

TamaGo Bare metal Go for ARM SoCs

Secure embedded unikernels with drastically reduced attack surface

Andrea Barisani

Head of Hardware Security - F-Secure

@AndreaBarisani - andrea.bio

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





Andrea Barisani

Information Security Researcher

Founder of INVERSE PATH (acquired in 2017)

Head of Hardware Security at

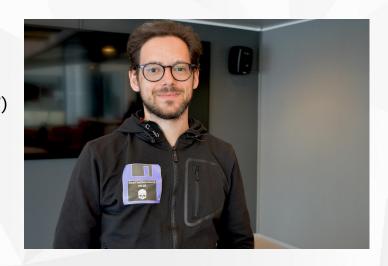


Maker of the USB armory



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





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 == unsafe¹ 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 unsafe¹) 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.

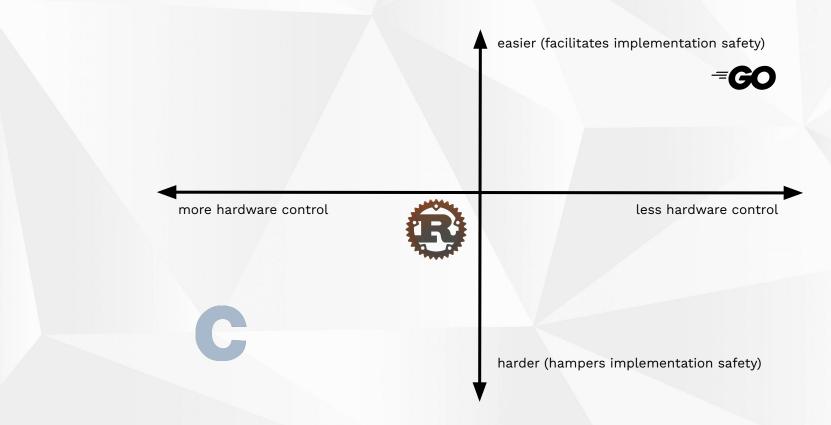
TamaGo spawns from the desire of **reducing the attack surface** of embedded systems firmware by **removing any runtime dependency on C code and inherently complex Operating Systems**.

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.



Speed vs Safety

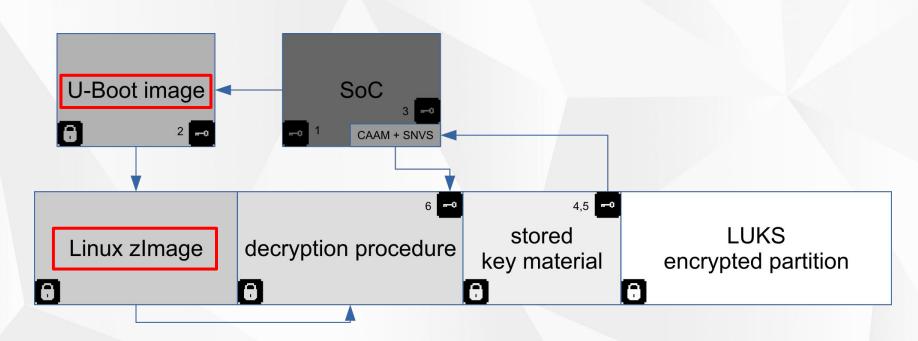






Reducing the attack surface



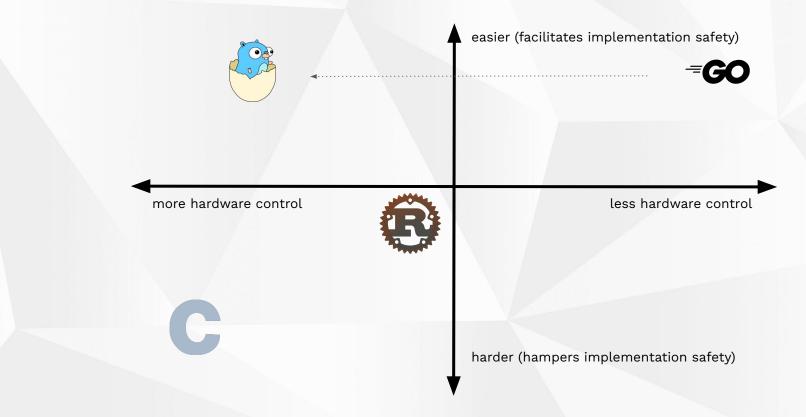


Typical secure booted firmware with authentication and confidentiality, taken from USB armory implementation example (NXP i.MX6UL).



Speed vs Safety







Unikernels / library OS



Unikernels¹ 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



Unikernel security



From a security standpoint leveraging on Unikernels (whatever the kind) to run multiple applications or an individual C applications is not ideal¹.

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 minimally¹ patched Go distribution to enable GOOS=tamago support, which provides freestanding execution on GOARCH=arm bare metal.
- A set of packages² to provide board support (e.g. hardware initialization and drivers).

TamaGo currently provides drivers for the NXP i.MX6UL System-on-Chip family (USB armory Mk II) as well as the BCM2835 (Raspberry Pi Zero, Pi 1, Pi 2).

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/f-secure-foundry/tamago-go

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.



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 distribution²**.

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.
- ★ ~4000 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 Internals¹.
- * 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.



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

Go distribution modifications¹



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-used² 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:

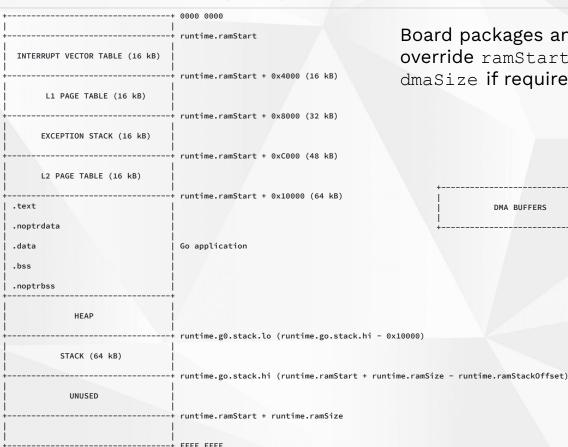
```
rt0_tamago_arm.s (LOC: ~30) sys_tamago_arm.s (LOC: ~130) rand tamago.go (LOC: ~20) os tamago arm.go (LOC: ~200)
```

https://github.com/golang/go/compare/go1.17...f-secure-foundry:tamago1.17



TamaGo memory layout





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





Go runtime support



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

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

ARM MMU initialization and exception handling are all performed outside the Go runtime in tamago 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.

```
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 = 0x200000000 // 512 MB

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



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()
                                  imx6.go (imx6 package)
//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
                   (R2), R4
         MOVW
                   (R1), R5
         MOVW
                   R5, R3
         BNE
                   R3, ret_hi+4(FP)
         MOVW
         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



```
import "github.com/f-secure-foundry/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



```
//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++ {</pre>
                    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.

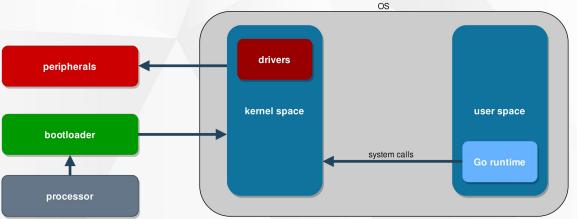
As shown before, printk is provided by the application to define method for writing on standard output (e.g. UART).

```
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
```

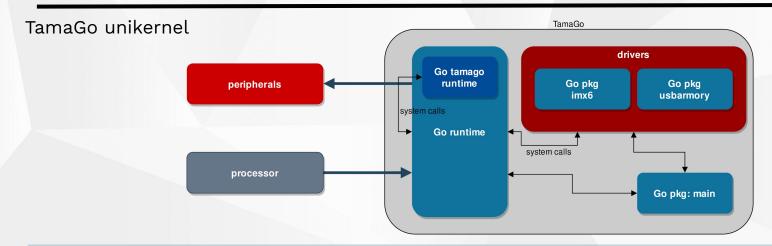


TamaGo





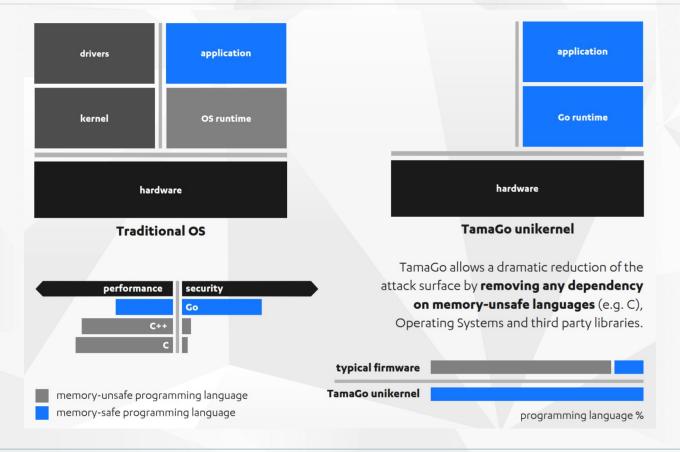
Traditional OS





Enabling trust







Developing, building and running



The full Go runtime is supported¹ without any specific changes required on the application side (Rust on bare metal², for comparison, requires #! [no std] pragma).

```
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.

- 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.
- 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 imx6 package supports imximage creation for native loading (no bootloader required!).



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 ~230 LOC.

```
workPacket := WorkPacket{}
workPacket.Control0 |= (1 << DCP_CTRL0_OTP_KEY)</pre>
workPacket.Control1 |= (AES128 << DCP_CTRL1_CIPHER_SELECT)</pre>
workPacket.Control1 |= (CBC << DCP_CTRL1_CIPHER_MODE)</pre>
workPacket.Control1 |= (UNIQUE_KEY << DCP_CTRL1_KEY_SELECT)</pre>
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(bvtes.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-compatible¹ if required.



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 else²). 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)
```

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



```
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 (Mult0)
         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++ {</pre>
                    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.

```
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!



i.MX6ULZ driver: USB networking



```
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 = 0 \times 0001
         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
```

```
func configureECM(device *usb.Device) {
          conf.Interfaces = append(conf.Interfaces, iface)
          ep1IN := &usb.EndpointDescriptor{}
          ep1IN.SetDefaults()
          eplIN.EndpointAddress = 0x81
          ep1IN.Attributes = 2
          ep1IN.MaxPacketSize = 512
          ep1IN.Function = ECMTx
          iface.Endpoints = append(iface.Endpoints, ep1IN)
          ep10UT := &usb.EndpointDescriptor{}
          ep10UT.SetDefaults()
          ep10UT.EndpointAddress = 0x01
          ep10UT.Attributes = 2
          ep10UT.MaxPacketSize = 512
          ep10UT.Function = ECMRx
          iface.Endpoints = append(iface.Endpoints, ep10UT)
```

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

```
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,
                                payload,
                    Data:
          // gvisor tcpip channel link
          link.InjectInbound(proto, pkt)
          return
```

Developed in less than 2 hours and ~150 LOC.



i.MX6ULZ driver: uSDHC (MMC/SD)



```
// 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).



Demo

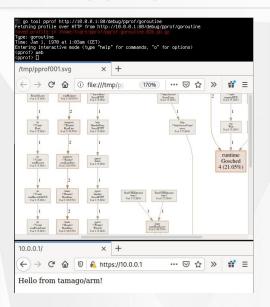


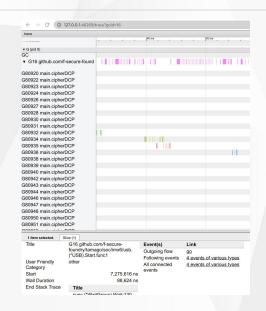
```
example $ make clean && make qemu
tamago/arm (gol.17.1) • cc683da lcars@lambda on 2021-09-27 09:55:13 • i.MX6UL 1188 MHz (emulated)
-- begin tests ------
launched 6 test goroutines
-- btc -----
Script Hex: 76a914128004ff2fcaf13b2b91eb654b1dc2b674f7ec6188ac
Script Disassembly: OP_DUP OP_HASH160 128004ff2fcaf13b2b91eb654b1dc2b674f7ec61 OP_EQUALVERIFY OP_CHECKSIG
Script Class: pubkeyhash
Addresses: [12gpXQVcCL2qhTNQgyLVdCFG2Qs2px98nV]
Required Signatures: 1
Transaction successfully signed
-- fs ------
listing /dev:
/dev/null (0 bytes)
reading 20 bytes from /dev/random
read /dev/random (20 bytes): 3e6fd8589f71000eec02c958b556023b4a41685b
writing 101 bytes to /dir/tamago.txt
read /dir/tamago.txt (101 bytes)
listing /dir:
/dir/tamago.txt (101 bytes)
-- timer -----
waking up timer after 100ms
-- sleep ------
sleeping 100ms
bb888478d861b96d31a9ce381628717497f5682584c34e326da73327a2af4f19
retrieved 32000 random bytes in 4.841888ms
-- ecdsa -----
ECDSA sign and verify with p256 took 53.115632ms
slept 100ms (194.65648ms)
woke up at 220747184 (195.173056ms)
completed 6 goroutines (284.942416ms)
-- memory allocation (9 runs) ------
allocating 9 * 17 MiB chunks (9/9)
1439 MiB allocated (Mallocs: 4784 Frees: 3545 HeapSys: 176062464 NumGC:47)
Goodbye from tamago/arm (2.318812s)
exit with code 0 halting
```



Debugging







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.

```
Breakpoint 1, main.main () at /mnt/git/public/tamago-example/example.go:286
       func main() (
     31354 main.main+0 ldr r1, [r10, #8]
     31358 main.main+4 cmp
3135c main.main+8 bls
                              sp, r1
      31360 main.main+12 str lr, [sp, #-60]
   Memory
Registers
                                                   0x00000000
                                                                                       0x00000000
                8×88888888
                                                   0x8088888
                                                                                       0x000000000
               8×86888888
                                                   0x00000000
                                                   0x84448003
                8×88888888
               0×10101105
                                                   0x02010555
               8×88888888
                                                   0x000000000000000000
                                                                                       ауаавааваа
                                                   0x21232841
               0×000000000
                                                                                       0x00000000
                                                   0x00000000
                                                                                       0x00000000
               0×000000000
                                                   0×80888886
                                                   0x00000000
                                                                                       0x00000000
               8×88888888
                                                   0x00000000
                                                                                       0x000000000
               0x41002040
                                                                                       0x00000000
               TestUSDHC(card, count, readSize)
    func main() (
       start := time.Now()
       log.Println(banner)
  l from 0x80431354 in main.main+0 at /mnt/git/public/tamago-example/example.go:206
 11 id 1 from 0x80431354 in main.main+0 at /mnt/git/public/tamago-example/example.go:206
```



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 (~2500¹ 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

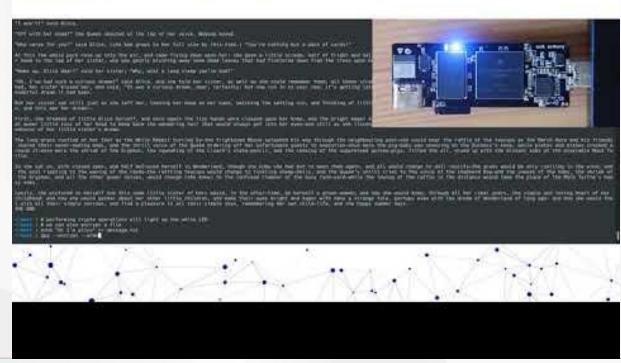
```
05EC DEB4 43FA 5C01 9C7A 51A2 E9C8 5194 3E46 C2B5
                656B E354 EE12 BFFB 988B 1607 556B 9659 5A2C D776
rsa4096/556B96595A2CD776 created: 2020-04-03 expires: 2022-04-03
                          card-no: F5EC D2093200
     rsa4096/E9C851943E46C2B5 created: 2020-04-03 expires: 2022-04-03
                          card-no: F5EC D2093200
                           # gather 32 bytes from TRNG via crypto/rand
                          # key unlock, prompts decryption passphrase
closing ssh connection
gpg: encrypted with 4096-bit RSA key, ID 556B96595A2CD776, created 2020-04-03
```



Demo: GoKey



F-Secure GoKey - the bare metal Go smart card



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).

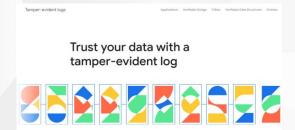
Unlike existing encrypted drive solutions the key is unlocked

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.



Armory Drive - Demo







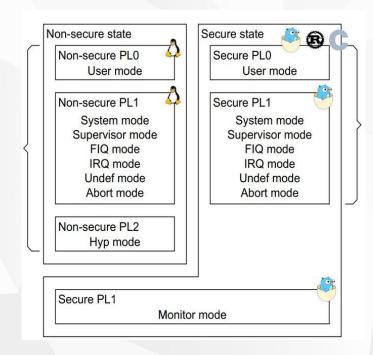
GoTEE - Trusted Execution Environment



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



```
PL1 tamago/arm (go1.16.4) • TEE system/monitor (Secure World)
                                                                                                     $ ssh gotee@10.0.0.1
PL1 loaded applet addr:0x82000000 size:3897203 entry:0x8206dab8
PL1 loaded kernel addr:0x84000000 size:3840614 entry:0x8406c6c4
                                                                                                     PL1 tamago/arm (go1.16.5) • TEE system/monitor (Secure World)
PL1 starting mode:USR ns:false sp:0x00000000 pc:0x8206dab8
PL1 starting mode:SYS ns:true sp:0x00000000 pc:0x8406c6c4
                                                                                                       help
                                                                                                                                            # this help
                                                                                                       reboot
                                                                                                                                            # reset the SoC/board
                                                                                                       stack
                                                                                                                                            # stack trace of current goroutine
PL1 in Normal World is about to perform DCP key derivation
                                                                                                       stackall
                                                                                                                                             # stack trace of all goroutines
                                                                                                       md <hex offset> <size>
                                                                                                                                             # memory display (use with caution)
PL1 in Normal World is about to yield back
                                                                                                       mw <hex offset> <hex value>
                                                                                                                                             # memory write (use with caution)
          r0:00000000 r1:848220c0 r2:00000001 r3:00000000
                                                                                                                                             # TrustZone test w/ TamaGo unikernels
                                                                                                       gotee
          r1:848220c0 r2:00000001 r3:00000000
                                                  r4:00000000
                                                                                                                                            # show config security levels (CSL)
          r5:00000000 r6:00000000 r7:00000000 r8:00000007
                                                                                                       csl <periph> <slave> <hex csl>
                                                                                                                                            # set config security level (CSL)
          r9:00000034 r10:848000e0 r11:802c2a48 r12:00000000
                                                                                                                                             # show security access (SA)
          sp:8484ff50 lr:841503c0 pc:8414a86c spsr:600c00df
                                                                                                       sa <id> <secure nonsecure>
                                                                                                                                            # set security access (SA)
PL1 stopped mode:SYS ns:true sp:0x8484ff50 lr:0x841503c0 pc:0x8414a86c err:exception mode MON
PLO tamago/arm (go1.16.4) • TEE user applet (Secure World)
PL1 re-launching kernel with TrustZone restrictions
PL1 starting mode:SYS ns:true sp:0x00000000 pc:0x8406c6c4
PL1 in Normal World is about to perform DCP key derivation
          r0:02280000 r1:8484e3a0 r2:00000001 r3:00000000
          r1:8484e3a0 r2:00000001 r3:00000000 r4:00000000
          r5:00000000 r6:00000000 r7:00000000 r8:00000007
          r9:00000044 r10:848000e0 r11:802c2a48 r12:00000000
          sp:8484ff34 lr:8414a990 pc:84011374 spsr:200c00df
PL1 stopped mode:SYS ns:true sp:0x8484ff34 lr:0x8414a990 pc:0x84011374 err:exception mode MON
PL1 in Secure World is about to perform DCP key derivation
PL1 in Secure World World successfully used DCP (e777b98dd28a4071a0c94821b7a1a4d1)
PL1 says goodbye
```



armory-boot - USB armory boot loader



A primary signed boot loader (~300 LOC) to launch authenticated Linux kernel images on secure booted¹ USB armory boards, replacing U-Boot.

```
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.RNGB.Reset()
                    imx6.ARM.DisableInterrupts()
                    imx6.ARM.FlushDataCache()
                    imx6.ARM.Disable()
                    exec(image, params)
          })
          svc()
```

```
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)
```



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 (cryptodev)	Linux userspace (AF_ALG)
65536 blocks for 10s	6143	4501	3138
4096 blocks for 10s	60985	56465	6578

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:

- avoid¹ 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 <code>syscall.SYS_IOCTL</code>), 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 limitations² Go is surprisingly adept to run on bare metal.



¹ or just force runtime. Gosched

² https://github.com/f-secure-foundry/tamago/wiki/Internals#go-application-limitations

Applications and future



TamaGo imx6 package supports a wide variety of i.MX6 SoC drivers, initial Raspberry Pi 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 we¹ 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.



Repository: https://github.com/f-secure-foundry/tamago

Documentation: https://github.com/f-secure-foundry/tamago/wiki

API: http://pkg.go.dev/github.com/f-secure-foundry/tamago

Q & A

Andrea Barisani

Head of Hardware Security - F-Secure

@AndreaBarisani - andrea.bio

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