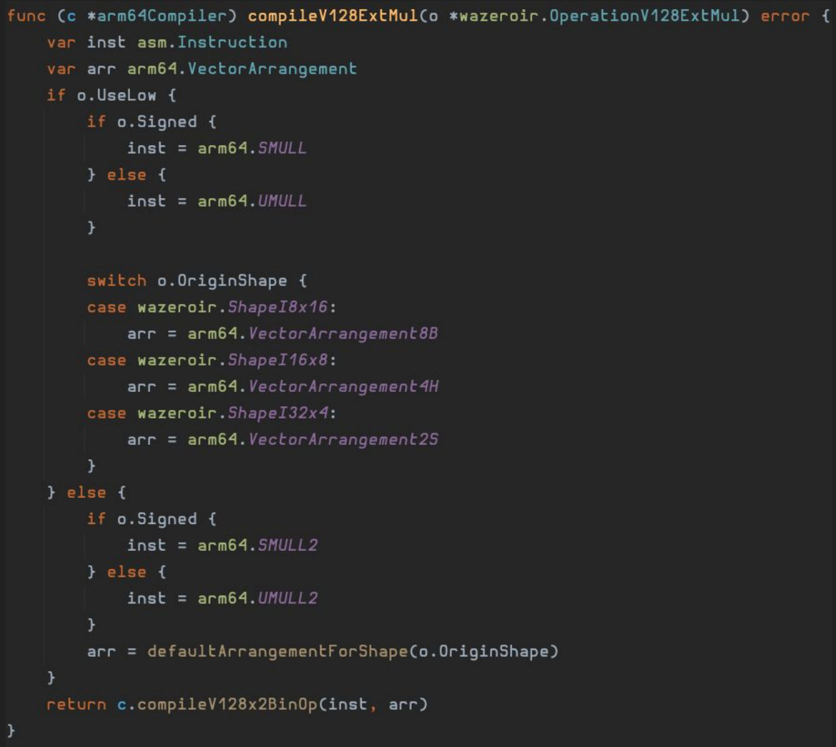
```
… = vm.ExportedFunction("myFunc").Call(...)
```

1. Machine code generation



```
VUMLAL: {u: 0b1, opcode: 0b1000, qAndSize: map[VectorArrangement]qAndSize{
    VectorArrangement2S: {q: 0b0, size: 0b10},
    VectorArrangement4H: {q: 0b0, size: 0b01},
    VectorArrangement8B: {q: 0b0, size: 0b00},
}},
SMULL: {u: 0b0, opcode: 0b1100, qAndSize: map[VectorArrangement]qAndSize{
    VectorArrangement8B: {q: 0b0, size: 0b00},
    VectorArrangement4H: {q: 0b0, size: 0b01},
    VectorArrangement2S: {q: 0b0, size: 0b10},
}},
SMULL2: {u: 0b0, opcode: 0b1100, qAndSize: map[VectorArrangement]qAndSize{
    VectorArrangement16B: {q: 0b1, size: 0b00},
    VectorArrangement8H:  {q: 0b1, size: 0b01},
    VectorArrangement4S:  {q: 0b1, size: 0b10},
}},
UMULL: {u: 0b1, opcode: 0b1100, qAndSize: map[VectorArrangement]qAndSize{
    VectorArrangement8B: {q: 0b0, size: 0b00},
    VectorArrangement4H: {q: 0b0, size: 0b01},
    VectorArrangement2S: {q: 0b0, size: 0b10},
}},
UMULL2: {u: 0b1, opcode: 0b1100, qAndSize: map[VectorArrangement]qAndSize{
    VectorArrangement16B: {q: 0b1, size: 0b00},
    VectorArrangement8H:  {q: 0b1, size: 0b01},
    VectorArrangement4S:  {q: 0b1, size: 0b10},
}},
```

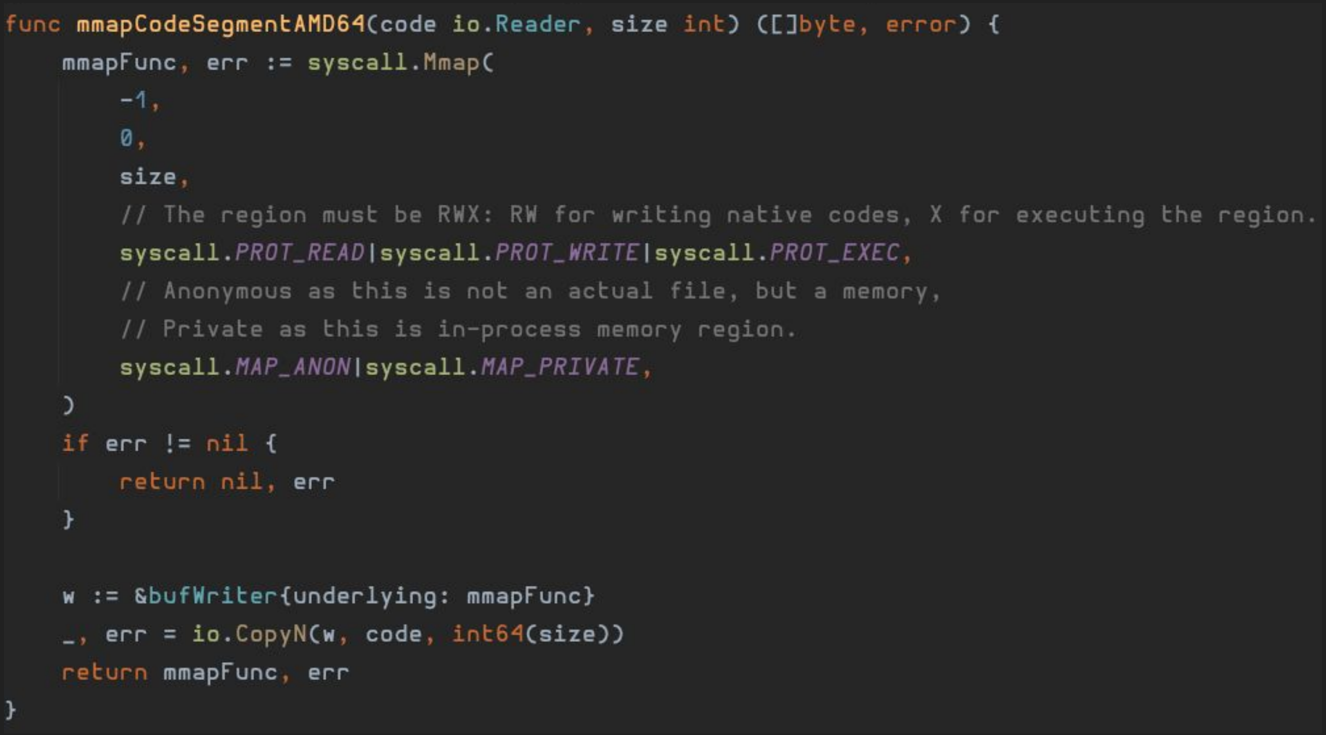
Assembler



```
func (c *arm64Compiler) compileV128ExtMul(o *wazeroir.OperationV128ExtMul) error {
    var inst asm.Instruction
    var arr arm64.VectorArrangement
    if o.UseLow {
        if o.Signed {
            inst = arm64.SMULL
        } else {
            inst = arm64.UMULL
        }

        switch o.OriginShape {
        case wazeroir.ShapeI8x16:
            arr = arm64.VectorArrangement8B
        case wazeroir.ShapeI16x8:
            arr = arm64.VectorArrangement4H
        case wazeroir.ShapeI32x4:
            arr = arm64.VectorArrangement2S
        }
    } else {
        if o.Signed {
            inst = arm64.SMULL2
        } else {
            inst = arm64.UMULL2
        }
        arr = defaultArrangementForShape(o.OriginShape)
    }
    return c.compileV128x2BinOp(inst, arr)
}
```

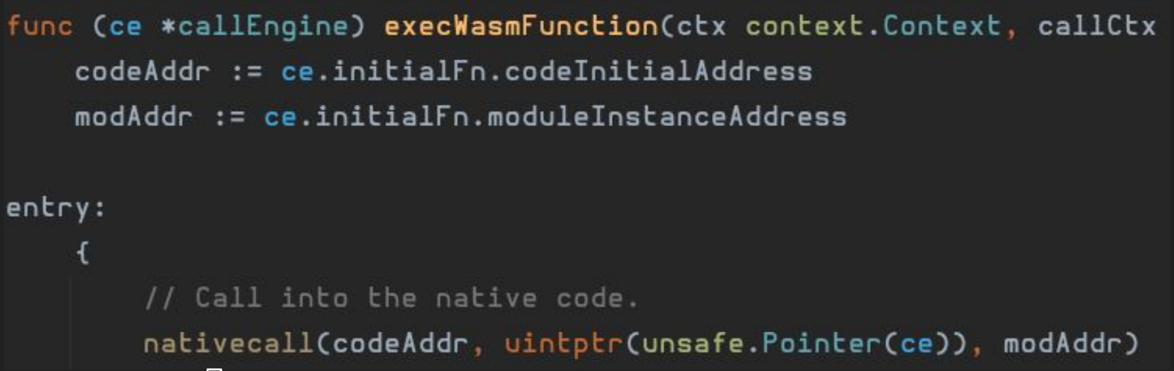
Code generation

2. Mmap machine code as executable

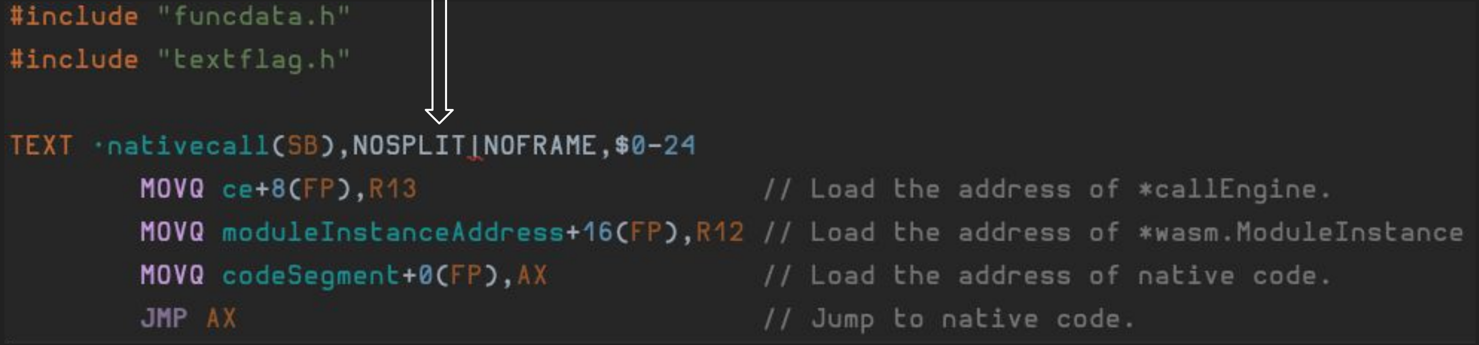


```go
func mmapCodeSegmentAMD64(code io.Reader, size int) ([]byte, error) {
    mmapFunc, err := syscall.Mmap(
        -1,
        0,
        size,
        // The region must be RWX: RW for writing native codes, X for executing the region.
        syscall.PROT_READ|syscall.PROT_WRITE|syscall.PROT_EXEC,
        // Anonymous as this is not an actual file, but a memory,
        // Private as this is in-process memory region.
        syscall.MAP_ANON|syscall.MAP_PRIVATE,
    )
    if err != nil {
        return nil, err
    }

    w := &bufWriter{underlying: mmapFunc}
    _, err = io.CopyN(w, code, int64(size))
    return mmapFunc, err
}
```

3. Jump into machine code



```go
func (ce *callEngine) execWasmFunction(ctx context.Context, callCtx
    codeAddr := ce.initialFn.codeInitialAddress
    modAddr := ce.initialFn.moduleInstanceAddress

entry:
    {
        // Call into the native code.
        nativecall(codeAddr, uintptr(unsafe.Pointer(ce)), modAddr)
```



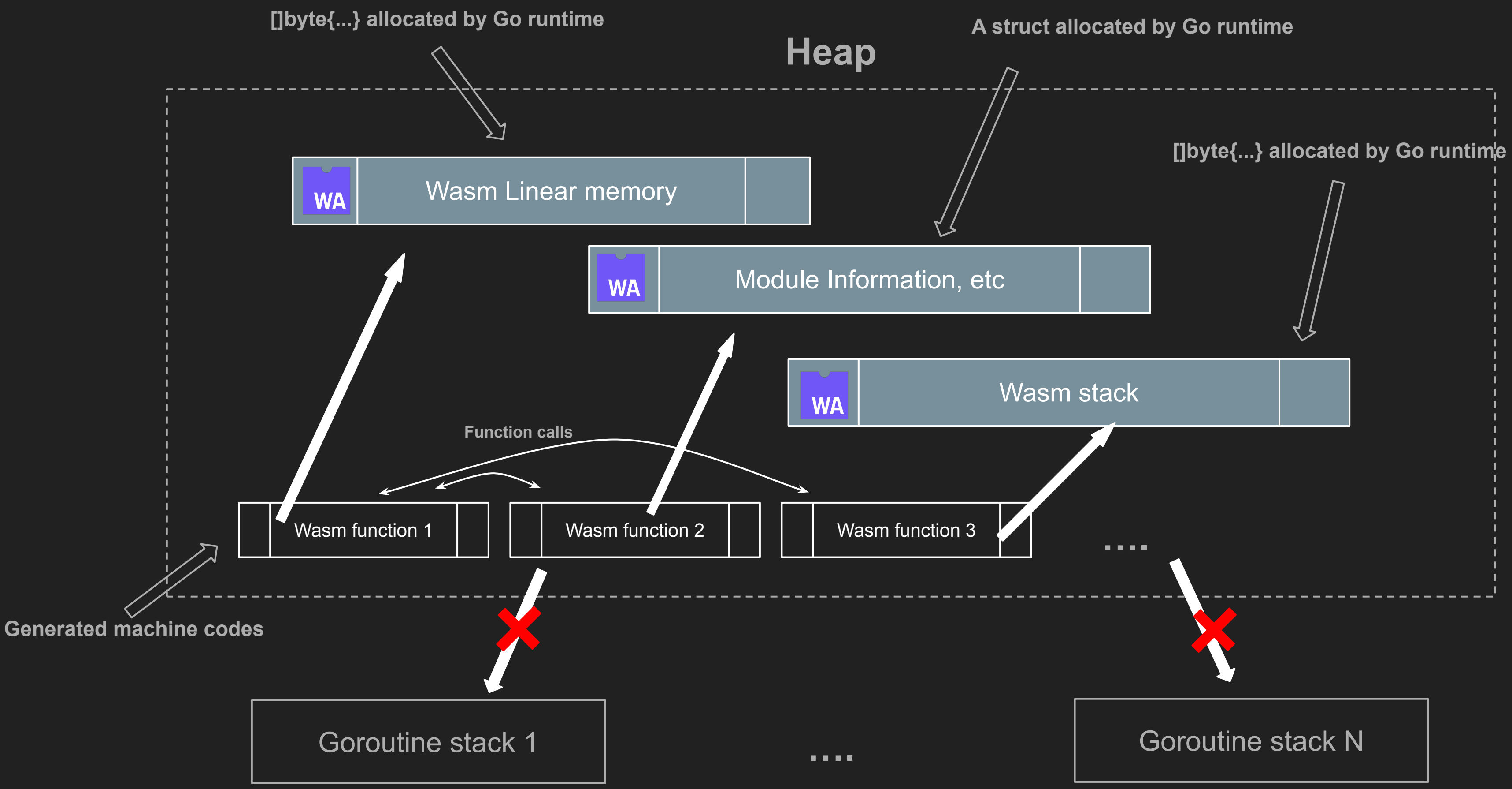
```
#include "funcdata.h"
#include "textflag.h"

TEXT ·nativecall(SB),NOSPLIT|NOFRAME,$0-24
        MOVQ ce+8(FP),R13                        // Load the address of *callEngine.
        MOVQ moduleInstanceAddress+16(FP),R12    // Load the address of *wasm.ModuleInstance
        MOVQ codeSegment+0(FP),AX                // Load the address of native code.
        JMP AX                                   // Jump to native code.
```

Challenges in AOT compiler implementation

- Do not modify Goroutine-stack! (e.g. “call” instruction)
- Do not access Goroutine-stack allocated variable from machine code
- Debugging is extremely difficult
- Single pass compiler: optimizations are TODOs

Challenges in AOT compiler implementation

- Do not modify Goroutine-stack! (e.g. “call” instruction)
- Do not access Goroutine-stack allocated variable from machine code
- Debugging is extremely difficult
- Single pass compiler: optimizations are TODOs


[]byte{...} allocated by Go runtime
Heap
A struct allocated by Go runtime
[]byte{...} allocated by Go runtime
WA
Wasm Linear memory
WA
Module Information, etc
WA
Wasm stack
Function calls
Wasm function 1
Wasm function 2
Wasm function 3
....
Generated machine codes
Goroutine stack 1
....
Goroutine stack N

Challenges in AOT compiler implementation

- Do not modify Goroutine-stack! (e.g. “call” instruction)
- Do not access Goroutine-stack allocated variable from machine code
- Debugging is extremely difficult
- Naive single pass compiler: optimizations are TODOs

Wrap up!

- FFI == Calling non-Go functions from Go
- CGO works, but has some issues
- CGO-less FFI is possible with wazero+WebAssembly
- wazero is written in pure Go, zero dependency!

Thank you!

Twitter,GitHub: @mathetake
Gopher Slack: #wazero