F-Secure®

TamaGo

Bare metal Go framework for ARM SoCs

Andrea Barisani

Head of Hardware Security - F-Secure

@AndreaBarisani - andrea.bio

andrea.barisani@f-secure.com - foundry.f-secure.com

$ whoami

F-Secure

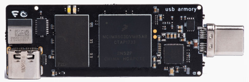
Andrea Barisani

Information Security Researcher

Founder of INVERSE PATH

Head of Hardware Security at

Maker of the USB armory

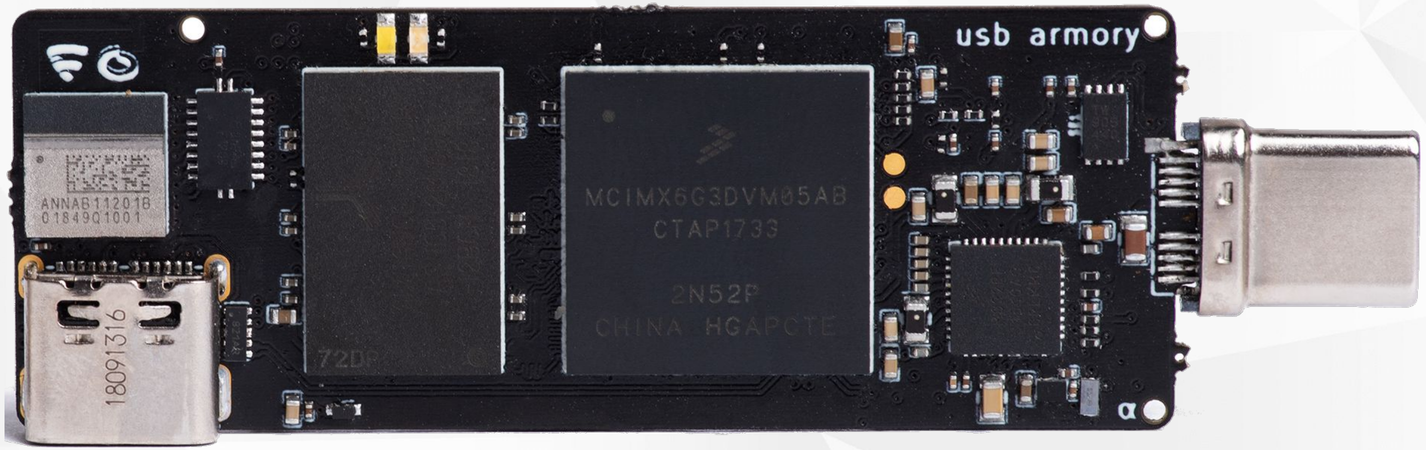


Speaker at numerous conferences.

Security auditing and engineering with focus on safety critical systems in the automotive, avionics, industrial domain.

twitter: @AndreaBarisani website: https://andrea.bio

Motivation: USB armory firmware



https://github.com/inversepath/usbarmory/wiki https://www.f-secure.com/en/consulting/foundry/usb-armory

Motivation

In an ideal world **you should be free to choose the language you prefer**.

In an ideal world **all compilers would generate machine code with the same efficiency**.

However in real world lower specs heavily dictate language choices:

Microcontroller (MCU) firmware == unsafe[1] low level languages (C)

Examples: cryptographic tokens, cryptocurrency wallets, hardware diodes, lower specs IoT and "smart" appliances.

[1] **Pro tip**: certification does not matter.

Motivation

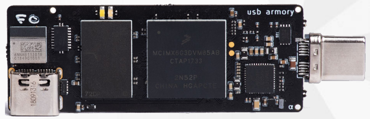
In an ideal world using **higher level languages should not entail complex dependencies**.

In an ideal world **higher level languages should reduce complexity**.

Complexity should be reduced for the entire environment, not just being shifted away.

However in real world higher specs heavily dictate OS requirements:

System-on-Chip (SoC) firmware == complex OS + safe (or unsafe[1]) languages



Examples: TEE applets, infotainment units, avionics gateways, home routers, higher specs IoT and "smart" appliances.

[1] Privileged C-based apps running under Linux to "parse stuff" are very common, like your car infotainment/parking ECU.

Killing C

When security matters software and hardware optimizations matter less.

This means that less constrained hardware (e.g. SoCs in favor of MCUs) and higher level code are perfectly acceptable.

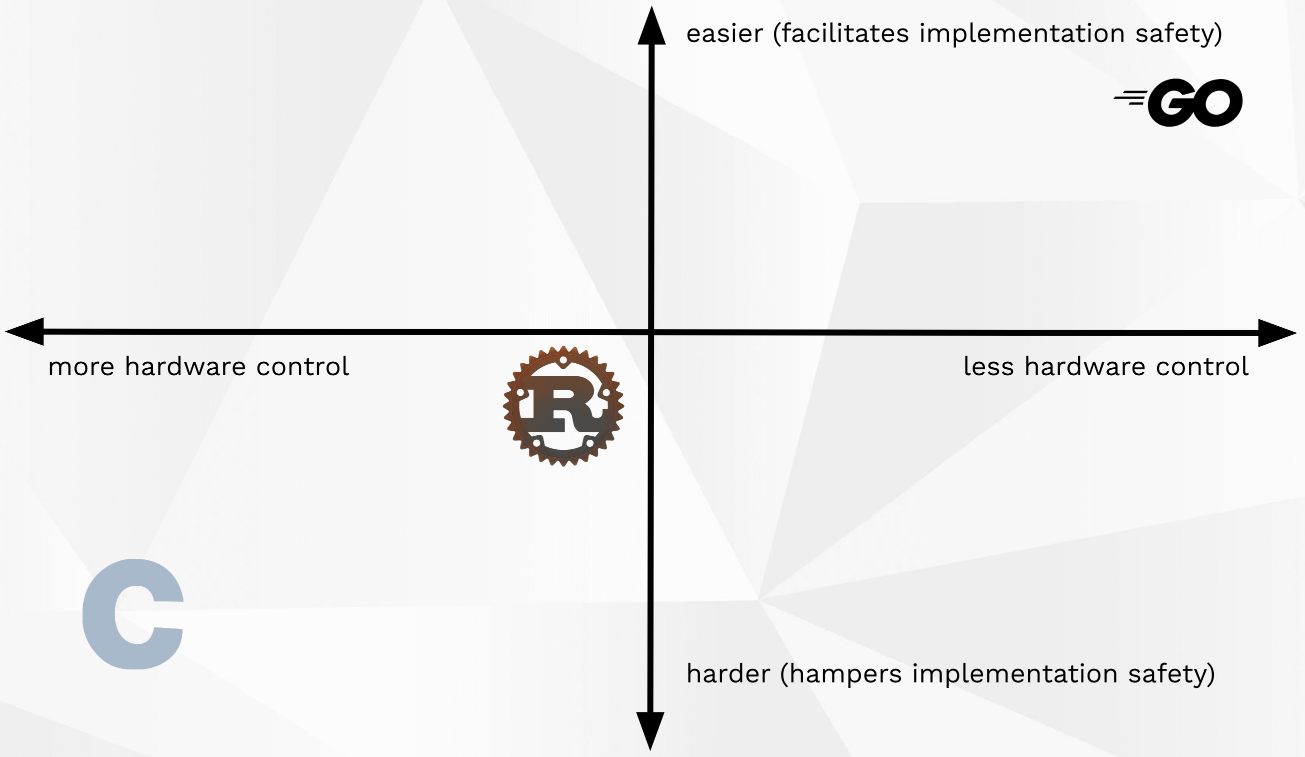
However high level programming typically entails several layers (e.g. OS, libraries) to serve runtime execution.

TamaGo spawns from the desire of **reducing the attack surface** of embedded systems firmware by **removing any runtime dependency on C code and inherently complex Operating Systems**.

In other words we want to **avoid shifting complexity around** and run a **higher level language**, such as Go in our effort, **directly on the bare metal**.

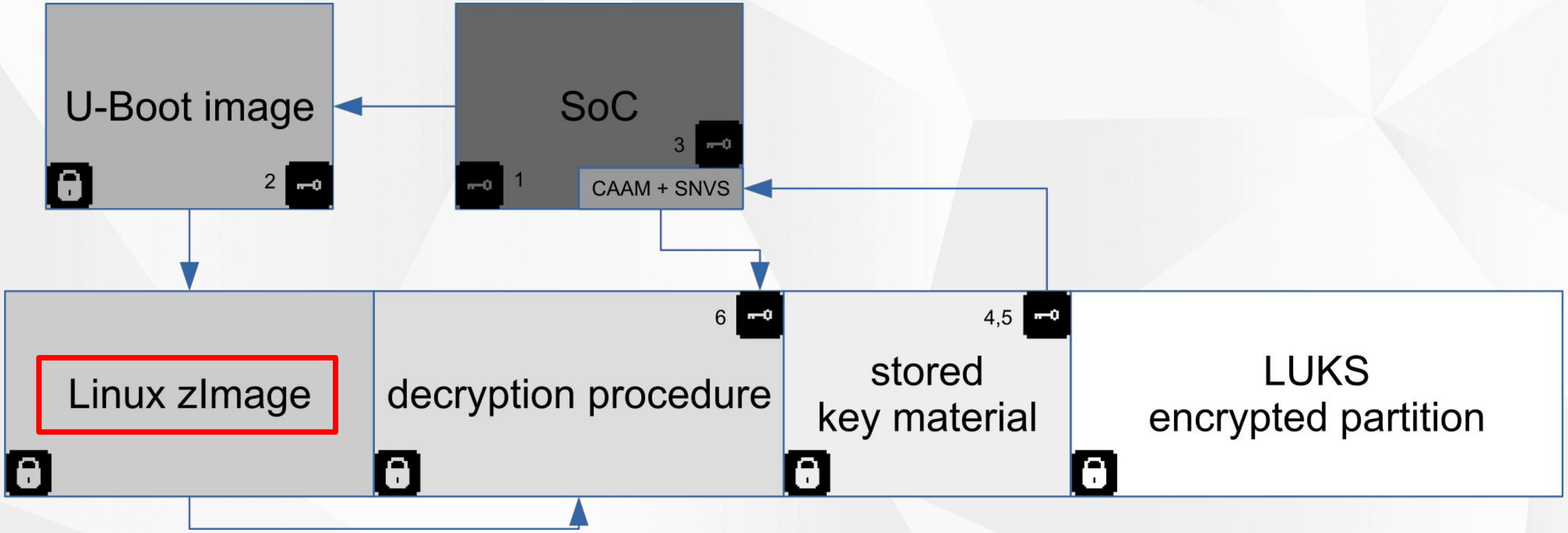
Audience mind reading trick: you are thinking "why not Rust?" ... well why not *both* ?

Speed vs Safety



Disclaimer: chart presented for discussion and not to claim that language X is better than language Y, also scale is subjective.

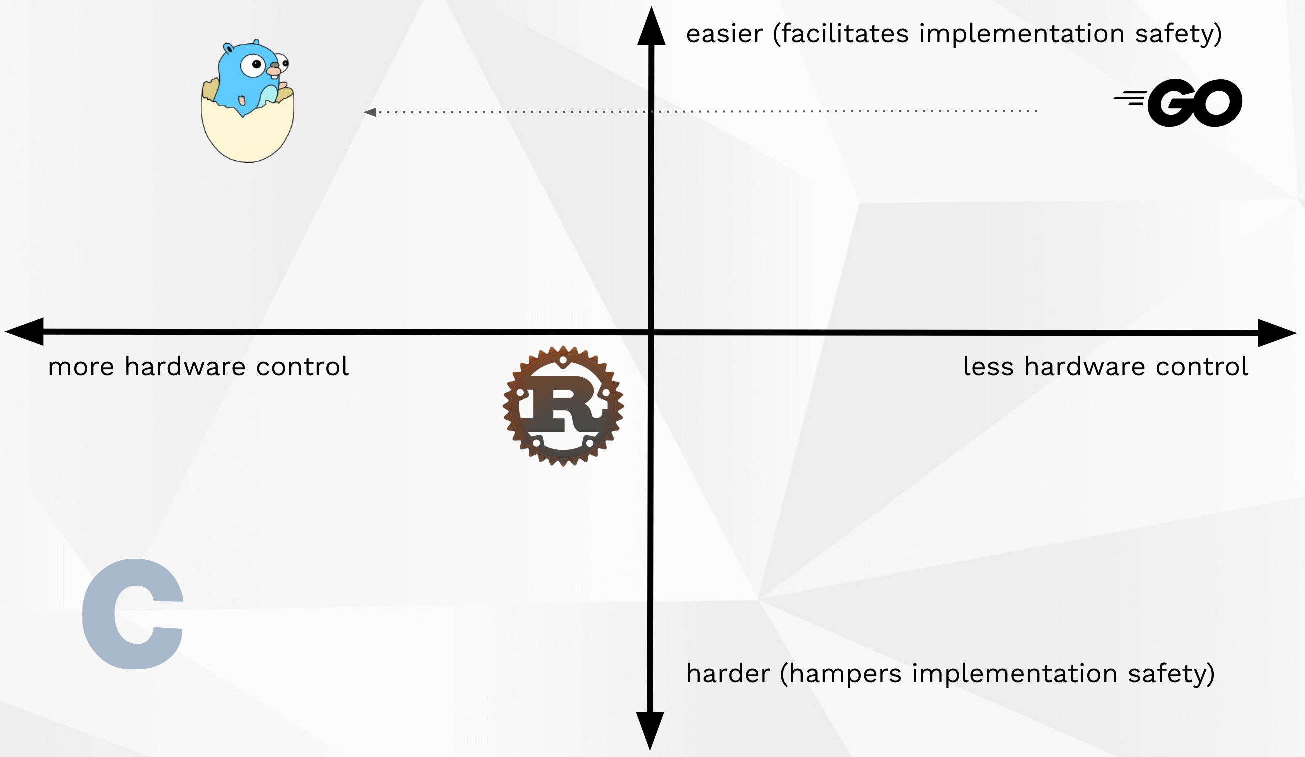
Reducing the attack surface



Typical secure booted firmware with authentication and confidentiality, taken from USB armory implementation example (NXP i.MX6ULL).

https://github.com/inversepath/usbarmory/wiki/Secure-boot-(Mk-II)

Speed vs Safety


easier (facilitates implementation safety)
GO
more hardware control
less hardware control
C
harder (hampers implementation safety)

Disclaimer: chart presented for discussion and not to claim that language X is better than language Y, also scale is subjective.

Unikernels / library OS

Unikernels[1] are a single address space image to executed a “library operating system”, typically running under bare metal.

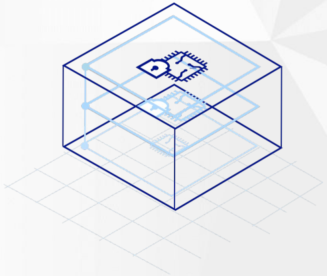
The focus is reducing the attack surface, carrying only strictly necessary code.

“True” unikernels are mostly unicorns, as a good chunk of available ones do not fit in this category and represent “fat” unikernels running under hypervisors and/or other (mini) OSes And just shift around complexity (e.g. the app is PID 1).

Apart for some exceptions there is always still a lot of C/dependencies involved in the underlying OS, drivers or hypervisor.

Running or importing *BSD kernels
Rump kernels (NetBSD based)
OSv (re-uses code from FreeBSD)

Running under hypervisor and 3rd party kernel
MirageOS (Solo5)
ClickOS (MiniOS)



Running under hypervisor
Nanos (Xen/KVM/Qemu) HalVM (Haskell, Xen)
LING (Erlang, Xen) RustyHermit (KVM)

Bare metal
GRISP (Erlang)
IncludeOS

[1] https://en.wikipedia.org/wiki/Unikernel

An excellent summary: https://github.com/cetic/unikernels

Unikernel security

From a security standpoint leveraging on Unikernels (whatever the kind) to run multiple applications or an individual C applications is not ideal[1].

Having an industry standard OS is necessary to support required security measures which otherwise are not present or rather primitive on most Unikernels.

Again, we want to **kill C** from the entire environment while keeping code efficiency, developing drivers having "only" to worry about interpreting reference manuals.

Unlike most unikernel projects we focus on **small embedded systems**, not the cloud.

We chose **Go** for its shallow learning curve, productivity, strong cryptographic library and standard library.

Languages like Rust have already proven they role in bare metal world, Go on the other hand needs to...as it really can.

[1] https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2019/april/assessing-unikernel-security/

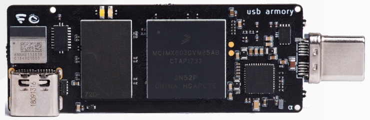
TamaGo in a nutshell

TamaGo is made of two main components.

- A **minimally**[1] patched Go compiler to enable `GOOS=tamago` support, which provides freestanding execution on `GOARCH=arm` bare metal.

- A set of packages[2] to provide board support (e.g. hardware initialization and drivers).

TamaGo currently provides drivers for the NXP i.MX6ULL System-on-Chip to enable support of the USB armory Mk II.



We target development of security applications and we want to leverage our existing open source tooling for i.MX6 Secure Boot authentication.

We plan to support additional boards, such as the Raspberry Pi Zero, in the near future.

[1] https://github.com/inversepath/tamago-go

[2] https://github.com/inversepath/tamago

Similar Go efforts

Biscuit (unmaintained) - https://github.com/mit-pdos/biscuit

> Go kernel for non-Go software underneath, larger scope, intermediate C bootloader, hijacks `GOOS=linux`, only for `GOARCH=amd64`, redoes memory allocation and threading.

G.E.R.T (unmaintained) - https://github.com/ycoroneos/G.E.R.T

> ARM adaptation of Biscuit but without non-Go software support, intermediate C bootloader, hijacks `GOOS=linux` for `GOARCH=arm`, redoes memory allocation and threading.

AtmanOS (unmaintained) - https://github.com/atmanos

> Similar to TamaGo but targets execution under the Xen hypervisor, adds `GOOS=atman` but with limited runtime support.

Tiny Go (active and rocking!) - https://github.com/tinygo-org

> LLVM based compiler (not original one) aimed at MCUs and minimal footprint, does not support the entire runtime and Go language support differs from standard Go.

Embedded Go (brand new, November 2019) - https://github.com/embeddedgo

> Similar to TamaGo but targets ARMv7-M/Thumb2 adding new support for it, as not native to Go. Adds `GOOS=noos GOARCH=thumb`, features interrupt/timer support.

All these projects greatly supported us in proving feasibility and identify TamaGo unique approach, diversity is good.

Enabling trust

TamaGo not only wants to prove that it is possible to have a bare metal Go runtime, but wants to prove that it can be achieved with **clean and minimal modifications against the original Go compiler**[2].

Much of the effort has been placed to understand whether Go bare metal support can be achieved without complex re-implementation of memory allocation, threading, ASM/C OS primitives that would “pollute” the Go runtime to unacceptable levels.

Less is more. Complexity is the enemy of verifiability.

The acceptance of this (and similar) efforts hinges on maintainability, ease of review, clarity, simplicity and **trust**.

★ Designed to achieve upstream inclusion and with commitment to always sync to latest Go release.
★ ~3000 LOC of compiler changes with clean separation from other `GOOS` support.
★ Strong emphasis on code reuse from existing architectures of standard Go runtime, see Internals[1].
★ Requires only one import (“library OS”) on the target Go application.
★ Supports unencumbered Go applications with nearly full runtime availability.
★ In addition to the compiler, aims to provide a complete set of driver peripherals for SoCs.

[1] https://github.com/inversepath/tamago/wiki/Internals

[2] Which by the way is self-hosted and has reproducible builds.

Go compiler modifications[1]

Glue code (~340 lines, ~100 files): patches to adds `GOOS=tamago` to the list of supported architectures and required stubs for unsupported operations. All changes are benign (no logic/function):

```
// +build aix darwin dragonfly freebsd js,wasm linux nacl netbsd openbsd solaris tamago
```

Re-used[2] code (~2700 lines, 6 files): patches that clone original Go runtime functionality from an existing architecture to `GOOS=tamago`, either unmodified or with minimal changes:

- `plan9` memory allocation is re-used with 2 LOC changed (`brk` vs simple pointer)
- `js,wasm` locking is re-used identically (with JS VM hooks removed)
- `nacl` in-memory filesystem is re-used (TODO: eMMC/VFAT support)

New code (~600 lines, 12 files): `tamago` architecture specific functionality, mainly provides ARMv7 initialization functions and set Go heap arena size to available RAM:

```
rt0_tamago_arm.s     (LOC: 13)      sys_tamago_arm.s (LOC: 133)
rand_tamago.os       (LOC: 29)      os_tamago_arm.os (LOC: 377)
```

https://github.com/golang/go/compare/go1.13.6...inversepath:tamago1.13.6

[1] As of 2019-12-27 against `go1.13.5`

[2] a.k.a. Go Frankenstein

TamaGo memory layout

```
+-----------------------------------+ 0000 0000
|                                   |
+-----------------------------------+ runtime.ramStart
|                                   |
|  INTERRUPT VECTOR TABLE (16 kB)   |
|                                   |
+-----------------------------------+ runtime.ramStart + 0x4000 (16 kB)
|                                   |
|        L1 PAGE TABLE (16 kB)      |
|                                   |
+-----------------------------------+ runtime.ramStart + 0x8000 (32 kB)
|                                   |
|       EXCEPTION STACK (16 kB)      |
|                                   |
+-----------------------------------+ runtime.ramStart + 0xC000 (48 kB)
|                                   |
|          UNUSED (16 kB)           |
|                                   |
+-----------------------------------+ runtime.ramStart + 0x10000 (64 kB)
| .text                             |
|                                   |
| .noptrdata                        |
|                                   |
| .data                             | Go application
|                                   |
| .bss                              |
|                                   |
| .noptrbss                         |
+-----------------------------------+
|                                   |
|             HEAP                  |
|                                   |
+-----------------------------------+ runtime.g0.stack.lo (runtime.go.stack.hi - 0x10000)
|                                   |
|          STACK (64 kB)            |
|                                   |
+-----------------------------------+ runtime.go.stack.hi (runtime.ramStart + runtime.ramSize - runtime.stackOffset)
|                                   |
|             UNUSED                |
|                                   |
+-----------------------------------+ runtime.ramStart + runtime.ramSize
|                                   |
|                                   |
+-----------------------------------+ FFFF FFFF
```

https://github.com/inversepath/tamago/wiki/Internals

Go runtime support

```
// the following variables must be provided externally
var ramStart uint32
var ramStackOffset uint32
var ramSize uint32

// the following functions must be provided externally
func hwinit()
func printk(byte)
func getRandomData([]byte)
func initRNG()
func nanotime() int64

func mmuinit() {
	// Initialize page tables and map regions in privileged system area.
	//
	// MMU initialization is required to take advantage of data cache.
	// http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13835.html
	//
	// Define an empty L1 page table, the MMU is enabled only for caching to work.
	// The L1 page table is located 16KB after ramStart.

	l1pageTableStart := ramStart + l1pageTableOffset
	memclrNoHeapPointers(unsafe.Pointer(uintptr(l1pageTableStart)), uintptr(l1pageTableSize))
	dmb()

	set_ttbr0(unsafe.Pointer(uintptr(l1pageTableStart)))
}
```

Example of separation between Go runtime, SoC and board packages with pre-defined hooks using `go:linkname`.

```
package imx6ul

//go:linkname ramStart runtime.ramStart
var ramStart uint32 = 0x80000000

// ramSize defined in board package
//go:linkname ramStackOffset runtime.ramStackOffset
var ramStackOffset uint32 = 0x100
```

```
package usbarmory

//go:linkname ramSize runtime.ramSize
var ramSize uint32 = 0x20000000 // 512 MB

//go:linkname printk runtime.printk
func printk(c byte) {
	imx6.UART2.Write(c)
}
```

https://github.com/inversepath/tamago-go

https://github.com/inversepath/tamago

Go runtime support

```
                    os_tamago_arm.go (Go runtime)
//go:linkname syscall_now syscall.now
func syscall_now() (sec int64, nsec int32) {
        sec, nsec, _ = time_now()
        return
}
                    timer.go (imx6 package)
//go:linkname nanotime runtime.nanotime
func nanotime() int64 {
        return int64(read_gtc() * timerMultiplier)
}
                    timer.s (imx6 package)
// func read_gtc() int64
TEXT ·read_gtc(SB),$0
        // Cortex™-A9 MPCore® Technical Reference Manual
        // 4.4.1 Global Timer Counter Registers, 0x00 and 0x04
        // p214, Table 2-1, ARM MP Global timer, IMX6DQRM
        MOVW $0x00a00204, R1
        MOVW $0x00a00200, R2
read:
        MOVW     (R1), R3
        MOVW     (R2), R4
        MOVW     (R1), R5
        CMP      R5, R3
        BNE      read

        MOVW     R3, ret_hi+4(FP)
        MOVW     R4, ret_lo+0(FP)

        RET
```

A small set of low-level functions are integrated directly with Go Assembly.

This follows existing patterns in the Go runtime.

In the example ARM Generic Timers (ARM-Cortex A7) are used to support ticks and time related functions.

Overall initialization code accounts for less than 500 lines of code.

Go runtime support

```
func setARMFreqIMX6ULL(hz uint32) (err error) {
        cacrr := (*uint32)(unsafe.Pointer(uintptr(CCM_CACRR)))
        pll := (*uint32)(unsafe.Pointer(uintptr(CCM_ANALOG_PLL_ARM)))
        curHz := ARMFreq()

        log.Printf("imx6_clk: changing ARM core frequency to %d MHz\n", hz/1000000)

...

        // set bypass source to main oscillator
        reg.SetN(pll, CCM_ANALOG_PLL_ARM_BYPASS_CLK_SRC, 0b11, 0)

        // bypass
        reg.Set(pll, CCM_ANALOG_PLL_ARM_BYPASS)

        // set PLL divisor
        reg.SetN(pll, CCM_ANALOG_PLL_ARM_DIV_SELECT, 0b1111111, div_select)

        // wait for lock
        log.Printf("imx6_clk: waiting for PLL lock\n")
        reg.Wait(pll, CCM_ANALOG_PLL_ARM_LOCK, 0b1, 1)

        // remove bypass
        reg.Clear(pll, CCM_ANALOG_PLL_ARM_BYPASS)

        // set core divisor
        reg.SetN(cacrr, CCM_CACRR_ARM_PODF, 0b111, arm_podf)

        setOperatingPointIMX6ULL(uV)

...
```

Example: changing the i.MX6ULL SoC ARM core clock frequency.

Go's `unsafe` can be easily identified throughout all drivers to spot areas that require care (e.g. pointer arithmetic).

https://github.com/inversepath/tamago-go https://github.com/inversepath/tamago

Go runtime support

```go
//go:linkname syscall
func syscall(number, a1, a2, a3 uintptr) (r1, r2, err uintptr) {
	switch number {
	case 1: // SYS_WRITE
		r1 := write(a1, unsafe.Pointer(a2), int32(a3))
		return uintptr(r1), 0, 0
	default:
		throw("unexpected syscall")
	}

	return
}

//go:nosplit
func write(fd uintptr, buf unsafe.Pointer, count int32) int32 {
	if fd != 1 && fd != 2 {
		throw("unexpected fd, only stdout/stderr are supported")
	}

	c := uintptr(count)

	for i := uintptr(0); i < c; i++ {
		p := (*byte)(unsafe.Pointer(uintptr(buf) + i))
		printk(*p)
	}

	return int32(c)
}
```

Only the `write` syscall is required for the overwhelming majority of basic runtime support.

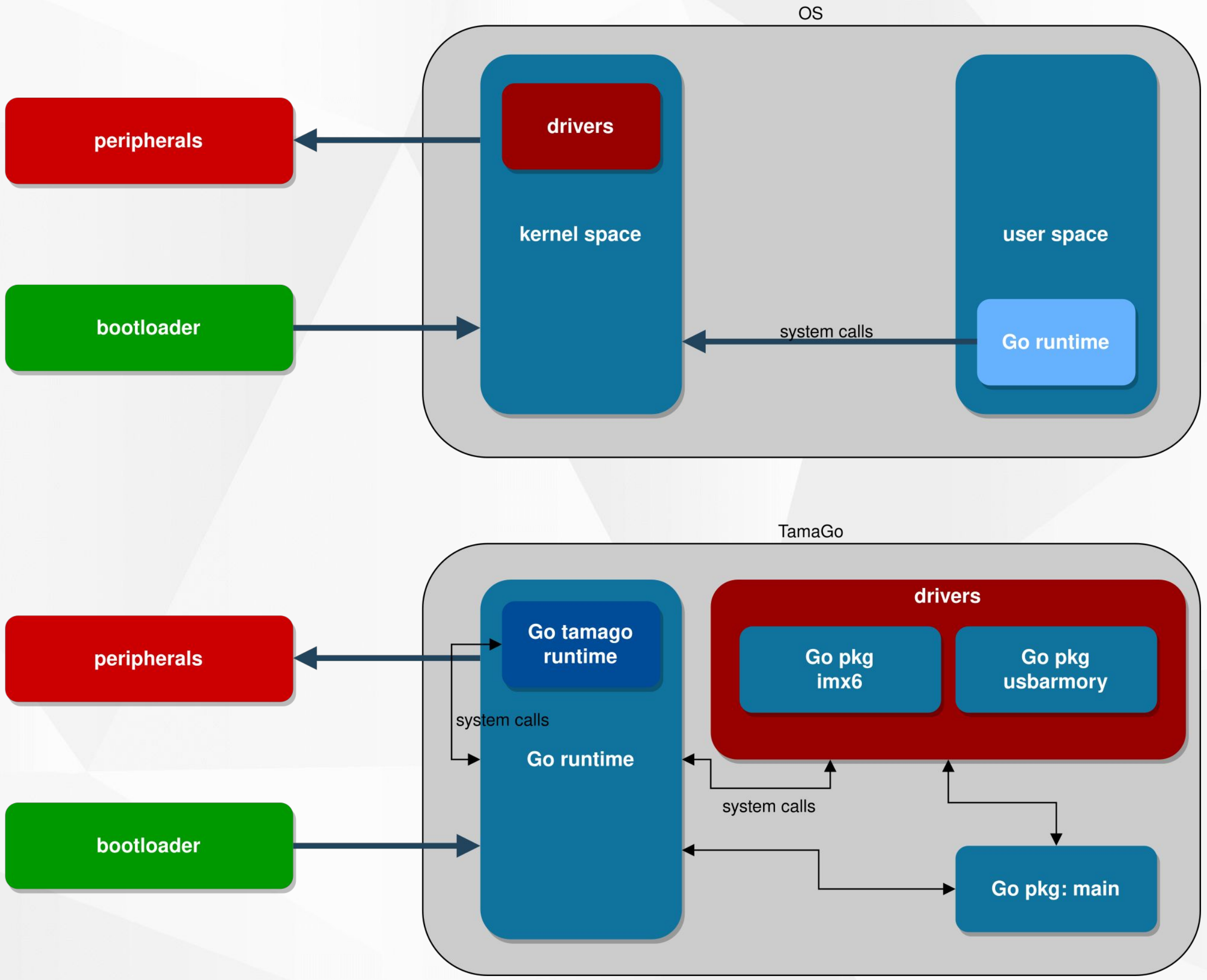
As shown before, `printk` is provided by the application to define method for writing on standard output (e.g. UART).

```
imx6_clk: changing ARM core frequency to 900 MHz
imx6_clk: changing ARM core operating point to 575000 uV
imx6_clk: 450000 uV -> 575000 uV
imx6_clk: waiting for PLL lock
imx6_clk: 396 MHz -> 900 MHz
imx6_soc: i.MX6ULL (0x65, 0.1) @ freq:900 MHz - native:true
```

https://github.com/inversepath/tamago-go https://github.com/inversepath/tamago

TamaGo

F-Secure



https://github.com/inversepath/tamago/wiki/Internals

Developing, building and running

The full Go runtime is supported[1] without any specific changes required on the application side (Rust on bare metal[2], for comparison, requires `#![no_std]` pragma).

```
package main

import (
	_ "github.com/inversepath/tamago/usbarmory/mark-two"
)

func main() {
	// your code
}
```

1. The application requires a single import for the board package to enable necessary initializations.

2. Go code can be written with very few limitations and the `imx6` package can be used for any SoC specific driver operation.

```
GO_EXTLINK_ENABLED=0 CGO_ENABLED=0 GOOS=tamago GOARM=7 GOARCH=arm \
  ${TAMAGO} build -ldflags "-T 0x80010000 -E _rt0_arm_tamago -R 0x1000"
```

3. `go build` can be used as usual (reproducible builds!) with few linker flags to define entry point.

```
=> ext2load mmc $dev:1 0x90000000 go.elf

=> bootelf -p 0x90000000
```

4. The resulting ELF binary can be passed to the bootloader, such as U-Boot, for load and execution.

Examples shown for USB armory Mk II / i.MX6ULL.

[1] https://github.com/inversepath/tamago/wiki/Import-report [2] https://rust-embedded.github.io/book/intro/no-std.html

i.MX6ULL driver: Data Co-Processor (DCP)

The DCP provides hardware accelerated crypto functions and use of the SoC unique OTPMK key for device unique encryption/decryption operations. The driver takes ~240 LOC.

```go
workPacket := &WorkPacket{}
reg.Set(&workPacket.Control0, DCP_CTRL0_INTERRUPT_ENABL)
reg.Set(&workPacket.Control0, DCP_CTRL0_DECR_SEMAPHORE)
reg.Set(&workPacket.Control0, DCP_CTRL0_ENABLE_CIPHER)
reg.Set(&workPacket.Control0, DCP_CTRL0_CIPHER_ENCRYPT)
reg.Set(&workPacket.Control0, DCP_CTRL0_CIPHER_INIT)

// Use device-specific hardware key, payload does not contain the key.
reg.Set(&workPacket.Control0, DCP_CTRL0_OTP_KEY)

workPacket.Control1 |= (AES128 << DCP_CTRL1_CIPHER_SELECT)
workPacket.Control1 |= (CBC << DCP_CTRL1_CIPHER_MODE)
workPacket.Control1 |= (UNIQUE_KEY << DCP_CTRL1_KEY_SELECT)

workPacket.BufferSize = uint32(len(diversifier))
workPacket.SourceBufferAddress = &diversifier[0]
workPacket.DestinationBufferAddress = &key[0]

// p1073, Table 13-12. DCP Payload Field, MCIMX28RM
workPacket.PayloadPointer = &iv[0]

hw.Lock()
defer hw.Unlock()

*(hw.pkt) = workPacket
```

```go
diversifier := []byte{0xde, 0xad, 0xbe, 0xef}
iv := make([]byte, aes.BlockSize)

key, err := imx6.DCP.DeriveKey(diversifier, iv)
```

```
-- i.mx6 dcp ---------------------------------------------------------
imx6_dcp: derived test key 75f9022d5a867ad430440feec6611f0a
```

USB armory Mk II example DCP + SNVS run (w/ Secure Boot)

```
-- i.mx6 dcp ---------------------------------------------------------
imx6_dcp: error, SNVS unavailable, not in trusted or secure state
```

USB armory Mk II example DCP + SNVS run (w/o Secure Boot)

Note that Go defined structs (such as `WorkPacket`) can be easily made C-compatible[1].

https://github.com/inversepath/usbarmory/wiki/Hardware-security-features-(Mk-II)

[1] Use cgo `-godefs`.

i.MX6ULL driver: Random Number Generator

F-Secure

The RNGB provides a hardware True Random Number Generator, useful to gather the initial seed on embedded systems without a battery backed RTC (and not much else[2]). The driver takes ~150 LOC and is hooked as provider for crypto/rand.

```go
var getRandomDataFn func([]byte)

//go:linkname getRandomData runtime.getRandomData
func getRandomData(b []byte) {
	getRandomDataFn(b)
}

func (hw *rngb) getRandomData(b []byte) {
	read := 0
	need := len(b)


	for read < need {
		if reg.Get(hw.status, HW_RNG_SR_ERR, 0x1) != 0 {
			panic("imx6_rng: panic\n")
		}


		if reg.Get(hw.status, HW_RNG_SR_FIFO_LVL, 0xf) > 0 {
			val := *hw.fifo
			read = fill(b, read, val)
		}
	}
}
```

```go
for i := 0; i < 10; i++ {
		rng := make([]byte, size)
		rand.Read(rng)
		fmt.Printf("%x\n", rng)
}
```

```
-- rng ----------------------------------------------------------------
imx6_rng: self-test
imx6_rng: seeding
f90b00053a50b9edd42df027c982769d1a7d25445e31ce98486bd4a9676bef42
56baf6ecc32bf02fb9d09c2d8c607baa487e2283b6856486b42cdf954277d4d5
49fc0c03f8cbc45f7aeb58ba71c0d561a91dbeae697d7bc511482697bf96b2f8
345db47ab3395272a9db9531f03160b3e1654b7e8b7267c1a3bc97206f3cb8c7
cb54154b105a2bd3938fbd99f1f2f5409c0be09dc5f64189f473ae905d264b25
275994ee93e0c779f3eb30d770eeabfcb5ab0b8a5da68cc28a07dfbdb46a1e08
6215cc716b9ed577d3c6cd34d57f2dc3ed93c9b6aaedf120d68a4532393e1056
d691d7f93c57a54462f90ca76528beec4bda1a40220e5d5fbf43986308f9013b
6ea213b27eb3e0e4243b3c872e7a07b7898d9f07ea205b8a50c30e62c7204602
4544d5dff957471972331532aaf34eb5644bc430f854dd6593177640e07e4f00
```

USB armory Mk II example TRNG run

[1] https://media.ccc.de/v/32c3-7441-the_plain_simple_reality_of_entropy

i.MX6UL driver: USB

```go
// addDTD configures an endpoint transfer descriptor as described in p3787, 56.4.5.2 Endpoint Transfer Descriptor (dTD), IMX6ULLRM.
func buildDTD(n int, dir int, ioc bool, data []byte) (dtd *dTD, dtdBuf *mem.AlignmentBuffer, pages *mem.AlignmentBuffer, err error) {
	size := len(data)

	if size > DTD_PAGES*DTD_PAGE_SIZE {
		return nil, nil, nil, errors.New("unsupported transfer size")
	}

	// p3809, 56.4.6.6.2 Building a Transfer Descriptor, IMX6ULLRM
	dtdBuf = mem.NewAlignmentBuffer(unsafe.Sizeof(dTD{}), 32)
	dtd = (*dTD)(unsafe.Pointer(dtdBuf.Addr))
	// interrupt on completion (ioc)
	reg.Set(&dtd.token, 15)
	// multiplier override (MultO)
	reg.SetN(&dtd.token, 10, 0b11, 0)
	// active status
	reg.Set(&dtd.token, 7)
	// total bytes
	reg.SetN(&dtd.token, 16, 0xffff, uint32(size))

	pages = mem.NewAlignmentBuffer(DTD_PAGE_SIZE*DTD_PAGES, DTD_PAGE_SIZE)
	mem.Copy(pages, data)

	for n := 0; n < DTD_PAGES; n++ {
		dtd.buffer[n] = pages.Addr + uintptr(DTD_PAGE_SIZE*n)
	}

	// invalidate next pointer
	dtd.next = (*dTD)(unsafe.Pointer(uintptr(1)))

	return
}
```

Example of Endpoint Transfer Descriptor (dTD) configuration.

The custom `NewAlignmentBuffer` class is used to cast a structure over the right offset of a byte array to honor alignment requirements.

To keep GC happy we must preserve the underlying buffer as needed.

These two concerns, along with flushing memory caches when needed, is the only non-conventional (for Go code) aspects that needs to be taken care of when building drivers in Go.

Using Go goroutines, channels, mutexes, interfaces freely in low level drivers is a delight!

https://github.com/inversepath/tamago/tree/master/imx6/usb

i.MX6UL driver: USB networking

```go
func configureEthernetDevice(device *usb.Device) {
	// Supported Language Code Zero: English
	device.SetLanguageCodes([]uint16{0x0409})

	// device descriptor
	device.Descriptor = &usb.DeviceDescriptor{}
	device.Descriptor.SetDefaults()
	device.Descriptor.DeviceClass = 0x2
	device.Descriptor.VendorId = 0x0525
	device.Descriptor.ProductId = 0xa4a2
	device.Descriptor.Device = 0x0001
	device.Descriptor.NumConfigurations = 1

	iManufacturer, _ := device.AddString(`TamaGo`)
	device.Descriptor.Manufacturer = iManufacturer

	iProduct, _ := device.AddString(`RNDIS/Ethernet Gadget`)
	device.Descriptor.Product = iProduct

	iSerial, _ := device.AddString(`0.1`)
	device.Descriptor.SerialNumber = iSerial

	// device qualifier
	device.Qualifier = &usb.DeviceQualifierDescriptor{}
	device.Qualifier.SetDefaults()
	device.Qualifier.DeviceClass = 2
	device.Qualifier.NumConfigurations = 2
}
```

```go
func configureECM(device *usb.Device) {
...
	conf.Interfaces = append(conf.Interfaces, iface)

	ep1IN := &usb.EndpointDescriptor{}
	ep1IN.SetDefaults()
	ep1IN.EndpointAddress = 0x81
	ep1IN.Attributes = 2
	ep1IN.MaxPacketSize = 512
	ep1IN.Function = ECMTx

	iface.Endpoints = append(iface.Endpoints, ep1IN)

	ep1OUT := &usb.EndpointDescriptor{}
	ep1OUT.SetDefaults()
	ep1OUT.EndpointAddress = 0x01
	ep1OUT.Attributes = 2
	ep1OUT.MaxPacketSize = 512
	ep1OUT.Function = ECMRx

	iface.Endpoints = append(iface.Endpoints, ep1OUT)
}
```

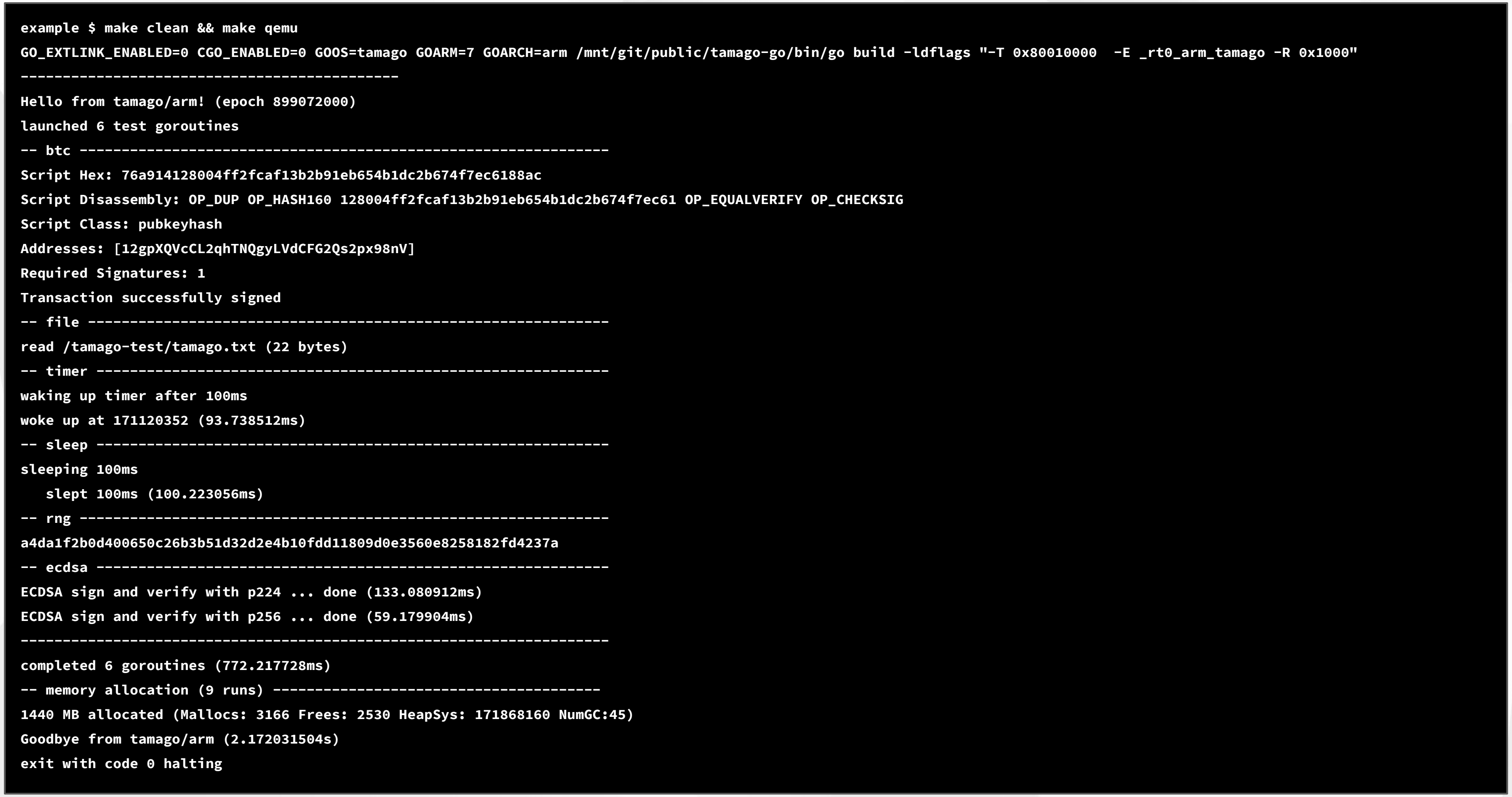
```go
func ECMTx(_ []byte, lastErr error) (in []byte) {
	// gvisor tcpip channel link
	pkt := <-link.C:
...
	// Ethernet frame header
	in = append(in, hostMAC...)
	in = append(in, deviceMAC...)
	in = append(in, proto...)
	// packet header
	in = append(in, hdr...)
	// payload
	in = append(in, payload...)

	return
}

func ECMRx(out []byte, lastErr error) ([]byte) {
...
	pkt := tcpip.PacketBuffer{
		LinkHeader: hdr,
		Data:       payload,
	}

	// gvisor tcpip channel link
	link.InjectInbound(proto, pkt)

	return
}
```

Example USB Ethernet (CDC ECM) driver integrated with Google netstack (`gvisor.dev/gvisor/pkg/tcpip`) for pure Go networking.

Developed in less than 2 hours and few LOC.

https://github.com/inversepath/tamago/blob/master/example/usb_ethernet.go

Demo



```
example $ make clean && make qemu
GO_EXTLINK_ENABLED=0 CGO_ENABLED=0 GOOS=tamago GOARM=7 GOARCH=arm /mnt/git/public/tamago-go/bin/go build -ldflags "-T 0x80010000  -E _rt0_arm_tamago -R 0x1000"
--------------------------------------------
Hello from tamago/arm! (epoch 899072000)
launched 6 test goroutines
-- btc -------------------------------------------------------------
Script Hex: 76a914128004ff2fcaf13b2b91eb654b1dc2b674f7ec6188ac
Script Disassembly: OP_DUP OP_HASH160 128004ff2fcaf13b2b91eb654b1dc2b674f7ec61 OP_EQUALVERIFY OP_CHECKSIG
Script Class: pubkeyhash
Addresses: [12gpXQVcCL2qhTNQgyLVdCFG2Qs2px98nV]
Required Signatures: 1
Transaction successfully signed
-- file ------------------------------------------------------------
read /tamago-test/tamago.txt (22 bytes)
-- timer -----------------------------------------------------------
waking up timer after 100ms
woke up at 171120352 (93.738512ms)
-- sleep -----------------------------------------------------------
sleeping 100ms
   slept 100ms (100.223056ms)
-- rng -------------------------------------------------------------
a4da1f2b0d400650c26b3b51d32d2e4b10fdd11809d0e3560e8258182fd4237a
-- ecdsa -----------------------------------------------------------
ECDSA sign and verify with p224 ... done (133.080912ms)
ECDSA sign and verify with p256 ... done (59.179904ms)
--------------------------------------------------------------------
completed 6 goroutines (772.217728ms)
-- memory allocation (9 runs) ---------------------------------------
1440 MB allocated (Mallocs: 3166 Frees: 2530 HeapSys: 171868160 NumGC:45)
Goodbye from tamago/arm (2.172031504s)
exit with code 0 halting
```

Performance

Go code runs (expectedly) with identical, or improved, speed compared to the same code executed under a full blown OS.

TamaGo is in early stages of development and drivers are yet to be optimized, serious overhead is not expected and anyway absolute performance is not a main focus of the effort, which remains security oriented.

Go ECDSA testsuite[1] under Linux

```
    ECDSA sign and verify with p224 ... done (116.069971ms)
    ECDSA sign and verify with p256 ... done (46.654623ms)
    ECDSA sign and verify with p384 ... done (1.894097668s)
    ECDSA sign and verify with p521 ... done (3.60261379s)
```

Same testsuite compiled with TamaGo

```
    ECDSA sign and verify with p224 ... done (115.087625ms)
    ECDSA sign and verify with p256 ... done (48.71225ms)
    ECDSA sign and verify with p384 ... done (1.859368625s)
    ECDSA sign and verify with p521 ... done (3.484454s)
```

[1] https://github.com/golang/go/blob/go1.13.6/src/crypto/ecdsa/ecdsa_test.go#L124

Current limitations

The TamaGo runtime is single threaded therefore:

- avoid[1] tight loops without function calls
- avoid deadlocks (e.g. do not sleep in `main()` if nothing else is happening)

File operations currently work on a volatile in-memory virtual filesystem.

Packages/applications which rely on unsupported system calls do not compile (e.g. terminal prompt packages that require `syscall.SYS_IOCTL`), though usually such packages do not make sense in the context of OS-less unikernel operations.

Importing libraries that require `cgo` can only be done with internal linking, integrating C code with `cgo` is possible as long as such code is free standing.

There is no OS, there are no users, there are no signals, there are no environment variables. This is a feature, not a bug.

With the exception of few surprises[2] Go is surprisingly adept to run on bare metal.

[1] or just force `runtime.Gosched`

[2] Here's a fun Go bug: https://play.golang.org/p/RIMIZDWEcZT

Applications and future

Access the Go crypto library without an underlying OS, the added benefit of i.MX6UL SoC drivers allow device specific key derivation/encryption.

We are growing i.MX6UL specific I/O capabilities (e.g. UART, USB, BLE, etc.), storage (eMMC) and filesystem (FAT) support. We are also actively working on Raspberry Pi Zero board support. In time we shall try[2] for upstream adoption of Go compiler patches.

TamaGo laids out the foundation for development of pure Golang
HSMs,
cryptocurrency wallets,
authentication tokens,
TrustZone secure monitors,
and much more…

It is our policy to keep comments and references (document title and page number) for all low level interactions within drivers, TamaGo source code is a great tool to learn on low level SoC development!

We plan to base new security apps and port our existing INTERLOCK[1] one with TamaGo.

[1] https://github.com/inversepath/interlock

[2] We are trying real hard to keep things clean, separate and nice.

What have we[1] learned?

Bare metal applications can play a big role in
the future of secure embedded systems and
can be built by **reducing complexity**.

We feel the need for a paradigm shift and think
there is no place for C code in complex drivers or applications anymore.

Go is a language that, among others, can definitely play a role in this.

To achieve trust we proved that
Go compiler modifications can be minimal to achieve
bare metal execution.

We completely **killed C** in runtime[2] execution.

It’s all about enabling choice
and building trust.

[1] “We” as in the authors, but maybe the audience as well.

[2] Only the bootloader remains, we are working on it.

Thanks!

The F-Secure Hardware Security Team[1]

Andrej Rosano
Daniele Bianco
Thierry Decroix
Dmitry Janushkevich
Emma Raeis

Roberto Clapis[2] from Google

Jyrki Tulokas and **Johanna Orjatsalo** from F-Secure

Tomi Tuominen from T2.fi

[1] https://foundry.f-secure.com

[2] https://twitter.com/empijei

TamaGo

Q & A ?

Andrea Barisani

Head of Hardware Security - F-Secure

@AndreaBarisani - andrea.bio

andrea.barisani@f-secure.com - foundry.f-secure.com