TamaGo

Bare metal Go for ARM/RISC-V SoCs

Secure embedded unikernels
with drastically reduced attack surface

Andrea Barisani

Head of Hardware Security - WithSecre

@AndreaBarisani - andrea.bio

andrea.barisani@withsecure.com - foundry.withsecure.com

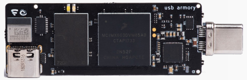
$ whoami

Andrea Barisani

Information Security Researcher

Founder of INVERSE PATH (acquired in 2017)

Head of Hardware Security at W/

Maker of the USB armory 

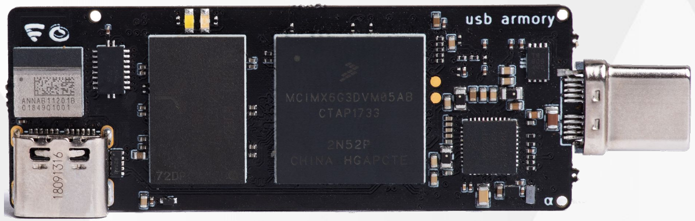
Speaker at too many conferences...

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

twitter: @AndreaBarisani

web: https://andrea.bio

Motivation: USB armory firmware



The USB armory is a tiny, but powerful, embedded platform for personal security applications.

Designed to fit in a pockets, laptops, PCs and servers.

The USB armory targets the following primary applications:

- Encrypted storage solutions
- Hardware Security Module (HSM)
- Enhanced smart cards
- Electronic vaults (e.g. cryptocurrency wallets) and key escrow services
- Authentication, provisioning, licensing tokens
- USB firewall

open hardware

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

WITH secure

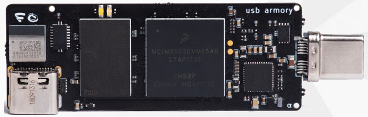
In an ideal world using **higher level languages should not entail complex dependencies**.

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

WITH secure

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.

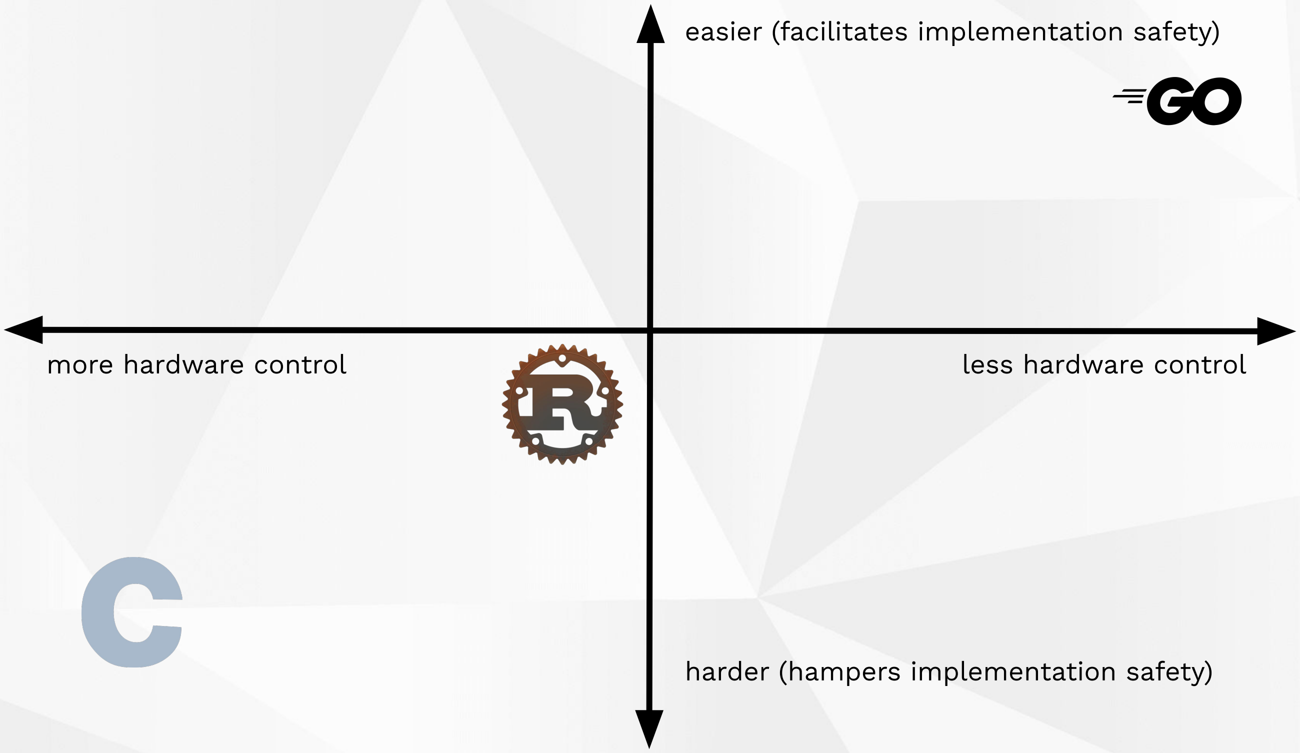
However high level programming typically entails several layers (e.g. OS, libraries) to serve runtime execution.

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**.

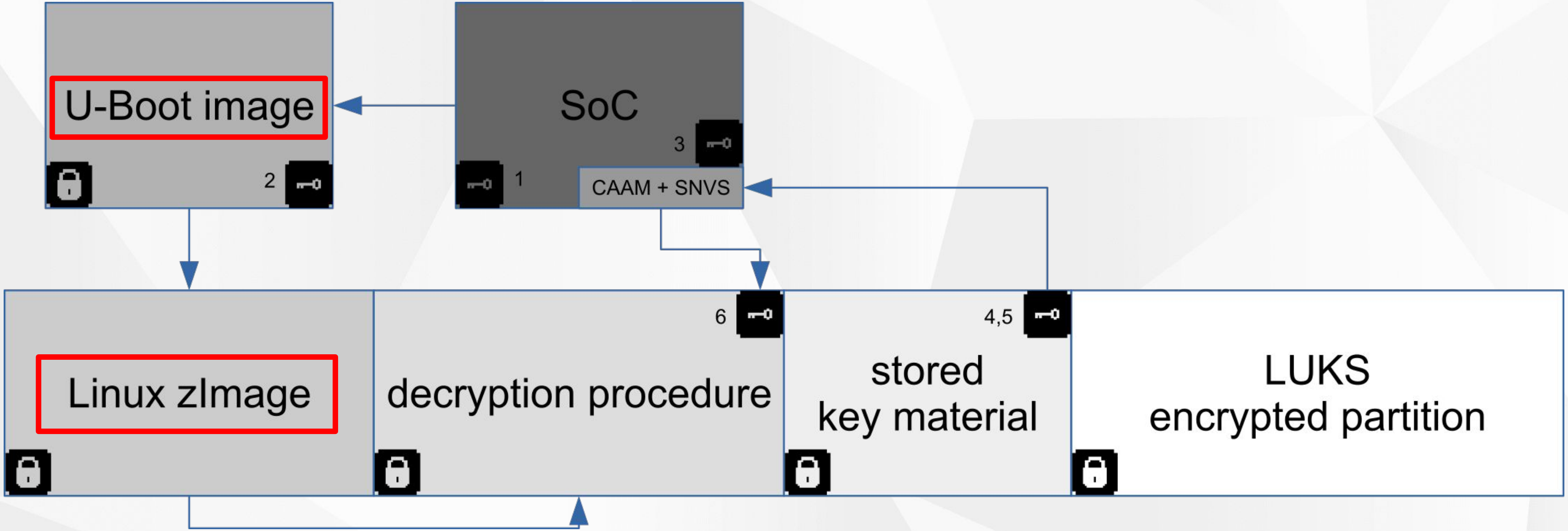
Audience mind reading trick: you are thinking “why not Rust?” … well why not *both* ?

Speed vs Safety



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

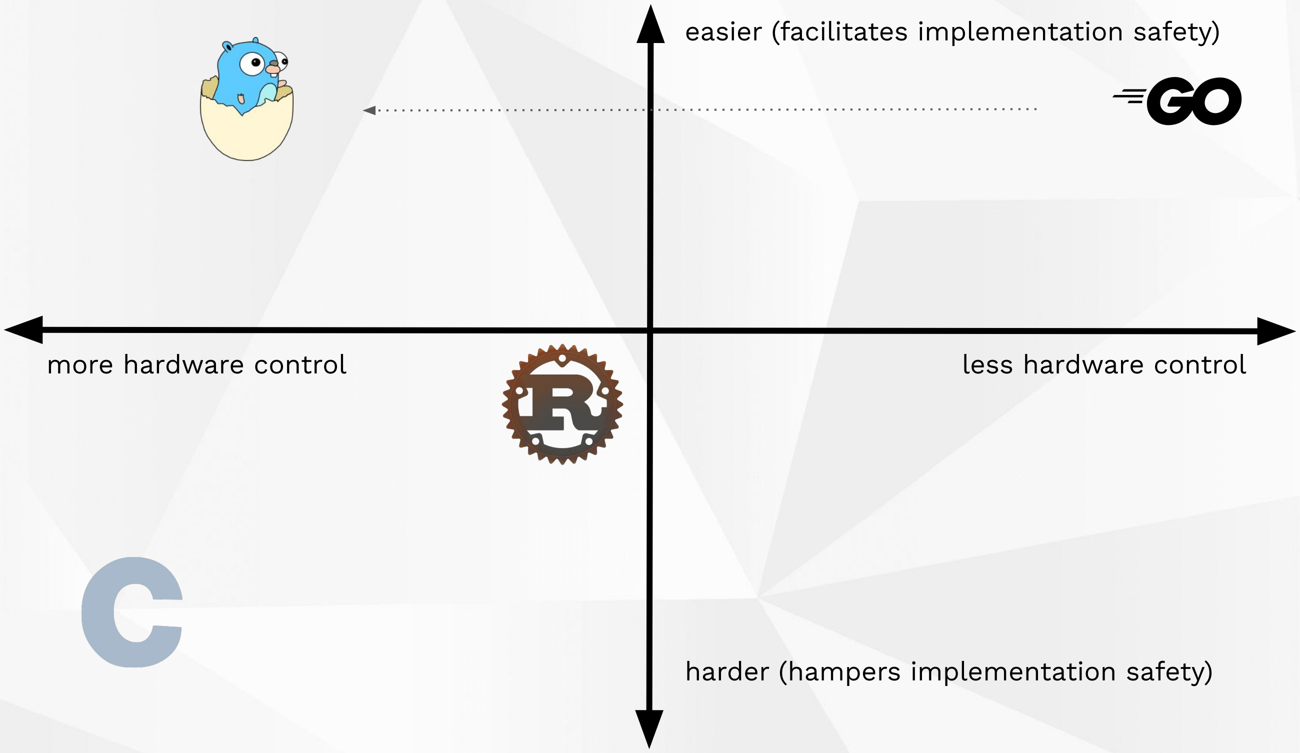
Reducing the attack surface



Typical secure booted firmware with authentication and confidentiality, taken from USB armory implementation example (NXP i.MX6UL).

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

Speed vs 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

W / T H
secure

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

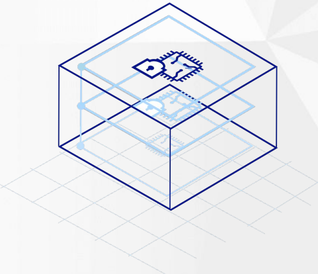
The focus is reducing the attack surface, carrying only strictly necessary code.

“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 … and it really can!

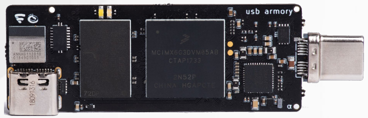
[1] https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2019/april/assessing-unikernel-security/

TamaGo in a nutshell

TamaGo is made of two main components.

- A **minimally**[1] patched Go distribution to enable `GOOS=tamago` support, which provides freestanding execution on `GOARCH=arm` and `GOARCH=riscv64` bare metal.
- A set of packages[2] to provide board support (e.g. hardware initialization and drivers).

TamaGo currently provides drivers for SoC families NXP i.MX6UL (USB armory Mk II), BCM2835 (Raspberry Pi Zero, Pi 1, Pi 2) and SiFive FU540.



On the i.MX6UL we target development of security applications, TamaGo is fully integrated with our existing open source tooling for i.MX6 Secure Boot (HAB) image signing.

TamaGo also provides full hardware initialization removing the need for intermediate bootloaders.

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

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

Similar efforts

Past/existing efforts

(all these projects greatly supported us in proving feasibility and identify TamaGo unique approach, diversity is good)

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

Go kernel for non-Go software underneath, larger scope, needs two C bootloaders, 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, needs two C bootloaders, hijacks `GOOS=linux` for `GOARCH=arm`, redoes memory allocation and threading.

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

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

TinyGo (active) - 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.

Newer efforts

Embedded Go (active) - https://github.com/embeddedgo

Similar to TamaGo but targets ARMv7-M/ARMv8-M (w/ Thumb2) adding new support for it, as not native to Go. Adds `GOOS=noos GOARCH=thumb`, features interrupt/timer support.

Egg OS (active) - https://github.com/icexin/eggos

Targets x86, uses native compiler and wraps `GOOS=linux` syscalls back to Go.

https://github.com/usbarmory/tamago/wiki/Frequently-Asked-Questions-(FAQ)#how-does-tamago-differ-from-pastcurrent-similar-efforts

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 distribution[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.
- ★ ~4500 LOC of changes against Go distribution 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 peripheral drivers for SoCs.

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

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

Go distribution modifications[1]

Glue code (~350 LOCs, ~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 code (~3000 LOCs, ~10 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 (raw SD/MMC access implemented in `imx6`)

New code (~600 LOCs, 12 files) - basic syscall and memory layout support:

```
rt0_tamago_arm.s   (LOC: ~40)        rt0_tamago_riscv64.s   (LOC: ~40)
sys_tamago_arm.go  (LOC: ~120)       sys_tamago_riscv64.go  (LOC: ~50)
os_tamago_arm.go   (LOC: ~200)       os_tamago_riscv64.go   (LOC: ~170)
```

https://github.com/golang/go/compare/go1.18...usbarmory:tamago1.18

[1] As of `tamago1.18` against `go1.18`

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)
|                                   |
|      L2 PAGE TABLE (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.ramStackOffset)
|                                   |
|              UNUSED               |
|                                   |
+-----------------------------------+ runtime.ramStart + runtime.ramSize
|                                   |
|                                   |
+-----------------------------------+ FFFF FFFF
```

Board packages and applications are free to override `ramStart`, `ramSize`, `dmaStart` and `dmaSize` if required.

```
+-----------------------------------+ mem.dmaStart
|                                   |
|            DMA BUFFERS            |
|                                   |
+-----------------------------------+ mem.dmaStart + mem.dmaSize
```

https://github.com/usbarmory/wiki/Internals

Go runtime support

```go
// 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 exceptionHandler()
func getRandomData([]byte)
func initRNG()
func nanotime1() int64
```

MMU initialization and exception handling are all performed outside the Go runtime in tamago architecture (e.g. `arm`) package.

This means low-level APIs (e.g. TrustZone) can all be implemented as a regular package.

The Go runtime modification is architecture independent for the most part.

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

```go
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
```

```go
package usbarmory

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

//go:linkname printk runtime.printk
func printk(c byte) {
        imx6ul.UART2.Write(c)
}
```

Go runtime support

WITH secure

os_tamago_arm.go (Go runtime)

```go
//go:linkname syscall_now syscall.now
func syscall_now() (sec int64, nsec int32) {
	sec, nsec, _ = time_now()
	return
}
```

imx6.go (imx6 package)

```go
//go:linkname nanotime1 runtime.nanotime1
func nanotime1() int64 {
	return int64(ARM.TimerFn() * ARM.TimerMultiplier)
}
```

timer.s (arm package)

```
// func read_gtc() int64
TEXT ·read_gtc(SB),$0-8
	// 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 low level access

```go
import "github.com/usbarmory/tamago/internal/reg"

func setARMFreqIMX6ULL(hz uint32) (err error) {
	var div_select uint32
	var arm_podf uint32
	var uV uint32

	curHz := ARMFreq()
...
	// 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.MX6UL SoC ARM core clock frequency.

Go's `unsafe` can be easily identified to spot areas that require care (e.g. pointer arithmetic), it is currently used only in register and DMA memory manipulation primitives.

There are overall only 3 occurrences of unsafe used in `dma` and `reg` packages.

Applications are never required to use any `unsafe` function.

Go runtime support

W / T H secure

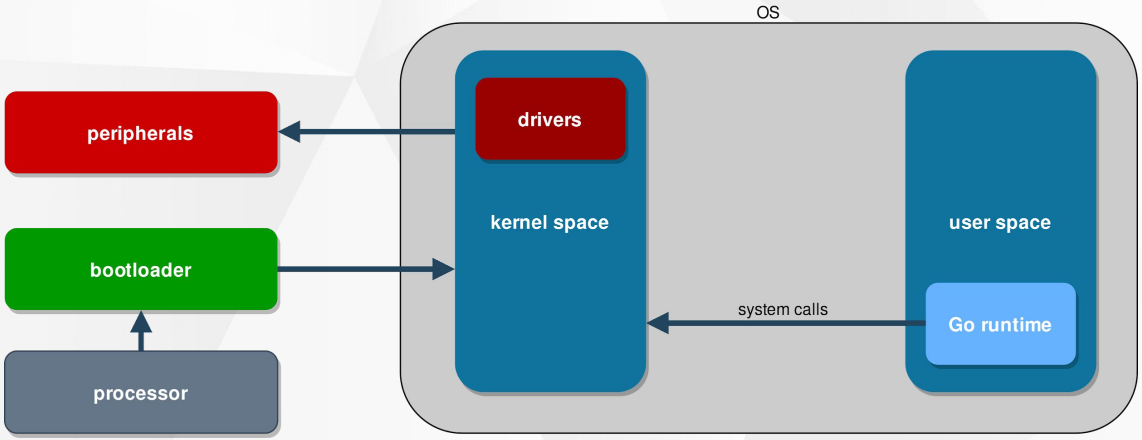
```go
//go:linkname syscall
func syscall(number, a1, a2, a3 uintptr) (r1, r2, err uintptr) {
	switch number {
	case 1: // SYS_WRITE
		r1 := write(a1, unsafe.Pointer(a2), int32(a3))
		return uintptr(r1), 0, 0
	default:
		throw("unexpected syscall")
	}

	return
}

//go:nosplit
func write1(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 the standard output writing function (e.g. UART).

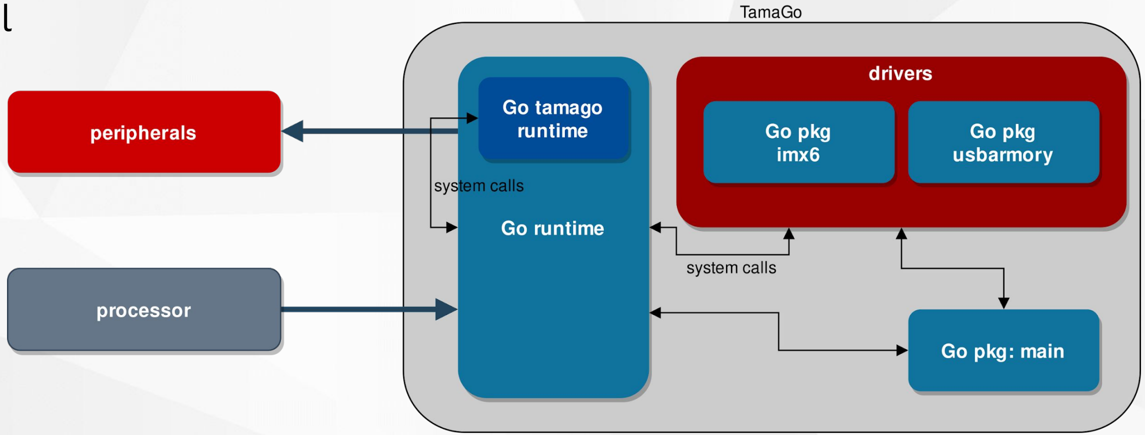
```
imx6_clk: changing ARM core frequency to 900 MHz
imx6_clk: changing ARM core operating point to 575000 uV
imx6_clk: 450000 uV -> 575000 uV
imx6_clk: waiting for PLL lock
imx6_clk: 396 MHz -> 900 MHz
imx6_soc: i.MX6ULL (0x65, 0.1) @ freq:900 MHz - native:true
```

TamaGo



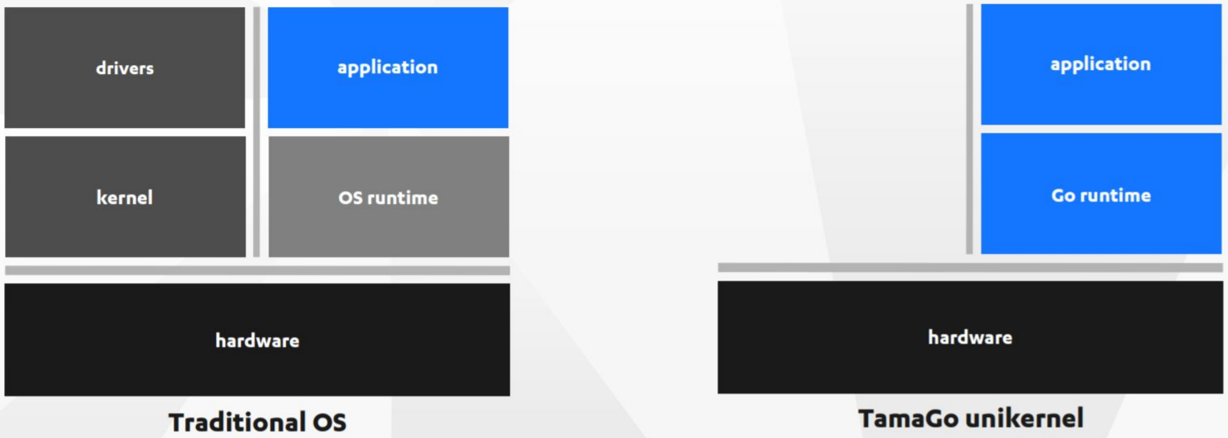
Traditional OS

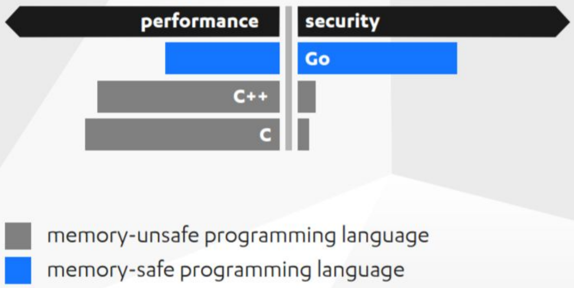
TamaGo unikernel



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

Enabling trust





TamaGo allows a dramatic reduction of the attack surface by **removing any dependency on memory-unsafe languages** (e.g. C), Operating Systems and third party libraries.

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/usbarmory/tamago/board/usbarmory/mk2"
)

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

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

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

=> bootelf -p 0x90000000
```

Examples shown for USB armory Mk II / i.MX6ULZ.

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

4a. The resulting ELF binary can be passed to a bootloader (e.g U-Boot).

4b. The SoC package supports native loading (no bootloader required!).

[1] https://github.com/usbarmory/tamago/wiki/Import-report

[2] https://rust-embedded.github.io/book/intro/no-std.html

i.MX6ULZ 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 ~250 LOC.

```
workPacket := WorkPacket{}
workPacket.Control0 |= (1 << 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 = dma.Alloc(diversifier, 0)
defer dma.Free(workPacket.SourceBufferAddress)

workPacket.DestinationBufferAddress = dma.Alloc(key, 0)
defer dma.Free(workPacket.DestinationBufferAddress)

workPacket.PayloadPointer = dma.Alloc(iv, 0)
defer dma.Free(workPacket.PayloadPointer)

buf := new(bytes.Buffer)
binary.Write(buf, binary.LittleEndian, &workPacket)

pkt := dma.Alloc(buf.Bytes(), 0)
defer dma.Free(pkt)

reg.Write(HW_DCP_CH0CMDPTR, pkt)
reg.Set(HW_DCP_CH0SEMA, 0)
```

```
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] if required.

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

[1] Use cgo `-godefs`.

i.MX6ULZ driver: Random Number Generator

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[1]). The driver takes ~140 LOC and is hooked as provider for crypto/rand.

```
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)
		}
	}
}
```

```
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.MX6ULZ driver: USB

```go
func buildDTD(n int, dir int, ioc bool, addr uint32, size int) (dtd *dTD) {
	dtd = &dTD{}

	// interrupt on completion (ioc)
	if ioc {
		bits.Set(&dtd.Token, 15)
	} else {
		bits.Clear(&dtd.Token, 15)
	}

	// invalidate next pointer
	dtd.Next = 0b1
	// multiplier override (MultO)
	bits.SetN(&dtd.Token, 10, 0b11, 0)
	// active status
	bits.Set(&dtd.Token, 7)
	// total bytes
	bits.SetN(&dtd.Token, 16, 0xffff, uint32(size))

	dtd._buf = addr
	dtd._size = uint32(size)

	for n := 0; n < DTD_PAGES; n++ {
		dtd.Buffer[n] = dtd._buf + uint32(DTD_PAGE_SIZE*n)
	}

	buf := new(bytes.Buffer)
	binary.Write(buf, binary.LittleEndian, dtd)
	dtd._dtd = dma.Alloc(buf.Bytes()[0:DTD_SIZE], DTD_ALIGN)

	return
}
```

Example of Endpoint Transfer Descriptor (dTD) configuration.

A custom DMA allocator is used to copy structures on memory reserved for DMA operation, with required alignements.

```go
addr = dma.Alloc(buf, align)
defer dma.Free(addr)
```

Buffers can be also reserved by the application to spare re-allocation (automatic detection of slices already in DMA memory).

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

All in ~1000 LOC !

https://github.com/usbarmory/tamago/tree/master/soc/imx6/usb

i.MX6ULZ driver: USB networking

W / T H secure

```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 ~300 LOC.

https://github.com/usbarmory/imx-usbnet

i.MX6ULZ driver: uSDHC (MMC/SD)

WITH secure

```
// p351, 35.4.5 SD card initialization flow chart, IMX6FG
// p57, 4.2.3 Card Initialization and Identification Process, SD-PL-7.10
func (hw *USDHC) initSD() (err error) {
	var arg uint32
	var bus_width uint32
	var mode uint32
	var root_clk uint32
	var clk int
	var tune bool


	if hw.LowVoltage == nil {
		hw.card.Rate = HS_MBPS
	} else if hw.card.Rate >= SDR50_MBPS {
		if err = hw.voltageSwitchSD(); err != nil {
			hw.card.Rate = HS_MBPS
		}
	}

	// CMD2 - ALL_SEND_CID - get unique card identification
	if err = hw.cmd(2, READ, arg, RSP_136, false, true, false, 0); err != nil {
		return
	}

	// CMD3 - SEND_RELATIVE_ADDR - get relative card address (RCA)
	if err = hw.cmd(3, READ, arg, RSP_48, true, true, false, 0); err != nil {
		return
	}
...
```

The uSDHC driver supports read/write operation on MMC/SD with speeds up to HS200 and SDR104 respectively.

All in ~1200 LOC !

It is used by armory-ums to allow export of the USB armory Mk II internal eMMC card as USB mass storage devices to ease firmware flashing.

In combination with packages such as `go-ext4` it allows filesystem access (see armory-boot).

https://github.com/usbarmory/usbarmory/tree/master/soc/imx6/usdhc

Demo

W / T H secure

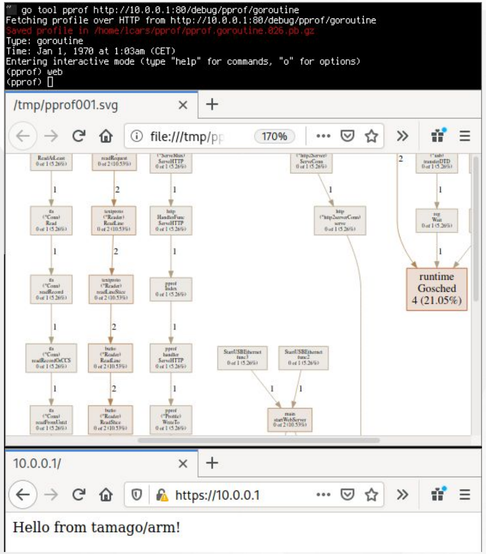
```
tamago-example $ make qemu

tamago/arm (go1.18.3) • 5e1f6aa lcars@lambda on 2022-07-14 08:02:11 • i.MX6UL 1188 MHz (emulated)

ble                                                      # BLE serial console
date                   (time in RFC339 format)?          # show/change runtime date and time
dcp                    <size> <sec>                      # benchmark hardware encryption
dns                    <fqdn>                            # resolve domain (requires routing)
exit, quit                                               # close session
help                                                     # this help
i2c                    <n> <hex target> <hex addr> <size>  # I²C bus read
info                                                     # device information
kem                                                      # benchmark post-quantum KEM
led                    (white|blue) (on|off)             # LED control
md                     <hex offset> <size>               # memory display (use with caution)
mmc                    <n> <hex offset> <size>           # MMC/SD card read
mw                     <hex offset> <hex value>          # memory write (use with caution)
otp                    <bank> <word>                     # OTP fuses display
rand                                                     # gather 32 random bytes
reboot                                                   # reset device
stack                                                    # stack trace of current goroutine
stackall                                                 # stack trace of all goroutines
test                                                     # launch tests

> kem
Kyber1024 89248f2f33f7f4f7051729111f3049c409a933ec904aedadf035f30fa5646cd5 (287.799024ms)
Kyber768  a1e122cad3c24bc51622e4c242d8b8acbcd3f618fee4220400605ca8f9ea02c2 (209.114896ms)
Kyber512  e9c2bd37133fcb40772f81559f14b1f58dccd1c816701be9ba6214d43baf4547 (149.049056ms)

> rand
db7d46647880be1e51731177b6f73645b71ca504242c97758df3a86842d93236

> md 80000000 96
00000000  18 f0 9f e5 18 f0 9f e5  18 f0 9f e5 18 f0 9f e5  |................|
00000010  18 f0 9f e5 18 f0 9f e5  18 f0 9f e5 18 f0 9f e5  |................|
00000020  04 d4 0f 80 38 d4 0f 80  6c d4 0f 80 a0 d4 0f 80  |....8...l.......|
00000030  d4 d4 0f 80 00 00 00 00  08 d5 0f 80 3c d5 0f 80  |............<...|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|

> date
1970-01-01T00:03:31Z
> date 2022-07-14T08:02:11Z
2022-07-14T08:02:11Z

> info
Runtime ......: go1.18.3 tamago/arm
Board ........: UA-MKII-β
SoC ..........: i.MX6UL 1188 MHz (emulated)
SDP ..........: false
Secure boot ..: false
Boot ROM hash : c6aeae82d3a49e6ce016e1f02fa93c918d50934f93847ae371816e5fdeb79dd5
```

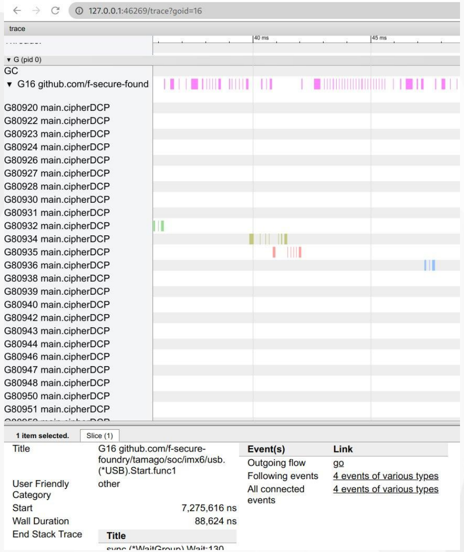
```
$ ssh 10.0.0.1

tamago/arm (go1.18.3) • 5e1f6aa lcars@lambda on 2022-07-14 10:00:48 • i.MX6ULL 900 MHz

> otp 0 0
OTP bank:0 word:0 val:0x00324003

 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00  OCOTP_LOCK
                                                                                                  Bank:0 Word:0
| 0| 0|  |  |  |  | 0|  | 0| 0| 1| 1| 0  0| 1| 0| 0| 1| 0  0| 0  0| 0  0|  | 0| 0  0| 0  0| 1  1|  R: 0x00000000
                                                                                                  W: 0x00000000
 31 ----------------------------------------------------------------------------------------------- GP3_RLOCK
    30 -------------------------------------------------------------------------------------------- GP4_RLOCK
                   25 ----------------------------------------------------------------------------- PIN_LOCK
                         23 ----------------------------------------------------------------------- GP4_LOCK
                            22 -------------------------------------------------------------------- MISC_CONF_LOCK
                               21 ----------------------------------------------------------------- ROM_PATCH_LOCK
                                  20 -------------------------------------------------------------- OTPMK_CRC_LOCK
                                     19 18 -------------------------------------------------------- ANALOG_LOCK
                                           17 ----------------------------------------------------- OTPMK_LOCK
                                              16 -------------------------------------------------- SW_GP_LOCK
                                                 15 ----------------------------------------------- GP3_LOCK
                                                    14 -------------------------------------------- SRK_LOCK
                                                       13 12 -------------------------------------- GP2_LOCK
                                                             11 10 -------------------------------- GP1_LOCK
                                                                   09 08 -------------------------- MAC_ADDR_LOCK
                                                                            06 -------------------- SJC_RESP_LOCK
                                                                               05 04 -------------- MEM_TRIM_LOCK
                                                                                     03 02 -------- BOOT_CFG_LOCK
                                                                                           01 00    TESTER_LOCK

> dns www.golang.org
;; opcode: QUERY, status: NOERROR, id: 28248
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.golang.org.        IN        ANY

;; ANSWER SECTION:
www.golang.org.        3600        IN        CNAME        golang.org.

> dcp 65536 10
Doing aes-128 cbc for 10s on 65536 blocks
6201 aes-128 cbc's in 10.00086575s

> info
Runtime ......: go1.18.4 tamago/arm
Board ........: UA-MKII-β
SoC ..........: i.MX6ULL 900 MHz
SDP ..........: true
Secure boot ..: true
Boot ROM hash : 1727a0f46dbde555b583e9a138ae359389974b7be4369ffd4a252a8730f7e59b
```

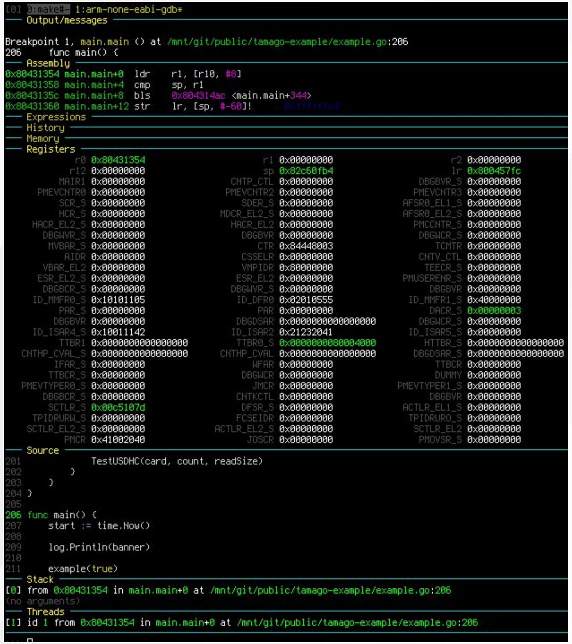
https://github.com/usbarmory/tamago-example

Debugging

W/TH secure







GDB can be used as usual, on emulated (QEMU) targets or real ones (JTAG).

On networked targets, such as the USB armory, the `pprof` package can be used as usual for tracing.

GoKey - The bare metal Go smart card

The GoKey application implements a composite USB OpenPGP 3.4 smartcard and FIDO U2F token, written in pure Go (~2500[1] LOC).

It allows to implement a radically different security model for smartcards, taking advantage of TamaGo to safely mix layers and protocols not easy to combine.

For instance authentication can happen over SSH instead of plaintext PIN transmission over USB.

	Trust anchor	Data protection	Runtime	Application	Requires tamper proofing	Encryption at rest
traditional smartcard	flash protection	flash protection	JCOP	JCOP applets	Yes	No
USB armory with GoKey	secure boot	SoC security element	TamaGo	Go application	No	Yes

GoKey

```
host ➤ gpg --card-status
Reader ...........: USB armory Mk II [Smart Card Control] (0.1) 00 00
Application ID ...: D276000124010304F5ECD209320C0000
Application type .: OpenPGP
Version ..........: 3.4
Manufacturer .....: F-Secure
Serial number ....: D209320C
Name of cardholder: Alice
Language prefs ...: [not set]
Salutation .......:
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: forced
Key attributes ...: rsa4096 rsa4096 rsa4096
Max. PIN lengths .: 254 127 127
PIN retry counter : 1 0 0
Signature counter : 0
Signature key ....: 05EC DEB4 43FA 5C01 9C7A  51A2 E9C8 5194 3E46 C2B5
      created ....: 2020-04-03 15:10:30
Encryption key....: 656B E354 EE12 BFFB 988B  1607 556B 9659 5A2C D776
      created ....: 2020-04-03 15:01:49
Authentication key: [none]
General key info..: sub  rsa4096/E9C851943E46C2B5 2020-04-03 Alice <alice@wonderl
and>
sec#  rsa4096/CBBE74C25E15EA0B  created: 2020-04-03  expires: 2022-04-03
ssb>  rsa4096/556B96595A2CD776  created: 2020-04-03  expires: 2022-04-03
                                card-no: F5EC D209320C
ssb>  rsa4096/E9C851943E46C2B5  created: 2020-04-03  expires: 2022-04-03
                                card-no: F5EC D209320C
ssb#  rsa4096/2EB31B5E996EE83D  created: 2020-04-03  expires: 2022-04-03
host ➤ ssh alice@10.0.0.10
GoKey • tamago/arm (go1.14) • 0330e82 user@host on 2020-04-09 07:42:11 • i.MX6ULL

  exit, quit                    # close session
  help                          # this help
  init                          # initialize card
  rand                          # gather 32 bytes from TRNG via crypto/rand
  reboot                        # restart
  status                        # display card status
  lock   (all|sig|dec)          # key lock
  unlock (all|sig|dec)          # key unlock, prompts decryption passphrase

resizing terminal (pty-req:80x66)
> unlock all
Passphrase:
VERIFY: 05 EC DE B4 43 FA 5C 01 9C 7A 51 A2 E9 C8 51 94 3E 46 C2 B5 unlocked
VERIFY: 65 6B E3 54 EE 12 BF FB 98 8B 16 07 55 6B 96 59 5A 2C D7 76 unlocked

> exit
logout
closing ssh connection
Connection to 10.0.0.10 closed.
host ➤ gpg --decrypt secret.asc
gpg: encrypted with 4096-bit RSA key, ID 556B96595A2CD776, created 2020-04-03
      "Alice <alice@wonderland>"
cheshire wrote:
        "Where do you want to go?"

alice wrote:
        "I don't know"

cheshire wrote:
        "Then, it really doesn't matter, does it?"
host ➤
```

https://github.com/usbarmory/gokey

[1] CCID: ~220 ICC: ~1000 U2F: 200

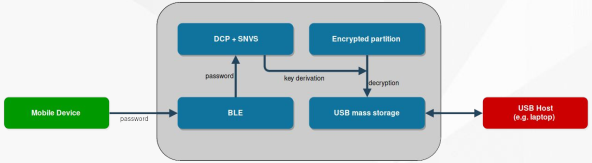
Demo: GoKey

https://github.com/usbarmory/gokey

https://youtu.be/WeO2eiYSeWM

Armory Drive - Encrypted USB Mass Storage

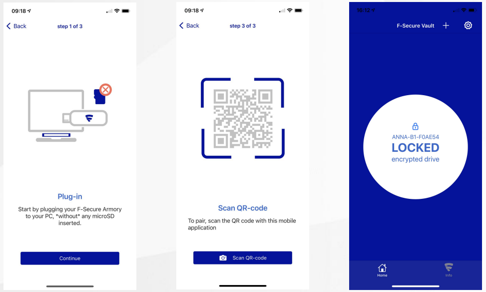
The application implements the **easiest to use encrypted drive solution** allowing secure access to any microSD card (no storage limit).

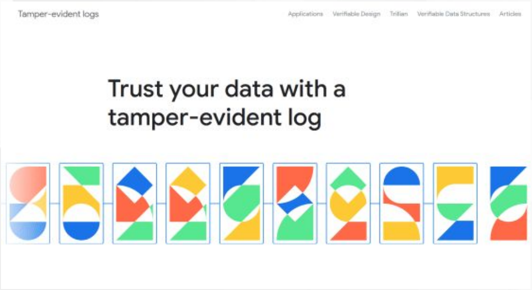


Unlike existing encrypted drive solutions the key is unlocked
With **3 factors** (user + mobile phone + armory) and **over Bluetooth**.

No trust (or driver requirements) are delegated to the host.

It consists of **~2400 LOC** of pure TamaGo code and an iOS app.





It uses Google Firmware Transparency framework to enable firmware update authentication on the installer as well as the device itself.

https://github.com/usbarmory/armory-drive

Armory Drive - Demo

https://github.com/usbarmory/armory-drive

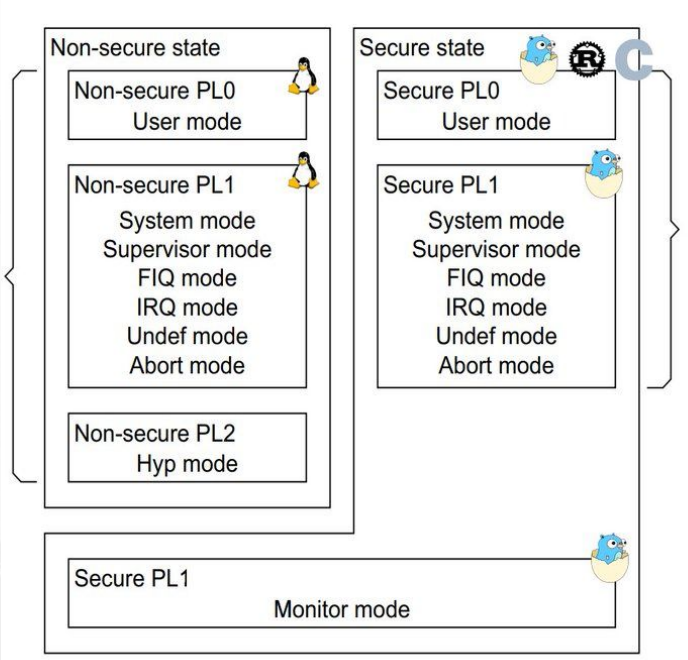
https://youtu.be/LAk1BBpwahM

GoTEE - Trusted Execution Environment

The GoTEE framework implements concurrent instantiation of TamaGo based unikernels in privileged and unprivileged modes, interacting with each other through monitor mode and custom system calls.

With these capabilities GoTEE implements a pure Go Trusted Execution Environment (TEE) bringing Go memory safety, convenience and capabilities to bare metal execution within TrustZone Secure World.

It supports any freestanding user mode applets (e.g. TamaGo, C, Rust) and any "rich" OS running in NonSecure World (e.g. Linux).



Demo: GoTEE

W / T H secure

```
> gotee
PL1 tamago/arm (go1.18.3) • TEE system/monitor (Secure World)
PL1 loaded applet addr:0x9c000000 size:4719809 entry:0x9c06f188
PL1 loaded kernel addr:0x80000000 size:4384184 entry:0x8006db70
PL1 starting mode:SYS ns:true sp:0x00000000 pc:0x8006db70
PL1 starting mode:USR ns:false sp:0x9e000000 pc:0x9c06f188
PL1 tamago/arm (go1.18.4) • system/supervisor (Normal World)
PL1 in Normal World is about to perform DCP key derivation
PL1 in Normal World successfully used DCP (df3eed2a50c9dd22daf7cf864f27bb90)
PL1 in Normal World is about to yield back
   r0:00000000  r1:814243f0  r2:00000001  r3:00000000
   r4:00000000  r5:00000000  r6:00000000  r7:8146bf14
   r8:00000007  r9:00000034 r10:814040f0 r11:802e9b21 cpsr:600001d6 (MON)
  r12:00000000  sp:8146bf54  lr:80185518  pc:80185648 spsr:600001df (SYS)
PL1 stopped mode:SYS ns:true sp:0x8146bf54 lr:0x80185518 pc:0x80185648 err:exit
PL0 tamago/arm (go1.18.4) • TEE user applet (Secure World)
PL0 obtained 16 random bytes from PL1: 10e742f0dad15db3f00aea14ee4a5acc
PL1 loaded kernel addr:0x80000000 size:4384184 entry:0x8006db70
PL1 re-launching kernel with TrustZone restrictions
PL1 starting mode:SYS ns:true sp:0x00000000 pc:0x8006db70
PL1 tamago/arm (go1.18.4) • system/supervisor (Normal World)
PL1 in Normal World is about to perform DCP key derivation
   r0:02280000  r1:814683a0  r2:8143c588  r3:00000001
   r4:00000000  r5:00000000  r6:00000000  r7:8146bf14
   r8:00000007  r9:00000044 r10:814040f0 r11:802e9b21 cpsr:200001d6 (MON)
  r12:00000000  sp:8146bf28  lr:80180398  pc:80011340 spsr:200001df (SYS)
PL1 stopped mode:SYS ns:true sp:0x8146bf28 lr:0x80180398 pc:0x80011340 err:DATA_ABORT
PL1 in Secure World is about to perform DCP key derivation
PL1 in Secure World World successfully used DCP (df3eed2a50c9dd22daf7cf864f27bb90)
```

```
$ ssh 10.0.0.1

PL1 tamago/arm (go1.18.3) • TEE system/monitor (Secure World)

  help                                  # this help
  reboot                                # reset the SoC/board
  stack                                 # stack trace of current goroutine
  stackall                              # stack trace of all goroutines
  md  <hex offset> <size>               # memory display (use with caution)
  mw  <hex offset> <hex value>          # memory write   (use with caution)

  gotee                                 # TrustZone test w/ TamaGo unikernels
  linux <uSD|eMMC>                      # boot NonSecure USB armory Debian image

  dbg                                   # show ARM debug permissions
  csl                                   # show config security levels (CSL)
  csl <periph> <slave> <hex csl>        #  set config security level  (CSL)
  sa                                    # show security access (SA)
  sa  <id> <secure|nonsecure>           #  set security access (SA)

> dbg
| type                    | implemented | enabled |
|-------------------------|-------------|---------|
| Secure non-invasive     |           1 |       0 |
| Secure invasive         |           1 |       0 |
| Non-secure non-invasive |           1 |       1 |
| Non-secure invasive     |           1 |       0 |

> linux eMMC
armory-boot: loading configuration at /boot/armory-boot-nonsecure.conf
PL1 loaded kernel addr:0x80000000 size:7603616 entry:0x80800000
PL1 launching Linux
PL1 starting mode:SVC ns:true sp:0x00000000 pc:0x80800000
Booting Linux on physical CPU 0x0
Linux version 5.15.52-0 (usbarmory@usbarmory) arm-linux-gnueabihf-gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1)
```

https://github.com/usbarmory/GoTEE/wiki/TrustZone-configuration

https://github.com/usbarmory/GoTEE-example

armory-boot - USB armory boot loader

A primary signed boot loader (~300 LOC) to launch authenticated Linux kernel images on secure booted[1] USB armory boards, replacing U-Boot.

```go
func boot(kernel []byte, dtb []byte, cmdline string) {
	dma.Init(dmaStart, dmaSize)
	mem, _ := dma.Reserve(dmaSize, 0)

	dma.Write(mem, kernel, kernelOffset)
	dma.Write(mem, dtb, dtbOffset)

	image := mem + kernelOffset
	params := mem + dtbOffset

	arm.ExceptionHandler = func(n int) {
		if n != arm.SUPERVISOR {
			panic("unhandled exception")
		}

		usbarmory.LED("blue", false)
		usbarmory.LED("white", false)

		imx6.ARM.DisableInterrupts()
		imx6.ARM.FlushDataCache()
		imx6.ARM.Disable()

		exec(image, params)
	})

	svc()
}
```

```go
func verifySignature(bin []byte, s []byte) (valid bool, err error) {
	sig, err := DecodeSignature(string(s))

	if err != nil {
		return false, fmt.Errorf("invalid signature, %v", err)
	}

	pub, err := NewPublicKey(PublicKeyStr)

	if err != nil {
		return false, fmt.Errorf("invalid public key, %v", err)
	}

	return pub.Verify(bin, sig)
}

func verifyHash(bin []byte, s string) bool {
	h := sha256.New()
	h.Write(bin)

	if hash, err := hex.DecodeString(s); err != nil {
		return false
	}

	return bytes.Equal(h.Sum(nil), hash)
}
```

https://github.com/usbarmory/armory-boot

[1] https://github.com/usbarmory/usbarmory/wiki/Secure-boot-(Mk-II)

Performance

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

TamaGo drivers operates comparably to their Linux counterparts, no serious overhead is present and anyway absolute performance is not a main focus of the effort, which remains security oriented.

Go ECDSA testsuite[1]	TamaGo	Linux
ECDSA sign+verify p224	115 ms	116 ms
ECDSA sign+verify p256	48 ms	46 ms
ECDSA sign+verify p384	1.85 s	1.89 s
ECDSA sign+verify p521	3.48 s	3.60 s

AES-128-CBC encryption w/ DCP	TamaGo	OpenSSL (afalg)
65536 blocks for 10s	6208	4528
4096 blocks for 10s	70326	60204

Go standard libraries run with comparable performance, while TamaGo hardware drivers highlight increased performance.

[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)

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 limitations[2] Go is surprisingly adept to run on bare metal.

[1] or just force `runtime.Gosched`

[2] https://github.com/usbarmory/tamago/wiki/Internals#go-application-limitations

Applications and future

TamaGo `imx6` package supports a wide variety of i.MX6 SoC drivers,
Raspberry Pi and RISC-V support is also available.

TamaGo lays 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 low level SoC development!

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 distribution modifications can be minimal to achieve bare metal execution.

We completely **killed C**[2].

It's all about enabling choice and building trust.

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

[2] The SoC boot ROM jumps directly to Go runtime.

Repository: https://github.com/usbarmory/tamago
Documentation: https://github.com/usbarmory/tamago/wiki
API: http://pkg.go.dev/github.com/usbarmory/tamago

Q & A

Andrea Barisani

Head of Hardware Security - WithSecure

@AndreaBarisani - andrea.bio

andrea.barisani@withsecure.com - foundry.withsecure.com