



15 Aug 2017

RUSTGO: CALLING RUST FROM GO WITH NEAR-ZERO OVERHEAD

[русский]

Go has good support for calling into assembly, and a lot of the fast cryptographic code in the stdlib is carefully optimized assembly, bringing speedups of over 20 times.

However, writing assembly code is hard, reviewing it is possibly harder, and cryptography is unforgiving. Wouldn't it be nice if we could write these hot functions in a higher level language?

This post is the story of a slightly-less-than-sane experiment to call Rust code from Go fast enough to replace assembly. No need to know Rust, or compiler internals, but knowing what a linker is would help.

Hello, readers from the future! In the seven years since writing this a couple things changed, but note that this was always a fun technical exploration, not a production guide.

If you're looking for *serious* ways to call Rust from Go, know that cgo these days is faster and still getting faster. There are also some experimental cgo-less alternatives, like purego and wazero.

Still, experimenting with this stuff is fascinating. I toyed for years with the idea of [using Wasm for cross-platform Go FFI](#) withg better DX. I admit I even used a technique similar to this in the standard library, to [invoke the macOS X.509 verifier](#).

As for me, I gave [a talk](#) about this at a Go conference wearing a (freshly printed) *Rust Evangelism Strike Force* t-shirt. That kind of landed me a job on the Go team at Google, which I left to [become an independent Go maintainer](#).

The fight against cryptographic assembly continues, but now with [strict policies](#) and [dedicated generators](#). Just yesterday I [livestreamed a benchmarking session to replace some P-256 curve assembly](#) with formally verified Go code.

[Subscribe](#)  | [Feed](#)  | [Bluesky](#)  | [Mastodon](#) 

Why Rust

I'll be upfront: I don't know Rust, and don't feel compelled to do my day-to-day programming in it. However, I know Rust is a very tweakable and optimizable language, while still more readable than assembly. (After all, everything is more readable than assembly!)

Go strives to find defaults that are good for its core use cases, and only accepts features that are fast enough to be enabled by default, in a constant and successful fight against knobs. I love it for that. But for what we are doing today we need a language that won't flinch when asked to generate stack-only functions with manually hinted away safety checks.

So if there's a language that we might be able to constrain enough to behave like assembly, and to optimize enough to be as useful as assembly, it might be Rust.

Finally, Rust is safe, actively developed, and not least, there's already a good ecosystem of high-performance Rust cryptography code to tap into.

Why not cgo

Go has a [Foreign Function Interface](#), `cgo`. `cgo` allows Go programs to call C functions in the most natural way possible—which is unfortunately not very natural at all. (I know [more than I'd like to about cgo](#), and I can tell you [it's not fun](#).)

By using the C ABI as lingua franca of FFIs, we can call anything from anything: Rust can compile into a library exposing the C ABI, and cgo can use that. It's awkward, but it works.

We can even use reverse-cgo to build Go into a C library and call it from random languages, like I did with Python as a stunt. (It was a stunt folks, stop taking me seriously.)

But cgo does a lot of things to enable that bit of Go naturalness it provides: it will setup a whole stack for C to live in, it makes defer calls to prepare for a panic in a Go callback... this ~~could be~~ will be a whole post of its own.

As a result, the performance cost of each cgo call is way too high for the use case we are thinking about—*small hot functions*.

Linking it together

So here's the idea: if we have Rust code that is as constrained as assembly, we should be able to use it **just like assembly**, and call straight into it. Maybe with a thin layer of glue.

We don't have to work at the IR level: the Go compiler converts both code and high-level assembly into machine code before linking since Go 1.3.

This is confirmed by the existence of "external linking", where the system linker is used to put together a Go program. It's how cgo works, too: it compiles C with the C compiler, Go with the Go compiler, and links it all together with `clang` or `gcc`. We can even pass flags to the linker with `CGO_LDFLAGS`.

Underneath all the safety features of cgo, we surely find a cross-language function call, after all.

It would be nice if we could figure out how to do this without patching the compiler, though. First, let's figure out how to link a Go program with a Rust archive.

I could not find a decent way to link against a foreign blob with `go build` (why should there be one?) except using `#cgo` directives. However, invoking cgo makes .s files go to the C compiler instead of the Go one, and my friends, we *will* need Go assembly.

Thankfully `go/build` is nothing but a frontend! Go offers a set of low level tools to compile and link programs, `go build` just collects files and invokes those tools. We can follow what it does by using the `-x` flag.

I built this small Makefile by following a `-x -ldflags "-v -linkmode=external '-extldflags=-v'"` invocation of a `cgo` build.

```
rustgo: rustgo.a
    go tool link -o rustgo -extld clang -buildmode exe -buildid b01dca11ab1e -link

rustgo.a: hello.go hello.o
    go tool compile -o rustgo.a -p main -buildid b01dca11ab1e -pack hello.go
    go tool pack r rustgo.a hello.o

hello.o: hello.s
    go tool asm -I "$(shell go env GOROOT)/pkg/include" -D G00S_darwin -D G0ARCH_a
```

This compiles a simple main package composed of a Go file (`hello.go`) and a Go assembly file (`hello.s`).

Now, if we want to link in a Rust object we first build it as a static library...

```
libhello.a: hello.rs
    rustc -g -O --crate-type staticlib hello.rs
```

... and then just tell the external linker to link it together.

```
rustgo: rustgo.a libhello.a
    go tool link -o rustgo -extld clang -buildmode exe -buildid b01dca11ab1e -link
```

```
$ make
go tool asm -I "/usr/local/Cellar/go/1.8.1_1/libexec/pkg/include" -D G00S_darwin -D G0
go tool compile -o rustgo.a -p main -buildid b01dca11ab1e -pack hello.go
go tool pack r rustgo.a hello.o
rustc --crate-type staticlib hello.rs
note: link against the following native artifacts when linking against this static lib
note: the order and any duplication can be significant on some platforms, and so may n
note: library: System
note: library: c
note: library: m

go tool link -o rustgo -extld clang -buildmode exe -buildid b01dca11ab1e -linkmode ext
HEADER = -H1 -T0x1001000 -D0x0 -R0x1000
searching for runtime.a in /usr/local/Cellar/go/1.8.1_1/libexec/pkg/darwin_amd64/runti
```

```

searching for runtime/cgo.a in /usr/local/Cellar/go/1.8.1_1/libexec/pkg/darwin_amd64/r
0.00 deadcode
0.00 pclntab=166785 bytes, funcdata total 17079 bytes
0.01 dodata
0.01 symsize = 0
0.01 symsize = 0
0.01 reloc
0.01 dwarf
0.02 symsize = 0
0.02 reloc
0.02 asmb
0.02 codeblk
0.03 datblk
0.03 sym
0.03 headr
0.06 host link: "clang" "-m64" "-gdwarf-2" "-Wl,-headerpad,1144" "-Wl,-no_pie" "-Wl,-
0.34 cpu time
12641 symbols
5764 liveness data

```

Jumping into Rust

Alright, so we linked it, but the symbols are not going to do anything just by sitting next to each other. We need to somehow call the Rust function from our Go code.

We know how to call a Go function from Go. In assembly the same call looks like `CALL hello(SB)`, where SB is a virtual register all global symbols are relative to.

If we want to call an assembly function from Go we make the compiler aware of its existence like a C header, by writing `func hello()` without a function body.

I tried all combinations of the above to call an external (Rust) function, but they all complained that they couldn't find either the symbol name, or the function body.

But cgo, which at the end of the day is just a giant code generator, somehow manages to eventually invoke that foreign function! How?

I stumbled upon [the answer](#) a couple days later.

```

//go:cgo_import_static _cgoPREFIX_Cfunc__Cmalloc
//go:linkname __cgofn__cgoPREFIX_Cfunc__Cmalloc _cgoPREFIX_Cfunc__Cmalloc
var __cgofn__cgoPREFIX_Cfunc__Cmalloc byte
var _cgoPREFIX_Cfunc__Cmalloc = unsafe.Pointer(&__cgofn__cgoPREFIX_Cfunc__Cmalloc)

```

That looks like an interesting pragma! `//go:linkname` just creates a symbol alias in the local scope (which can be used to call private functions!), and I'm pretty sure the `byte` trick is only cleverness to have something to take the address of, but `//go:cgo_import_static` ... this imports an external symbol!

2024 note: `cgo_import_static` is not usable anymore, but there are alternative strategies.

Armed with this new tool and the Makefile above, we have a chance to invoke this Rust function (`hello.rs`)

```
#[no_mangle]
pub extern fn hello() {
    println!("Hello, Rust!");
}
```

(The no-mangle-pub-extern incantation is from this tutorial.)

from this Go program (`hello.go`)

```
package main

//go:cgo_import_static hello

func trampoline()

func main() {
    println("Hello, Go!")
    trampoline()
}
```

with the help of this assembly snippet. (`hello.s`)

```
TEXT ·trampoline(SB), 0, $2048
    JMP hello(SB)
    RET
```

`CALL` was a bit too smart to work, but using a simple `JMP` ...

```
Hello, Go!
Hello, Rust!
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x0]
```



Well, it crashes when it tries to return. Also that `$2048` value is the whole stack size Rust is allowed (if it's even putting the stack in the right place), and don't ask me what happens if Rust tries to touch a heap... but hell, I'm surprised it works at all!

Calling conventions

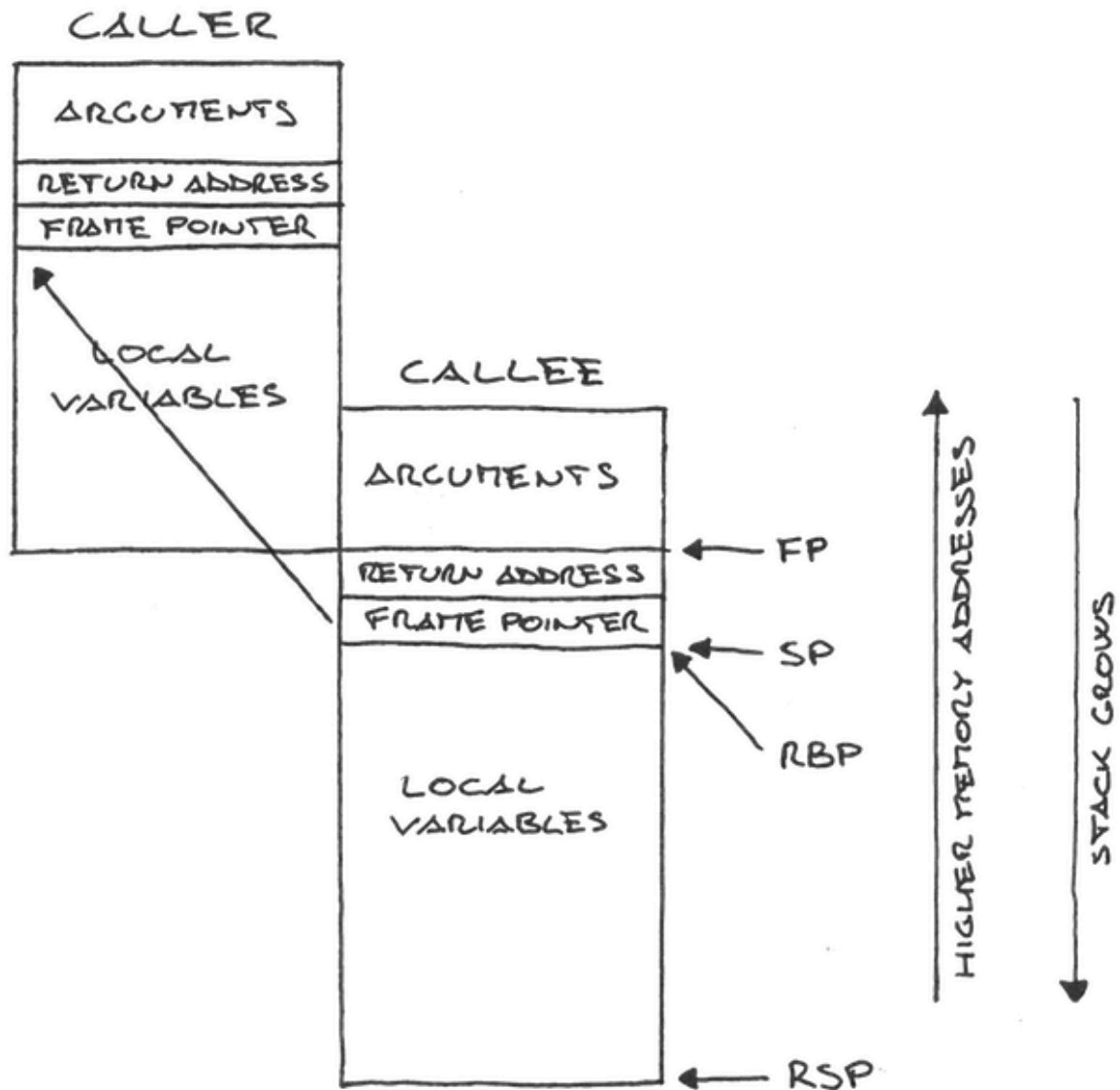
Now, to make it return cleanly, and take some arguments, we need to look more closely at the Go and Rust calling conventions. A calling convention defines where arguments and return values sit across function calls.

The Go calling convention is described here and here. For Rust we'll look at the default for FFI, which is the standard C calling convention.

To keep going we're going to need a debugger. (LLDB supports Go, but breakpoints are somehow broken on macOS, so I had to play inside a privileged Docker container.)

![Zelda dangerous to go alone](https://assets.buttndown.email/images/6bcc097d-3473-436f-99ea-30849a127584.png)

The Go calling convention



The Go calling convention is mostly undocumented, but we'll need to understand it to proceed, so here is what we can learn from a disassembly (amd64 specific). Let's look at a very simple function.

```
// func foo(x, y uint64) uint64
TEXT ·foo(SB), 0, $256-24
    MOVQ x+0(FP), DX
    MOVQ DX, ret+16(FP)
    RET
```

`foo` has 256 (0x100) bytes of local frame, 16 bytes of arguments, 8 bytes of return value, and it returns its first argument.

```
func main() {
    foo(0xf0f0f0f0f0f0f0f0, 0x5555555555555555)
```



```
rustgo[0x49d785]: movabsq $-0xf0f0f0f0f0f0f10, %rax
rustgo[0x49d78f]: movq    %rax, (%rsp)
rustgo[0x49d793]: movabsq $0x5555555555555555, %rax
rustgo[0x49d79d]: movq    %rax, 0x8(%rsp)
rustgo[0x49d7a2]: callq   0x49d8a0                ; main.foo at hello.s:14
```

The caller, seen above, does very little: it places the arguments on the stack in reverse order, at the bottom of its own frame (`rsp` to `16(rsp)` , remember that the stack grows down) and executes `CALL` . The `CALL` will push the return pointer to the stack and jump. There's no caller cleanup, just a plain `RET` .

Notice that `rsp` is fixed, and we have `movq s`, not `push s`.

```
rustgo`main.foo at hello.s:14:
rustgo[0x49d8a0]: movq    %fs:-0x8, %rcx
rustgo[0x49d8a9]: leaq    -0x88(%rsp), %rax
rustgo[0x49d8b1]: cmpq    0x10(%rcx), %rax
rustgo[0x49d8b5]: jbe     0x49d8ee                ; main.foo + 78 at hello.s:14
rustgo[0x49d8b6]: [...]
rustgo[0x49d8ee]: callq   0x49d5d0                ; runtime.morestack_noctxt at asm_
rustgo[0x49d8f3]: jmp     0x49d8a0                ; main.foo at hello.s:14
```

The first 4 and last 2 instructions of the function are checking if there is enough space for the stack, and if not calling `runtime.morestack` . They are probably skipped for `NOSPLIT` functions.

```
rustgo[0x49d8b7]: subq    $0x108, %rsp
rustgo[0x49d8b8]: [...]
rustgo[0x49d8e6]: addq    $0x108, %rsp
rustgo[0x49d8ed]: retq
```

Then there's the `rsp` management, which subtracts `0x108`, making space for the entire `0x100` bytes of frame in one go, and the 8 bytes of frame pointer. So `rsp` points to the bottom (the end) of the function frame, and is callee managed. Before returning, `rsp` is returned to where it was (just past the return pointer).

```
rustgo[0x49d8be]: movq    %rbp, 0x100(%rsp)
rustgo[0x49d8c6]: leaq    0x100(%rsp), %rbp
rustgo[0x49d8c7]: [...]
rustgo[0x49d8de]: movq    0x100(%rsp), %rbp
```

Finally the frame pointer, which is effectively pushed to the stack just after the return pointer, and updated at `rbp`. So `rbp` is also callee saved, and should be updated to point at where the caller's `rbp` is stored to enable stack trace unrolling.

```
rustgo[0x49d8ce]: movq    0x110(%rsp), %rdx
rustgo[0x49d8d6]: movq    %rdx, 0x120(%rsp)
```

Finally, from the body itself we learn that return values go just above the arguments.

Virtual registers

The Go docs say that `SP` and `FP` are virtual registers, not just aliases of `rsp` and `rbp`.

Indeed, when accessing `SP` from Go assembly, the offsets are adjusted relative to the real `rsp` so that `SP` points to the top, not the bottom, of the frame. That's convenient because it means not having to change all offsets when changing the frame size, but it's just syntactic sugar. Naked access to the register (like `MOVQ SP, DX`) accesses `rsp` directly.

The `FP` virtual register is simply an adjusted offset over `rsp`, too. It points to the bottom of the caller frame, where arguments are, and there's no direct access.

Note: Go maintains `rbp` and frame pointers to help debugging, but then uses a fixed `rsp` and `omit-stack-pointer`-style `rsp` offsets for the virtual `FP`. You can learn more about frame pointers and not using them from [this Adam Langley blog post](#).

The C calling convention

"sysv64", the default C calling convention on x86-64, is quite different:

- The arguments are passed via registers: `RDI`, `RSI`, `RDY`, `RCX`, `R8`, and `R9`.
- The return value goes to `RAX`.
- Some registers are callee-saved: `RBX`, `RBX`, and `R12–R15`. * We care little about this, since in Go all registers are caller-saved.
- The stack must be aligned to 16-bytes. * (I think this is why `JMP` worked and `CALL` didn't, we failed to align the stack!)

Frame pointers work the same way (and are generated by `rustc` with `-g`).

Gluing them together

Building a simple trampoline between the two conventions won't be hard. We can also look at [asmcgocall](#) for inspiration, since it does approximately the same job, but for cgo.

We need to remember that we want the Rust function to use the stack space of our assembly function, since Go ensured for us that it's present. To do that, we have to rollback `rsp` from the end of the stack.

```
package main

//go:cgo_import_static increment
func trampoline(arg uint64) uint64

func main() {
    println(trampoline(41))
}
```



```
TEXT ·trampoline(SB), 0, $2048-16
    MOVQ arg+0(FP), DI // Load the argument before messing with SP
    MOVQ SP, BX        // Save SP in a callee-saved registry
    ADDQ $2048, SP      // Rollback SP to reuse this function's frame
    ANDQ $~15, SP      // Align the stack to 16-bytes
    CALL increment(SB)
    MOVQ BX, SP        // Restore SP
    MOVQ AX, ret+8(FP) // Place the return value on the stack
    RET
```



```
#[no_mangle]
pub extern fn increment(a: u64) -> u64 {
    return a + 1;
}
```

CALL on macOS

`CALL` didn't quite work on macOS. For some reason, there the function call was replaced with an intermediate call to `_cgo_thread_start`, which is not that incredible considering we are using something called `cgo_import_static` and that `CALL` is virtual in Go assembly.

```
callq 0x40a27cd ; x_cgo_thread_start + 29
```

We can bypass that “helper” by using the full `//go:linkname` incantation we found in the standard library to take a pointer to the function, and then calling the function pointer, like this.

```
import _ "unsafe"

//go:cgo_import_static increment
//go:linkname increment increment
var increment uintptr
var _increment = &increment
```

```
MOVQ ·_increment(SB), AX
CALL AX
```

Is it fast?

The point of this whole exercise is to be able to call Rust instead of assembly for cryptographic operations (and to have fun). So a rustgo call will have to be almost as fast as an assembly call to be useful.

Benchmark time!

We’ll compare incrementing a uint64 inline, with a `//go:noinline` function, with the rustgo call above, and with a cgo call to the exact same Rust function.

Rust was compiled with `-g -0`, and the benchmarks were run on macOS on a 2.9GHz Intel Core i5.

name	time/op
CallOverhead/Inline	1.72ns ± 3%
CallOverhead/Go	4.60ns ± 2%
CallOverhead/rustgo	5.11ns ± 4%
CallOverhead/cgo	73.6ns ± 0%

rustgo is 11% slower than a Go function call, and almost 15 times faster than cgo!

The performance is even better when run on Linux without the function pointer workaround, with only a 2% overhead.

name	time/op
CallOverhead/Inline	1.67ns ± 2%
CallOverhead/Go	4.49ns ± 3%

```
CallOverhead/rustgo 4.58ns ± 3%
CallOverhead/cgo     69.4ns ± 0%
```

A real example

For a real-world demo, I picked the excellent `curve25519-dalek` library, and specifically the task of multiplying the curve basepoint by a scalar and returning its Edwards representation.

The Cargo benchmarks swing widely between executions because of CPU frequency scaling, but they suggest the operation will take $22.9\mu\text{s} \pm 17\%$.

```
test curve::bench::basepoint_mult    ... bench:    17,276 ns/iter (+/- 3,057)
test curve::bench::edwards_compress  ... bench:     5,633 ns/iter (+/- 858)
```

On the Go side, we'll expose a simple API.

```
func ScalarBaseMult(dst, in *[32]byte)
```

On the Rust side, it's not different from building an interface for normal FFI.

I'll be honest, it took me forever to figure out enough Rust to make this work.

```
#![no_std]

extern crate curve25519_dalek;
use curve25519_dalek::scalar::Scalar;
use curve25519_dalek::constants;

#[no_mangle]
pub extern fn scalar_base_mult(dst: &mut [u8; 32], k: &[u8; 32]) {
    let res = &constants::ED25519_BASEPOINT_TABLE * &Scalar(*k);
    dst.clone_from(res.compress_edwards().as_bytes());
}
```

To build the `.a` we use `cargo build --release` with a `Cargo.toml` that defines the dependencies, enables frame pointers, and configures `curve25519-dalek` to use its most efficient math and no standard library.

```
[package]
name = "ed25519-dalek-rustgo"
version = "0.0.0"

[lib]
crate-type = ["staticlib"]
```

```
[dependencies.curve25519-dalek]
version = "^0.9"
default-features = false
features = ["nightly"]

[profile.release]
debug = true
```

Finally, we need to adjust the trampoline to take two arguments and return no value.

```
TEXT ·ScalarBaseMult(SB), 0, $16384-16
    MOVQ dst+0(FP), DI
    MOVQ in+8(FP), SI

    MOVQ SP, BX
    ADDQ $16384, SP
    ANDQ $~15, SP

    MOVQ ·_scalar_base_mult(SB), AX
    CALL AX

    MOVQ BX, SP
    RET
```

The result is a transparent Go call with performance that closely resembles the pure Rust benchmark, and is almost 6% faster than cgo!

name	old time/op	new time/op	delta
RustScalarBaseMult	23.7µs ± 1%	22.3µs ± 4%	-5.88% (p=0.003 n=5+7)

For comparison, similar functionality is provided by github.com/agl/ed25519/edwards25519, and that pure-Go library takes almost 3 times as long.

```
h := &edwards25519.ExtendedGroupElement{}
edwards25519.GeScalarMultBase(h, &k)
h.ToBytes(&dst)
```

name	time/op
GoScalarBaseMult	66.1µs ± 2%

Packaging up

Now we know it actually works, that's exciting! But to be usable it will have to be an importable package, not forced into `package main` by a weird build process.

This is where `//go:binary-only-package` comes in! That annotation allows us to tell the compiler to ignore the source of the package, and to only use the pre-built `.a` library file in `$GOPATH/pkg`.

2024 note: `binary-only-package` is also gone, but using a `.syso` file was probably the correct answer back then, too. Still, this was a fun detour into linking tooling.

If we can manage to build a `.a` file that works with Go's native linker (`cmd/link`, referred to also as the *internal linker*), **we can redistribute that and it will let our users import the package as if it was a native one**, including cross-compiling (provided we included a `.a` for that platform)!

The Go side is easy, and pairs with the assembly and Rust we already have. We can even include docs for `go doc`'s benefit.

```
//go:binary-only-package

// Package edwards25519 implements operations on an Edwards curve that is
// isomorphic to curve25519.
//
// Crypto operations are implemented by calling directly into the Rust
// library curve25519-dalek, without cgo.
//
// You should not actually be using this.
package edwards25519

import _ "unsafe"

//go:cgo_import_static scalar_base_mult
//go:linkname scalar_base_mult scalar_base_mult
var scalar_base_mult uintptr
var _scalar_base_mult = &scalar_base_mult

// ScalarBaseMult multiplies the scalar in by the curve basepoint, and writes
// the compressed Edwards representation of the resulting point to dst.
func ScalarBaseMult(dst, in *[32]byte)
```

The Makefile will have to change quite a bit—since we aren't building a binary anymore we don't get to keep using `go tool link`.

A `.a` archive is just a pack of `.o` object files in an ancient format with a symbol table. If we could get the symbols from the Rust `libed25519-dalek-rustgo.a` library into the

edwards25519.a archive that `go tool compile` made, we *should* be golden.

`.a` archives are managed by the `ar` UNIX tool, or by its Go internal counterpart, `cmd/pack` (as in `go tool pack`). The two formats are ever-so-subtly different, of course. We'll need to use the platform `ar` for `libed25519_dalek_rustgo.a` and the Go `cmd/pack` for `edwards25519.a`.

(For example, the platform `ar` on my macOS uses the BSD convention of calling files `#1/LEN` and then embedding the filename of length `LEN` at the beginning of the file, to exceed the 16 bytes max file length. That was confusing.)

To bundle the two libraries I tried doing the simplest (read: hackish) thing: extract `libed25519_dalek_rustgo.a` into a temporary folder, and then pack the objects back into `edwards25519.a`.

```
edwards25519/edwards25519.a: edwards25519/rustgo.go edwards25519/rustgo.o target/release
go tool compile -N -l -o $@ -p main -pack edwards25519/rustgo.go
go tool pack r $@ edwards25519/rustgo.o # from edwards25519/rustgo.s
mkdir -p target/release/libed25519_dalek_rustgo && cd target/release/li
rm -f *.o && ar xv "$(CURDIR)/target/release/libed25519_dalek_r
go tool pack r $@ target/release/libed25519_dalek_rustgo/*.o

.PHONY: install
install: edwards25519/edwards25519.a
mkdir -p "$(shell go env GOPATH)/pkg/darwin_amd64/$(IMPORT_PATH)/"
cp edwards25519/edwards25519.a "$(shell go env GOPATH)/pkg/darwin_amd64
```

Imagine my surprise when it worked!

With the `.a` in place it's just a matter of making a simple program using the package.

```
package main

import (
    "bytes"
    "encoding/hex"
    "fmt"
    "testing"

    "github.com/FiloSottile/ed25519-dalek-rustgo/edwards25519"
)

func main() {
    input, _ := hex.DecodeString("39129b3f7bbd7e17a39679b940018a737fc3bf430fcbc827029e
    expected, _ := hex.DecodeString("1cc4789ed5ea69f84ad460941ba0491ff532c1af1fa126733
```



```

var dst, k [32]byte
copy(k[:], input)

edwards25519.ScalarBaseMult(&dst, &k)
if !bytes.Equal(dst[:], expected) {
    fmt.Println("rustgo produces a wrong result!")
}

fmt.Printf("BenchmarkScalarBaseMult\t%v\n", testing.Benchmark(func(b *testing.B) {
    for i := 0; i < b.N; i++ {
        edwards25519.ScalarBaseMult(&dst, &k)
    }
})))
}

```

And running `go build` !

```

$ go build -ldflags '-linkmode external -extldflags -lresolv'
$ ./ed25519-dalek-rustgo
BenchmarkScalarBaseMult    100000      19914 ns/op

```

Well, it almost worked. We cheated. The binary would not compile unless we linked it to `libresolv`. To be fair, the Rust compiler tried to tell us. (But who listens to everything the Rust compiler tells you anyway?)

```

note: link against the following native artifacts when linking against this static lib
note: the order and any duplication can be significant on some platforms, and so may r
note: library: System
note: library: resolv
note: library: c
note: library: m

```

Now, linking against system libraries would be a problem, because it will never happen with internal linking and cross-compilation...

But hold on a minute, *libresolv*?! Why does our `no_std`, “should be like assembly”, stack only Rust library want to *resolve DNS names*?

I really meant `no_std`

The problem is that the library is not actually `no_std`. Look at all that stuff in there! We want nothing to do with allocators!

```
$ ar t target/release/libed25519_dalek_rustgo.a
__SYMDEF
ed25519_dalek_rustgo-742a1d9f1c101d86.0.o
ed25519_dalek_rustgo-742a1d9f1c101d86.crate allocator.o
curve25519_dalek-03e3ca0f6d904d88.0.o
subtle-cd04b61500f6e56a.0.o
std-72653eb2361f5909.0.o
panic_unwind-d0b88496572d35a9.0.o
unwind-da13b913698118f9.0.o
arrayref-2be0c0ff08ae2c7d.0.o
digest-f1373d68da35ca45.0.o
generic_array-95ca86a62dc11ddc.0.o
nodrop-7df18ca19bb4fc21.0.o
odds-3bc0ea0bdf8209aa.0.o
typenum-a61a9024d805e64e.0.o
rand-e0d585156faee9eb.0.o
alloc_system-c942637a1f049140.0.o
libc-e038d130d15e5dae.0.o
alloc-0e789b712308019f.0.o
std_unicode-9735142be30abc63.0.o
compiler_builtins-8a5da980a34153c7.0.o
absvdi2.o
absvsi2.o
absvti2.o
[... snip ...]
truncsfhf2.o
ucmpdi2.o
ucmpti2.o
core-9077840c2cc91cbf.0.o
```

So how do we actually make it `no_std`? This turned out to be [an entire side-quest](#), but I'll give you a recap.

- If any dependency is not `no_std`, your `no_std` flag is nullified. One of the `curve25519-dalek` dependencies had this problem, `cargo update` fixed that.
- Actually making a `no_std` *staticlib* (that is, an library for external use, as opposed to for inclusion in a Rust program) is more like making a `no_std` *executable*, which is much harder as it must be self-contained.
- The docs on how to make a `no_std` *executable* are sparse. I mostly used [an old version of the Rust book](#) and eventually found [this section in the lang_items chapter](#). [This blog post](#) was useful.

- For starters, you need to define “lang_items” functions to handle functionality that is normally in the stdlib, like `panic_fmt` .
- Then you are without the Rust equivalents of `compiler-rt` , so you have to import the crate `compiler_builtins`. (rust-lang/rust#43264)
- Then there’s a problem with `rust_begin_unwind` being unexported, which don’t ask me why but is solved by marking `panic_fmt` as `no_mangle` , which the linter is not happy about. (rust-lang/rust#38281)
- Then you are without `memcpy` , but thankfully there’s a native Rust reimplementation in the `rlibc` crate. Super useful [learning](#) that `nm -u` will tell you what symbols are missing from an object.

This all boils down to a bunch of arcane lines at the top of our `lib.rs` .

```
#![no_std]
#![feature(lang_items, compiler_builtins_lib, core_intrinsics)]
use core::intrinsics;
#[allow(private_no_mangle_fns)] #[no_mangle] // rust-lang/rust#38281
#[lang = "panic_fmt"] fn panic_fmt() -> ! { unsafe { intrinsics::abort() } }
#[lang = "eh_personality"] extern fn eh_personality() {}
extern crate compiler_builtins; // rust-lang/rust#43264
extern crate rlibc;
```

And with that, `go build` works (!!!) on macOS.

Linux

On Linux nothing works.

External linking complains about `fmax` and other symbols missing, and it seems to be right.

```
$ ld -r -o linux.o target/release/libed25519_dalek_rustgo/*.o
$ nm -u linux.o
                 U _GLOBAL_OFFSET_TABLE_
                 U abort
                 U fmax
                 U fmaxf
                 U fmaxl
                 U logb
                 U logbf
                 U logbl
                 U scalbn
                 U scalbnf
                 U scalbnl
```

A friend thankfully suggested making sure that I was using `--gc-sections` to strip dead code, which might reference things I don't actually need. And sure enough, this worked. (That's three layers of flag-passing right there.)

```
$ go build -ldflags '-extld clang -linkmode external -extldflags -Wl,--gc-sections'
```

But umh, in the Makefile we aren't using a linker at all, so where do we put `--gc-sections`? The answer is to stop hacking `.a`s together and actually reading the [linker man page](#).

We can build a `.o` containing a given symbol and all the symbols it references with `ld -r --gc-sections -u $SYMBOL .` `-r` makes the object reusable for a later link, and `-u` marks a symbol as needed, or everything would end up garbage collected. `$SYMBOL` is `scalar_base_mult` in our case.

Why wasn't this a problem on macOS? It would have been if we linked manually, but the macOS compiler apparently does dead symbol stripping by default.

```
$ ld -e _scalar_base_mult target/release/libed25519_dalek_rustgo/*.o
Undefined symbols for architecture x86_64:
  "__assert_rtn", referenced from:
    _compiler_rt_abort_impl in int_util.o
  "_copysign", referenced from:
    __divdc3 in divdc3.o
    __muldc3 in muldc3.o
  "_copysignf", referenced from:
    __divsc3 in divsc3.o
    __mulsc3 in mulsc3.o
  "_copysignl", referenced from:
    __divxc3 in divxc3.o
    __mulxc3 in mulxc3.o
  "_fmax", referenced from:
    __divdc3 in divdc3.o
  "_fmaxf", referenced from:
    __divsc3 in divsc3.o
  "_fmaxl", referenced from:
    __divxc3 in divxc3.o
  "_logb", referenced from:
    __divdc3 in divdc3.o
  "_logbf", referenced from:
    __divsc3 in divsc3.o
  "_logbl", referenced from:
    __divxc3 in divxc3.o
  "_scalbn", referenced from:
    __divdc3 in divdc3.o
  "_scalbnf", referenced from:
    __divsc3 in divsc3.o
  "_scalbnl", referenced from:
```

```

__divxc3 in divxc3.o
ld: symbol(s) not found for inferred architecture x86_64
$ ld -e _scalar_base_mult -dead_strip target/release/libed25519_dalek_rustgo/*.o

```

This is also the part where we learn painfully that the macOS platform prepends a `_` to all symbol names, because reasons.

So here's the Makefile portion that will work with external linking out of the box.

```

edwards25519/edwards25519.a: edwards25519/rustgo.go edwards25519/rustgo.o edwards25519
go tool compile -N -l -o $@ -p main -pack edwards25519/rustgo.go
go tool pack r $@ edwards25519/rustgo.o edwards25519/libed25519_dalek_rustgo.c

edwards25519/libed25519_dalek_rustgo.o: target/${TARGET}/release/libed25519_dalek_rust
ifeq ($(shell go env GOOS),darwin)
    $(LD) -r -o $@ -arch x86_64 -u "$$(SYMBOL)" $^
else
    $(LD) -r -o $@ --gc-sections -u "$$(SYMBOL)" $^
endif

```

The last missing piece is internal linking on Linux. In short, it was not linking the Rust code, even if the compilation seemed to succeed. The relocations were not happening and the `CALL` instructions in our Rust function left pointing at meaningless addresses.

At that point I felt like it had to be a silent linker bug, the final boss in implementing rustgo, and reached out to people much smarter than me. One of them was guiding me in debugging cmd/link (which was fascinating!) when Ian Lance Taylor, the author of `cgo`, helpfully pointed out that `//cgo:cgo_import_static` is not enough for internal linking, and that I also wanted `//cgo:cgo_import_dynamic`.

```

//go:cgo_import_static scalar_base_mult
//go:cgo_import_dynamic scalar_base_mult

```

I still have no idea *why* leaving it out would result in that issue, but adding it finally made our rustgo package compile both with external and internal linking, on Linux and macOS, out of the box.

Redistributable

Now that we can build a `.a`, we can take the suggestion in the `//go:binary-only-package` spec, and build a tarball with `.a`s for `linux_amd64` / `darwin_amd64` and the package source, to

untar into a GOPATH to install.

```
$ tar tf ed25519-dalek-rustgo_go1.8.3.tar.gz
src/github.com/FiloSottile/ed25519-dalek-rustgo/
src/github.com/FiloSottile/ed25519-dalek-rustgo/.gitignore
src/github.com/FiloSottile/ed25519-dalek-rustgo/Cargo.lock
src/github.com/FiloSottile/ed25519-dalek-rustgo/Cargo.toml
src/github.com/FiloSottile/ed25519-dalek-rustgo/edwards25519/
src/github.com/FiloSottile/ed25519-dalek-rustgo/main.go
src/github.com/FiloSottile/ed25519-dalek-rustgo/Makefile
src/github.com/FiloSottile/ed25519-dalek-rustgo/release.sh
src/github.com/FiloSottile/ed25519-dalek-rustgo/src/
src/github.com/FiloSottile/ed25519-dalek-rustgo/target.go
src/github.com/FiloSottile/ed25519-dalek-rustgo/src/lib.rs
src/github.com/FiloSottile/ed25519-dalek-rustgo/edwards25519/rustgo.go
src/github.com/FiloSottile/ed25519-dalek-rustgo/edwards25519/rustgo.s
pkg/linux_amd64/github.com/FiloSottile/ed25519-dalek-rustgo/edwards25519.a
pkg/darwin_amd64/github.com/FiloSottile/ed25519-dalek-rustgo/edwards25519.a
```

Once installed like that, the package will be usable just like a native one, cross-compilation included (as long as we packaged a `.a` for the target)!

The only thing we have to worry about is that if we build Rust with `-Ctarget-cpu=native` it might not run on older CPUs. Thankfully benchmarks ([and the curve25519-dalek authors](#)) tell us that the only real difference is between post and pre-Haswell processors, so we only have to make a universal build and a Haswell one.

```
$ benchstat bench-none.txt bench-haswell.txt
name                old time/op  new time/op  delta
ScalarBaseMult/rustgo 22.0µs ± 3%  20.2µs ± 2%  -8.41% (p=0.001 n=7+6)
$ benchstat bench-haswell.txt bench-native.txt
name                old time/op  new time/op  delta
ScalarBaseMult/rustgo 20.2µs ± 2%  20.1µs ± 2%   ~      (p=0.945 n=6+7)
```

As the cherry on top, I made the Makefile obey GOOS/GOARCH, converting them as needed into Rust target triples, so if you have Rust set up for cross-compilation you can even cross-compile the `.a` itself.

Here's the result: github.com/FiloSottile/ed25519-dalek-rustgo/edwards25519. It's even on [godoc](#).

Turning it into a real thing

Well, this was fun.

But to be clear, rustgo is not a real thing that you should use ~~in production~~. For example, I suspect I should be saving `g` before the jump, the stack size is completely arbitrary, and shrinking the trampoline frame like that will probably confuse the hell out of debuggers. Also, a panic in Rust might get weird.

To make it a real thing I'd start by calling `morestack` manually from a `NOSPLIT` assembly function to ensure we have enough goroutine stack space (instead of rolling back `rsp`) with a size obtained maybe from static analysis of the Rust function (instead of, well, made up).

It could all be analyzed, generated and built by some "rustgo" tool, instead of hardcoded in Makefiles and assembly files. cgo itself is little more than a code-generation tool after all. It might make sense as a `go:generate` thing, but I know someone who wants to make it a cargo command. (Finally some Rust-vs-Go fighting!) Also, a Rust-side collection of FFI types like, say, `GoSlice` would be nice.

```
#[repr(C)]
struct GoSlice {
    array: *mut u8,
    len: i32,
    cap: i32,
}
```

Or maybe a Go or Rust adult will come and tell us to stop before we get hurt.

In the meantime, you might want to [follow me on Twitter](#).

2024 note: Twitter is *also* not usable anymore. Instead, you can follow me on Bluesky at [@filippo.abyssdomain.expert](#) or on Mastodon at [@filippo@abyssdomain.expert](#).




EDIT: It was pointed out to me that if we simply named the Rust object file

`libed25519_dalek_rustgo.syso`, we could skip all the `go tool` invocations and simply use `go build` which automatically links `.syso` files found in the package. But what's the fun in that.

Thanks (in no particular order) to David, Ian, Henry, Isis, Manish, Zaki, Anna, George, Kaylyn, Bill, David, Jess, Tony and Daniel for making this possible. Don't blame them for the mistakes and horrors, those are mine.

P.S. Before anyone tries to compare this to cgo (which has many more safety features) or pure Go, it's not meant to replace either. It's meant to replace manually written assembly with

something much safer and more readable, with comparable performance. Or better yet, it was meant to be a fun experiment.

[Subscribe](#)  | [Feed](#)  | [Bluesky](#)  | [Mastodon](#) 