TamaGo

Bare metal Go

Secure embedded unikernels
with drastically reduced attack surface

Andrea Barisani

https://andrea.bio

andrea@inversepath.com

andrea.barisani@reversec.com

$ whoami Andrea Barisani

Information Security Engineer and Researcher

Co-Founder: INVERSE PATH (2005 - 2017)

Head of Hardware Security:

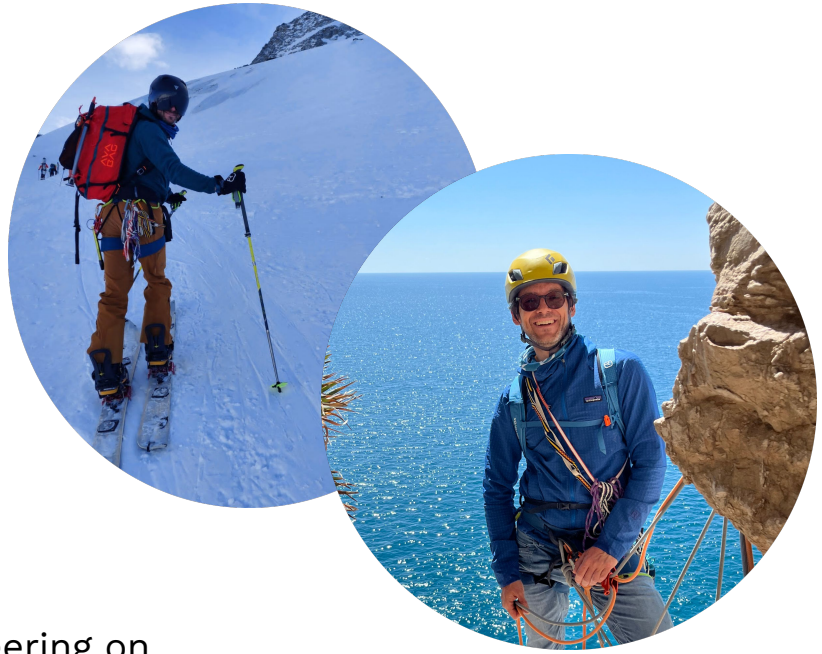
Head of Security Engineering:

Co-creator, USB armory:

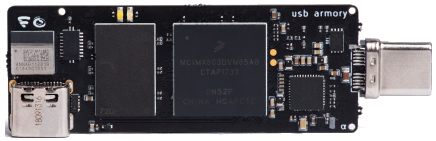
Speaker at too many conferences...

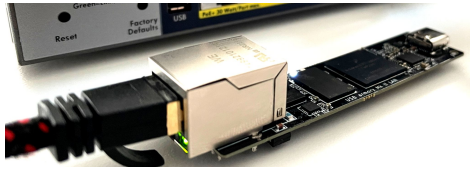
Background focused on security auditing and engineering on safety critical systems in the automotive, avionics, industrial domains.

Now also hacking compilers and runtimes...



Motivation: USB armory firmware





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

Designed to fit in a pocket, laptops, PCs, server and networks (LAN model with PoE).

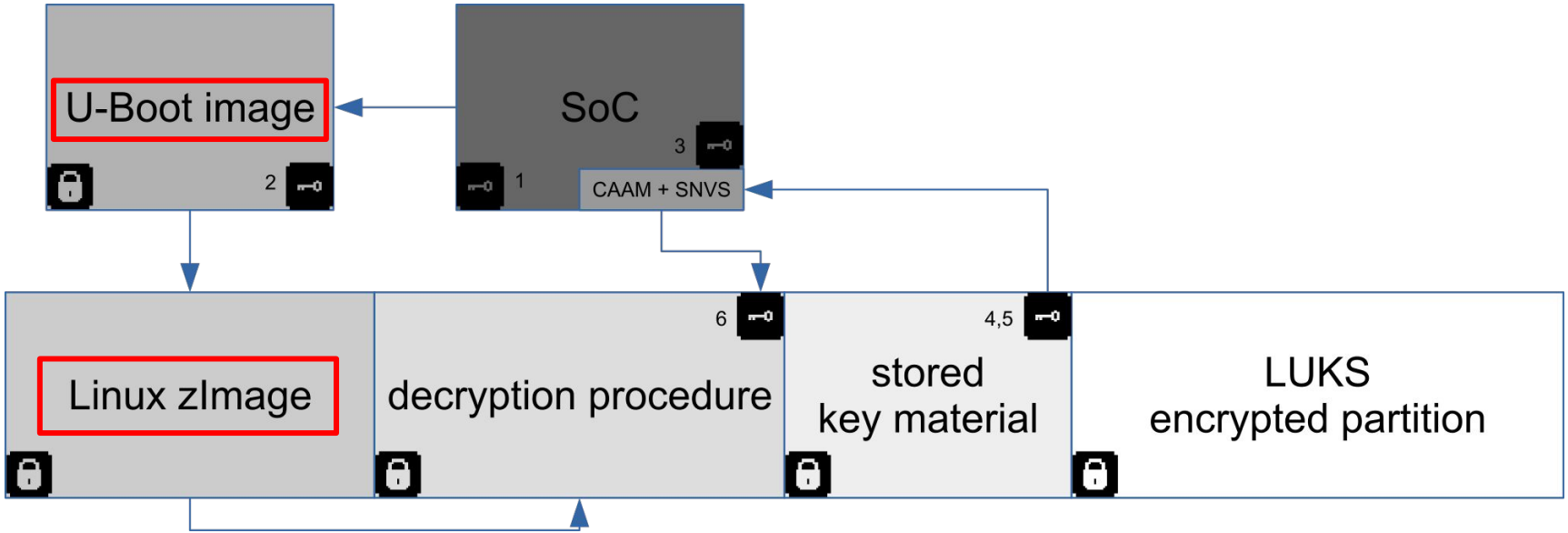
The USB armory targets the following primary applications:

- 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

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

Reducing the attack surface



Typical secure booted firmware with authentication and confidentiality on an NXP i.M6UL.

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

Motivation

In an ideal world **you should be free to choose the programming 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 data diodes, lower specs IoT and “smart” appliances.

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

Motivation

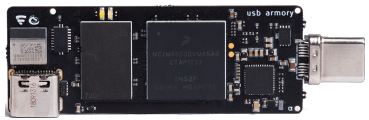
In an ideal world using **higher level languages should not entail complex dependencies**.

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

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

However in real world higher specs heavily dictate OS requirements:

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



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

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

Killing C

When security matters software and hardware optimizations matter less.

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

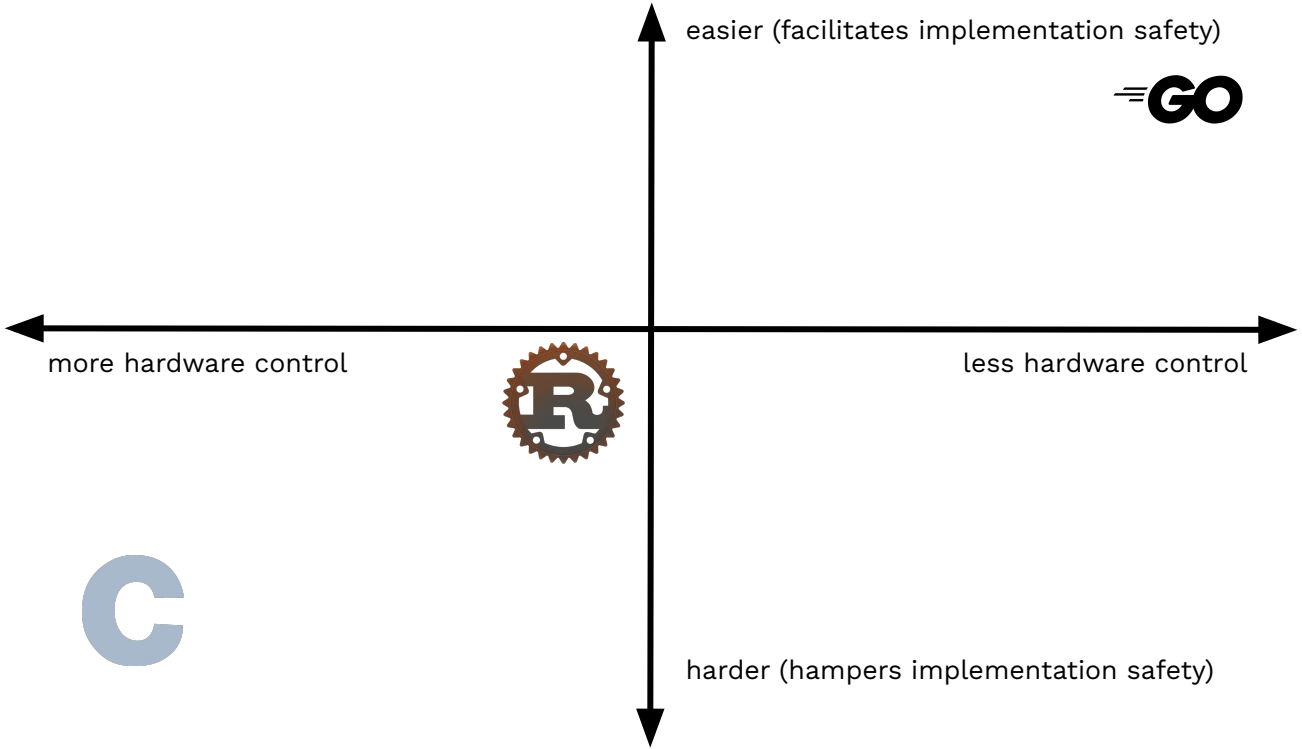
However high level programming typically entails several layers (e.g. OS, libraries) to serve runtime execution.

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

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

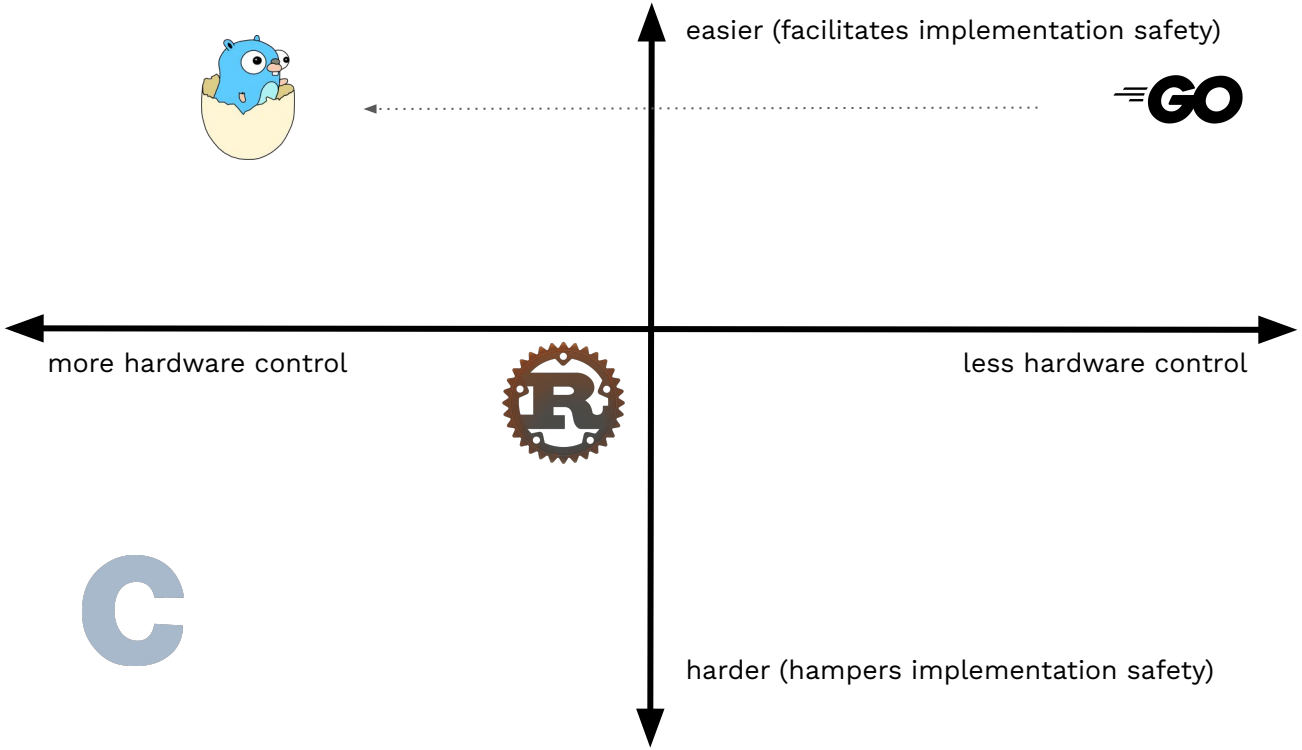
Audience mind reading trick: you are thinking “why not Rust?” … well why not *both* ?

Speed vs Safety



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

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

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

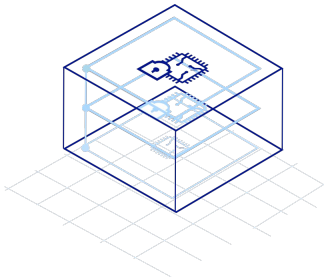
The focus is reducing the attack surface, carrying only strictly necessary code.

“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 focus started on **small embedded systems**, later focus **expanded** to TEEs, KVMs and bootloader applications.

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://research.nccgroup.com/2019/02/04/assessing-unikernel-security/

TamaGo in a nutshell

- A **minimally**[1] patched Go distribution to enable `GOOS=tamago` support, which provides freestanding execution on `GOARCH={amd64|arm|arm64|riscv64}` bare metal.

- A set of packages[2] for platform support, including hardware initialization (e.g. boot).

AMD64

`cloud_hypervisor/vm`	Cloud Hypervisor
`firecracker/microvm`	Firecracker
`qemu/microvm`	QEMU microvm, Google Compute Engine
`uefi/x64`	UEFI

ARM

`nxp/mx6ullevk`	NXP MCIMX6ULL-EVK
`raspberrypi/pi1`	Pi 1
`raspberrypi/pi2`	Pi 2
`raspberrypi/pizero`	Pi Zero
`usbarmory/mk2`	USB armory Mk II (USB+LAN)

ARM64

`nxp/imx8mpevk`	NXP 8MPLUSLPD4-EVK

RISCV64

`qemu/sifive_u`	QEMU SiFive HiFive Unleashed

[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 (inactive) - 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 (inactive) - 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 (inactive) - 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.

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`.

Egg OS (inactive) - 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 proves that it is possible to have a bare metal Go runtime, but does so 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.
- ★ ~7000 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 native runtime and stdlib support.
- ★ 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

Glue code - 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 - patches that clone original Go runtime functionality from an existing architecture to (e.g. `js`, `wasip1`) `GOOS=tamago`, either unmodified or with minimal changes:

- `plan9` memory allocation is re-used with 2 LOC changed (`brk` -> simple pointer)
- `nacl` in-memory filesystem is re-used (raw SD/MMC access implemented in `imx6`)
- `net` package is supported for networking through an external Socket function
- minimal architecture dependent code, unified `{amd64,arm,arm64,riscv64}` support

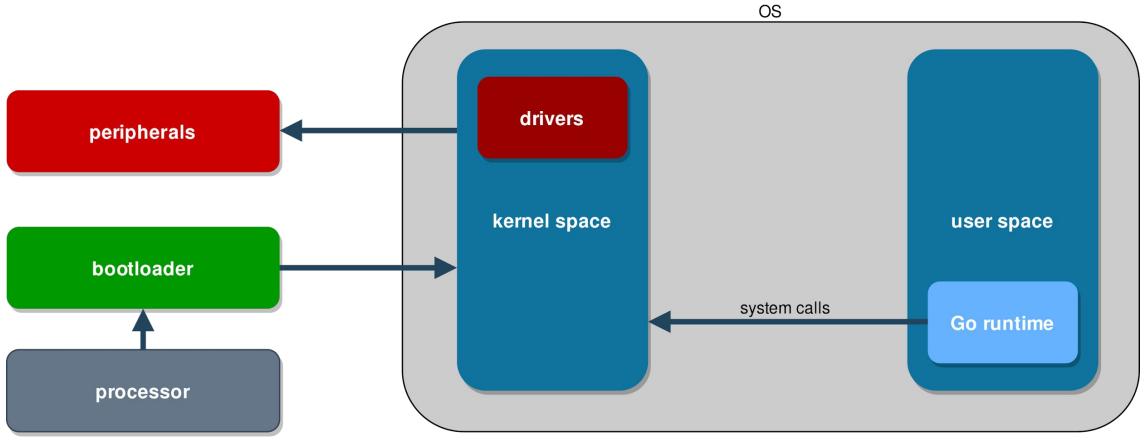
New code - basic syscall, memory layout and testing support:

```
rt0_tamago_{amd64,arm,arm64,riscv64}.s
sys_tamago_{amd64,arm,arm64,riscv64}.s
 os_tamago_{amd64,arm,arm64,riscv64}.go
testing_tamago_{amd64,arm,arm64,riscv64}.go
testing_tamago.go
     os_tamago.go
```

https://github.com/golang/go/compare/go1.25.0...usbarmory:tamago1.25.0

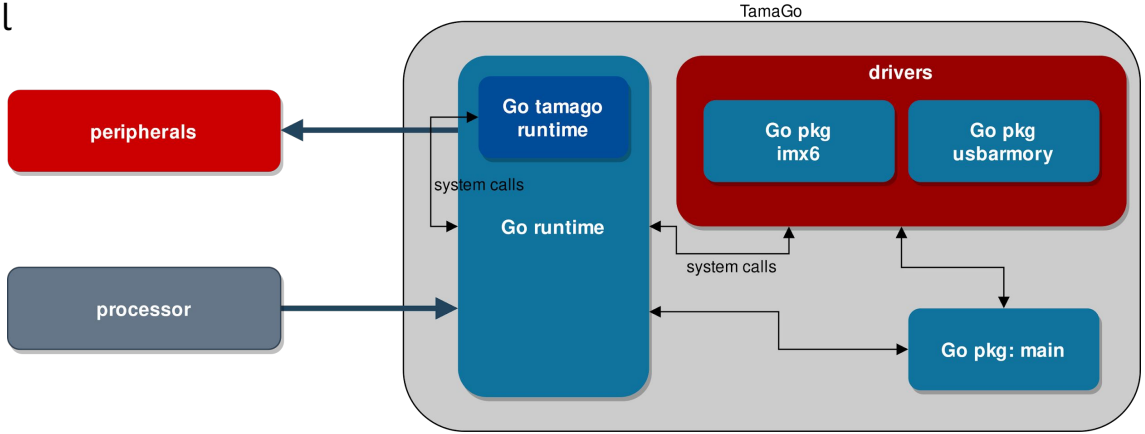
```
$ git diff go1.24.0 tamago1.24.0 --stat --stat-width "$(tput cols)" --color=always -- ":(exclude)*tamago_test.go" | sort -t '|' -n -k2 "$@"

272 files changed, 5961 insertions(+), 319 deletions(-)

 src/runtime/proc.go                                             |  14 ++++--
 ...
 src/crypto/internal/sysrand/rand_tamago.go                      |  21 ++++++++
 src/os/signal/signal_tamago.go                                  |  23 +++++++++
 src/runtime/mem_tamago.go                                       |  24 +++++++++
 .github/PULL_REQUEST_TEMPLATE                                   |  25 ----------
 src/syscall/zsyscall_tamago_amd64.go                            |  25 ++++++++++
 src/syscall/zsyscall_tamago_arm.go                              |  25 ++++++++++
 src/syscall/zsyscall_tamago_riscv64.go                          |  25 ++++++++++
 .github/workflows/build.yml                                     |  26 ++++++++++
 src/internal/goos/zgoos_tamago.go                               |  27 ++++++++++
 src/os/exec/lp_tamago.go                                        |  29 +++++++++++
 src/os/dirent_tamago.go                                         |  30 ++++++++++++
 src/runtime/os_tamago_riscv64.go                                |  30 ++++++++++++
 src/os/user/lookup_tamago.go                                    |  35 +++++++++++++
 src/net/sockopt_tamago.go                                       |  37 ++++++++++++++
 src/runtime/rt0_tamago_amd64.s                                  |  37 ++++++++++++++
 src/runtime/rt0_tamago_arm.s                                    |  37 ++++++++++++++
 src/runtime/rt0_tamago_riscv64.s                                |  38 ++++++++++++++
 src/runtime/os_tamago_amd64.go                                  |  39 +++++++++++++++
 src/runtime/note_tamago.go                                      |  40 +++++++++++++++
 src/internal/syscall/unix/net_tamago.go                         |  44 +++++++++++++++++
 src/runtime/os_tamago_arm.go                                    |  47 ++++++++++++++++++
 src/os/stat_tamago.go                                           |  51 +++++++++++++++++++
 src/time/sys_tamago.go                                          |  54 ++++++++++++++++++++
 src/testing/testing_tamago.go                                   |  55 +++++++++++++++++++++
 src/testing/testing_tamago_riscv64.s                            |  56 +++++++++++++++++++++
 src/testing/testing_tamago_amd64.s                              |  61 +++++++++++++++++++++++
 src/testing/testing_tamago_arm.s                                |  66 +++++++++++++++++++++++++
 src/testing/run_example_tamago.go                               |  76 ++++++++++++++++++++++++++++
 src/runtime/sys_tamago_riscv64.s                                | 127 +++++++++++++++++++++++++++++++++++++++++++++++
 src/runtime/lock_tamago.go                                      | 166 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/runtime/os_tamago.go                                        | 199 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/net/net_tamago.go                                           | 211 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/runtime/sys_tamago_arm.s                                    | 222 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/runtime/sys_tamago_amd64.s                                  | 251 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/syscall/fd_tamago.go                                        | 261 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/syscall/syscall_tamago.go                                   | 329 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/syscall/tables_tamago.go                                    | 494 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/syscall/fs_tamago.go                                        | 872 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/syscall/net_tamago.go                                       | 954 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

TamaGo



Traditional OS

TamaGo unikernel



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

Go low level access

```
func (hw *BEE) Init() {
	hw.mu.Lock()
	defer hw.mu.Unlock()

	hw.ctrl = hw.Base + BEE_CTRL
	hw.key = hw.Base + BEE_AES_KEY0_W0

	reg.Set(hw.ctrl, CTRL_CLK_EN)		// enable clock
	reg.Set(hw.ctrl, CTRL_SFTRST_N)		// disable reset
	reg.Clear(hw.ctrl, CTRL_BEE_ENABLE)	// disable
}

func (hw *BEE) generateKey() (err error) {
	// avoid key exposure to external RAM
	key, err := dma.NewRegion(uint(hw.key), aes.BlockSize, false)

	if err != nil {
		return
	}

	addr, buf := key.Reserve(aes.BlockSize, 0)

	if n, err := rand.Read(buf); n != aes.BlockSize || err != nil {
		return errors.New("could not set random key")
	}

	if addr != uint(hw.key) {
		return errors.New("invalid key address")
	}

	return
}
```

Example: encrypted RAM init

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.

No language changes mean that goroutines, channels, mutexes, interfaces can be used as usual.

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

https://github.com/usbarmory/tamago

type **Region**

Region represents a memory region allocated for DMA purposes.

func NewRegion

```
func NewRegion(addr uint, size int, unsafe bool) (r *Region, err error)
```

NewRegion initializes a memory region for DMA buffer allocation.

To avoid unforseen consequences the caller must ensure that allocated regions do not overlap among themselves or with the global one (see Init()).

To allow allocation of DMA buffers within Go runtime memory the unsafe flag must be set.

func (*Region) Alloc

```
func (r *Region) Alloc(buf []byte, align int) (addr uint)
```

Alloc reserves a memory region for DMA purposes, copying over a buffer and returning its allocation address, with optional alignment. The region can be freed up with Free().

If the argument is a buffer previously created with Reserve(), then its address is return without any re-allocation.

The optional alignment must be a power of 2 and word alignment is always enforced, 0 means 4 on 32-bit platforms and 8 on 64-bit ones.

type **BEE**

BEE represents the Bus Encryption Engine instance.

func (*BEE) Enable

```
func (hw *BEE) Enable(region0 uint32, region1 uint32) (err error)
```

Enable activates the BEE using the argument memory regions, each can be up to AliasRegionSize (512 MB) in size.

After activation the regions are encrypted using AES CTR. On secure booted systems the internal OTPMK is used as key, otherwise a random one is generated and assigned.

After enabling, both regions should only be accessed through their respective aliased spaces (see AliasRegion0 and AliasRegion1) and only with caching enabled (see arm.ConfigureMMU).

func (*BEE) Init

```
func (hw *BEE) Init()
```

Init initializes the BEE module.

func (*BEE) Lock

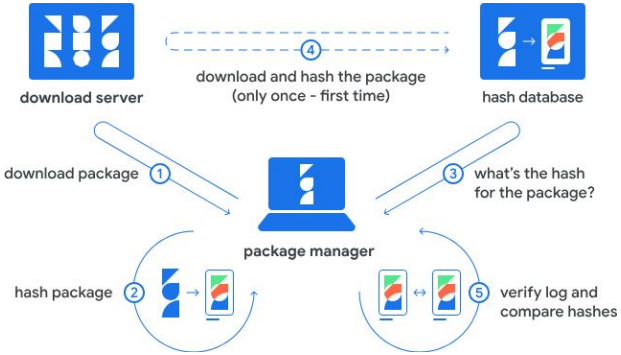
```
func (hw *BEE) Lock()
```

Lock restricts BEE registers writing.

```
import (
	"github.com/usbarmory/tamago/soc/nxp/bee"
	"github.com/usbarmory/tamago/soc/nxp/imx6ul"
)

// Encrypt 1GB of external RAM, this is the maximum extent either
// covered by the BEE or available on USB armory Mk II boards.
region0 := uint32(imx6ul.MMDC_BASE)
region1 := region0 + bee.AliasRegionSize

imx6ul.BEE.Init()
defer imx6ul.BEE.Lock()

if err := imx6ul.BEE.Enable(region0, region1); err != nil {
	log.Fatalf("could not activate BEE: %v", err)
}
```

Reducing the attack surface

Package	LOCs	Driver support
MK2	680	USB armory Mk II board support package
ARM	900	MMU, timer, exceptions, IRQ handling
BEE	130	OTF AES RAM encryption/decryption
CAAM	840	accel. AES/ECC/CMAC/SHA/TRNG, HUK derivation
DCP	450	accel. AES/SHA, HUK derivation
ENET	400	10/100-Mbps Ethernet driver, MII support
RNGB	80	True Random Number Generator
RPMB	230	Replay Protected Memory Block
RTIC	90	Run Time Integrity Checker
SNVS	180	tamper proof sensors
USB	1200	USB 2.0 in device mode
USDHC	1100	eMMC (HS200 speed) / SD (SDR104 speed)
AMD64	1000	MMU, SMP, exceptions, IRQ handling
MICROVM	150	Firecracker/QEMU microVM board support packages
UEFI	100	UEFI x64 support package
IOAPIC	60	Interrupt Controller
PCI	200	Peripheral Component Interconnect
RTC	50	Real Time Clock
CLOCK	30	KVM Clock Pairing
PVCLOCK	80	KVM PV Clock
VIRTIO	600	VirtIO over MMIO/PCI



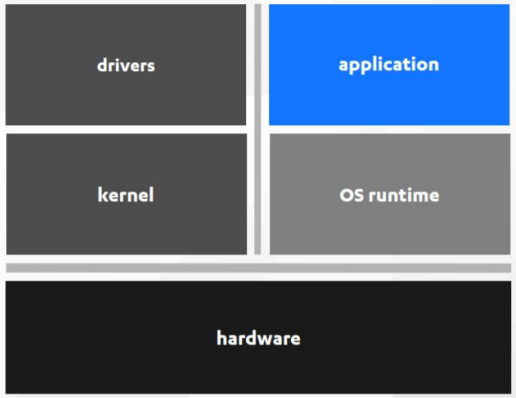

```
module github.com/usbarmory/GoTEE                    go.mod

go 1.25.0

require github.com/usbarmory/tamago v1.25.0
```

TamaGo allows creation of true unikernels, incorporating in a single binary boot code, peripheral drivers, libraries and application code with minimal dependencies and all the benefits of the full Go ecosystem, including supply chain security, build reproducibility and debugging.

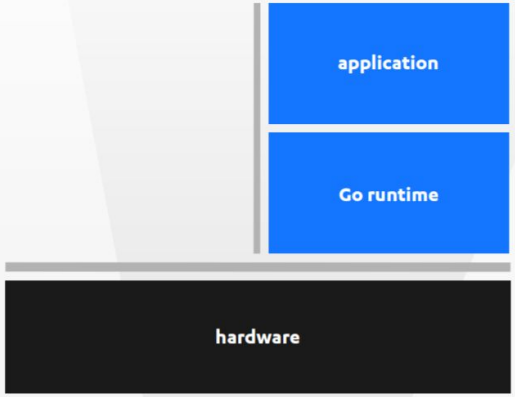
https://github.com/usbarmory/tamago/tree/master/soc/nxp

https://github.com/usbarmory/tamago/tree/master/soc/intel

Improving memory safety



Traditional OS



TamaGo unikernel

memory-unsafe programming language
memory-safe programming language

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.

programming language %

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
}
```

```
GOOS=tamago GOARM=7 GOARCH=arm
  ${TAMAGO} build -ldflags "-T 0x80010000 -R 0x1000" main.go
```

All Go ecosystem features in terms of build reproducibility, dependency management, profiling, debugging, remain intact.

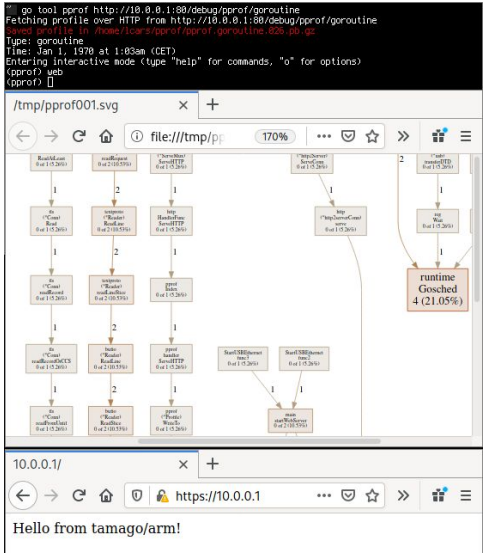
1. The application requires a single import for the board package to enable necessary initializations.
2. Go code can be written with very few limitations and the SoC package exposes driver APIs.
3. `go build` can be used as usual (reproducible builds!) with few linker flags to define entry point.
4. SoC and KVM packages support native loading (no bootloader!).

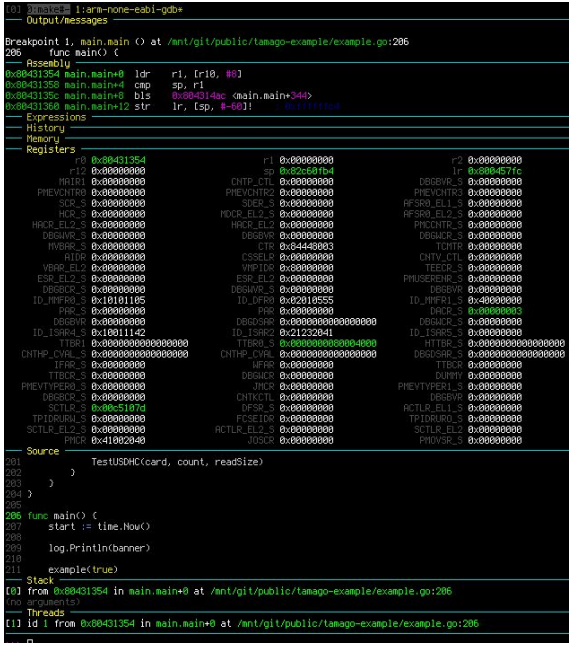
Firmware/unikernels can be compiled just as easily on Linux, Windows, macOS.

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

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

Debugging





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

The `pprof` package can be used as usual for tracing or profile guided optimization.

TamaGo runtime API

Go runtime “Rosetta Stone”

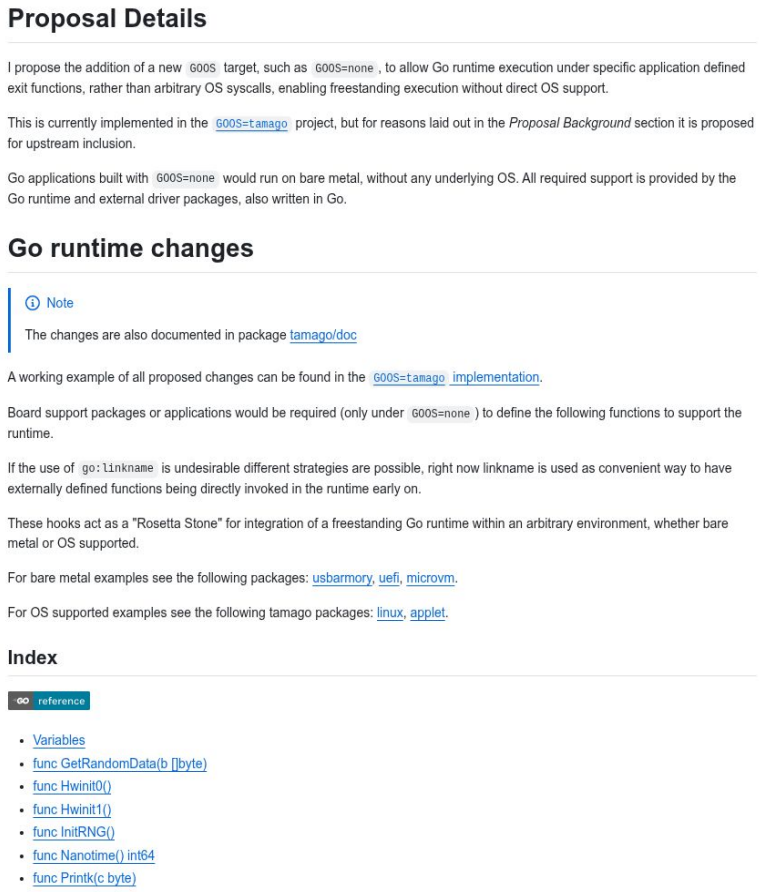
As part of the upstreaming proposal, the implementation of `GOOS=tamago` has been refined in a “Rosetta Stone” for integration of a freestanding Go runtime.

MMU initialization, exception and IRQ handling are implemented outside the Go runtime and imported as architecture packages (e.g. `amd64`, `arm`).

This means low-level APIs (e.g. TrustZone), networking, can can be regular packages.

The Go runtime modifications are board and architecture independent.

This entails that hardware ports can be achieved through external packages, whether bare metal, OS kernel space, OS user space, TEE monitor or TEE applet.



Proposal Details

I propose the addition of a new `GOOS` target, such as `GOOS=none`, to allow Go runtime execution under specific application defined exit functions, rather than arbitrary OS syscalls, enabling freestanding execution without direct OS support.

This is currently implemented in the `GOOS=tamago` project, but for reasons laid out in the *Proposal Background* section it is proposed for upstream inclusion.

Go applications built with `GOOS=none` would run on bare metal, without any underlying OS. All required support is provided by the Go runtime and external driver packages, also written in Go.

Go runtime changes

Note

The changes are also documented in package tamago/doc

A working example of all proposed changes can be found in the `GOOS=tamago` implementation.

Board support packages or applications would be required (only under `GOOS=none`) to define the following functions to support the runtime.

If the use of `go:linkname` is undesirable different strategies are possible, right now linkname is used as convenient way to have externally defined functions being directly invoked in the runtime early on.

These hooks act as a "Rosetta Stone" for integration of a freestanding Go runtime within an arbitrary environment, whether bare metal or OS supported.

For bare metal examples see the following packages: usbarmory, uefi, microvm.

For OS supported examples see the following tamago packages: linux, applet.

Index

go reference

- Variables
- func GetRandomData(b []byte)
- func Hwinit0()
- func Hwinit1()
- func InitRNG()
- func Nanotime() int64
- func Printk(c byte)

https://github.com/golang/go/issues/73608

Go runtime API

`func GetRandomData(b []byte)`

Must be linked as `runtime.GetRandomData`, generates len(b) random bytes and writes them into b.

`func InitRNG()`

Must be linked as `runtime.initRNG`, initializes random number generation.

amd64/rng.go

```
// defined in rng.s
func rdrand() uint32

// GetRandomData returns len(b) random bytes gathered from the RDRAND instruction.
//
//go:linkname GetRandomData runtime.GetRandomData
func GetRandomData(b []byte) {
        read := 0
        need := len(b)

        for read < need {
                read = rng.Fill(b, read, rdrand())
        }
}

//go:linkname initRNG runtime.initRNG
func initRNG() {}
```

https://pkg.go.dev/github.com/usbarmory/tamago/doc

Go runtime API

`func Hwinit0()`

Must be linked as `runtime.hwinit0`, takes care of lower level initialization triggered before runtime setup (pre World start).

`func Hwinit1()`

Must be linked as `runtime.hwinit1`, takes care of lower level initialization triggered early in runtime setup (post World start).

arm/init.go

```
//go:linkname Init runtime.hwinit0

func Init() {
        if int(read_cpsr()&0x1f) != SYS_MODE {
                // initialization required only when in PL1
                return
        }

        vfp_enable()
}
```

It must be defined using Go's Assembler to retain Go's commitment to backward compatibility.

Otherwise care must be taken as the lack of World start does not allow memory allocation.

microvm/microvm.go

```
//go:linkname Init runtime.hwinit1

func Init() {
        // initialize CPU
        AMD64.Init()

        // initialize I/O APIC
        IOAPIC0.Init()

        // initialize serial console
        UART0.Init()

        runtime.Exit = func(_ int32) {
                AMD64.Reset()
        }
}
```

https://pkg.go.dev/github.com/usbarmory/tamago/doc

Go runtime API

`func Nanotime() int64`

Must be linked as `runtime.nanotime1`, returns the system time in nanoseconds.

`func Printk(c byte)`

Must be linked as `runtime.printk`, handles character printing to standard output.

imx6ul/timer.go

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

arm/timer.go

```
func (cpu *CPU) GetTime() int64 {
        return int64(float64(read_cntpct())*cpu.TimerMultiplier) + cpu.TimerOffset
}
```

usbarmory/mk2/console.go

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

Must be defined using Go's Assembler to retain Go's commitment to backward compatibility.

Otherwise care must be taken as the lack of World start does not allow memory allocation.

https://pkg.go.dev/github.com/usbarmory/tamago/doc

Go runtime API

RAM layout defined by architecture and board packages

```
var ramStart uint
var ramSize uint
var ramStackOffset uint
```

imx6ul/timer.go

```
//go:linkname ramStart runtime.ramStart
var ramStart uint32 = MMDC_BASE
```

usbarmory/mk2/mem.go

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

CPU idle time management (optional)

```
var Idle func(until int64)
```

Runtime termination (optional)

```
var Exit func(code int32)
```

arm/arm.go

```
func (cpu *CPU) Init(vbar uint32) {
        runtime.Idle = func(pollUntil int64) {
                // we have nothing to do forever
                if pollUntil == math.MaxInt64 {
                        cpu.WaitInterrupt()
                }
        }
}
```

Network socket implementation (optional)

```
var SocketFunc func(ctx context.Context,
        net string,
        family, sotype int,
        laddr, raddr Addr) (interface{}, error)
```

imx-enet/runtime.go

```
func (iface *Interface) Socket(ctx context.Context ...) {
        ...
        gonet.DialContextTCP(ctx, stack, rFullAddr, proto)
}
```

tamago-example/network/imx-enet.go

```
// hook interface to Go runtime
net.SocketFunc = iface.Socket
```

https://pkg.go.dev/github.com/usbarmory/tamago/doc

Go runtime API

SMP

HW/OS threading (optional)

```
var Task func(sp mp, gp, fn unsafe.Pointer)
```

The call is invoked only when `runtime.GOMAXPROCS` > 1.

The initialization of supplemental cores (APs) is entirely outside the Go distribution, only `Task` is required as runtime hook.

```
> info
Runtime ......: go1.25.1 tamago/amd64 GOMAXPROCS=8
RAM ..........: 0x10000000-0x50000000 (1024 MiB)
Board ........: microvm
CPU ..........: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
Cores ........: 8
Frequency ....: 2.8032 GHz
VirtIO Net0 ..: 52:54:00:12:34:56

> smp 100
8 cores detected, launching 100 goroutines from CPU 0
CPU 1   8:
CPU 3  14:
CPU 5   9:
CPU 6  13:
CPU 7  24:
CPU 0  13:
CPU 2  19:
Total 100
```

amd64/smp.go

```
// Task schedules a goroutine on a previously initialized
// Application Processor.
func (cpu *CPU) Task(sp, mp, gp, fn unsafe.Pointer) {
        t := &task{
                sp: uint64(uintptr(sp)),
                mp: uint64(uintptr(mp)),
                gp: uint64(uintptr(gp)),
                pc: uint64(uintptr(fn)),
        }

        ...

        t.Write(taskAddress)

        // set last initialized CPU, signal task via NMI
        cpu.init += 1
        cpu.LAPIC.IPI(cpu.init, 0, lapic.ICR_DLV_NMI)
}

// NumCPU returns the number of logical CPUs initialized on
the platform.
func (cpu *CPU) NumCPU() (n int) {
        return 1 + len(cpu.aps)
}

// ID returns the processor identifier.
func (cpu *CPU) ID() uint64 {
        return uint64(cpu.LAPIC.ID())
}
```

https://pkg.go.dev/github.com/usbarmory/tamago/doc

Re-implementing Linux as “2nd class port”

`cpuinit`	→	`sys_mmap(ramStart, ramSize, PROT_READ|PROT_WRITE, ...)`
`func GetRandomData(b []byte)`	→	`sys_getrandom(b, len(b))`
`func Nanotime() int64`	→	`sys_clock_gettime()`
`func Printk(c byte)`	→	`sys_write(c)`
`func Task(sp, mp, gp, fn unsafe.Pointer)`	→	`sys_clone(flags, sp, mp, gp, fn)`
`runtime.Exit`	→	`sys_exit_group`

user/linux/syscall_arm64.s

```
TEXT cpuinit(SB),NOSPLIT|NOFRAME,$0
        MOVD    runtime·ramStart(SB), R0
        MOVD    runtime·ramSize(SB), R1
        MOVW    $0x3, R2        // PROT_READ   | PROT_WRITE
        MOVW    $0x22, R3       // MAP_PRIVATE | MAP_ANONYMOUS
        MOVW    $0xffffffff, R4
        MOVW    $0, R5
        MOVW    $SYS_mmap, R8
        SVC

        B       _rt0_tamago_start(SB)

// func sys_exit(code int32)
TEXT ·sys_exit(SB), $0-4
        MOVW    code+0(FP), R0
        MOVD    $SYS_exit, R8
        SVC
        RET
```

A single unified Go file allows hookup of TamaGo runtime API back to Linux.

Architecture specific Go assembly files, which are verbatim copies of relevant functions from `Sys_linux_{amd64,arm,arm64,riscv64}.s` provide Linux system call hooks.

This approach allow creation of support for `GOOS=tamago` in OS userspace for `testing` and `user/linux` packages.

https://pkg.go.dev/github.com/usbarmory/tamago/user/linux

Userspace execution

```go
package main

import (
	"fmt"
	"net"
	"os"

	_ "github.com/usbarmory/tamago/user/linux"
)

func main() {
	_, err := net.Dial("tcp", "8.8.8.8:53")
	fmt.Printf("** I can't get out!    ;-( ** %s\n", err)

	_, err = os.ReadFile("/etc/passwd")
	fmt.Printf("** I can't get out!    ;-( ** %s\n", err)
}
```

```
$ GOOS=tamago ./bin/go run main.go

** I can't get out!    ;-( ** dial tcp 8.8.8.8:53: net.SocketFunc is nil
** I don't see the FS! ;-( ** open /etc/passwd: No such file or directory
```

https://github.com/usbarmory/tamago/wiki#userspace-targets

Any imported package can implement the required runtime changes with OS supervision instead of bare metal drivers.

This allows user space execution with the benefit of system calls isolation.

The executable cannot leverage on the Go runtime to directly access OS resources.

This allows custom APIs for networking, RNG, time handling and is also used for Trusted Applet implementation (see later GoTEE coverage).

Go distribution testing for GOOS=tamago

```
$ cd tamago-go
$ GO_BUILDER_NAME=tamago GOOS=tamago GOARCH=arm64 ./bin/go tool dist test

##### Testing packages.
ok      archive/tar     1.465s
ok      archive/zip     156.222s
ok      bufio   0.683s
ok      bytes   88.814s
ok      cmp     0.106s
ok      compress/bzip2  0.299s
ok      compress/lzw    0.501s
ok      compress/zlib   3.909s
ok      container/heap  0.116s
ok      container/list  0.103s
ok      container/ring  0.099s
ok      context 0.195s
ok      crypto  0.105s
ok      crypto/aes      0.175s
ok      crypto/cipher   0.124s
ok      crypto/des      0.149s
ok      crypto/dsa      814.124s
ok      crypto/ecdh     3.932s
ok      crypto/ecdsa    79.698s
ok      crypto/ed25519  4.154s
ok      crypto/elliptic 14.773s
ok      crypto/hmac     0.218s
...
ok      cmd/internal/src        0.138s
ok      cmd/internal/test2json  0.115s
ok      cmd/link        0.216s
ok      cmd/link/internal/benchmark     0.161s
ok      cmd/link/internal/ld    0.346s
ok      cmd/link/internal/loader        0.268s
ok      cmd/nm  0.254s
ok      cmd/objdump     0.249s
ok      cmd/pack        0.224s
ok      cmd/pprof       0.321s
ok      cmd/relnote     0.164s
ok      cmd/trace       0.257s
ok      cmd/vet 0.290s
```

Demonstrates importing, execution and test compliance for the entire Go stdlib on **GOARCH={amd64,arm,arm64,riscv64}**.

Runs under Linux (natively or using qemu via binfmt_misc) bridging required external functions with Linux syscalls.

In-memory filesystem and networking support `testdata` access and network tests.

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

Releases

v1.25.2 Latest

This release introduces improved CPU idle management support functions for `amd64`, extending the pattern enabled by the previous release on single-core to multi-core CPUs:

```
runtime.Idle = func(pollUntil int64) {
    if pollUntil == 0 {
        return
    }

    cpu.SetAlarm(pollUntil)
    cpu.WaitInterrupt()
    cpu.SetAlarm(0)
}
```

Major changes for tamago package API:

- `amd64`, `arm`, `riscv64`: `(*CPU).DefaultIdleGovernor` new function to export default CPU idle time management
- `amd64.(*CPU).ClearInterrupts`: new function to signal end-of-interrupt safely under SMP (`e4346ed`)
- `amd64.(*CPU).EnableInterrupts`: function deprecated in favor of `(*CPU).ClearInterrupts` (`e4346ed`)

Major changes for tamago package internals:

- `amd64`: interrupts are now enabled on supplemental cores (APs)
- `amd64`: IRQ handling implementation improved to prevent SMP race conditions
- `amd64`: tight loops on register wait are now avoided also under SMP (`c4bd784`)
- `amd64`: fix page tables setup for correct operation under WSL and Google Cloud KVMs (#53)

Full Changelog: `v1.25.1...v1.25.2`

This release requires `GOOS=tamago` support in the Go distribution, it requires at least tamago-go1.25.2.

Assets 2

Source code (zip)	2 minutes ago
Source code (tar.gz)	2 minutes ago

https://github.com/usbarmory/tamago/releases

tamago-go1.25.2 Latest

This is a release for the Go distribution as modified by the TamaGo framework, which enables compilation and execution of unencumbered Go applications on bare metal AMD64/ARM/RISC-V processors.

This release adds `GOOS=tamago` support to Go 1.25.2.

Major changes for `GOOS=tamago GOARCH=amd64` runtime API:

- `runtime.Asleep`: new function to verify targets for `runtime.Wake` and aid race-free SMP ISR patterns (`3d633a7`)
- `testing`: new `user_linux` build tag now required with `go test` to distinguish from regular testing import (`6d5e3fe`)

Major changes for `GOOS=tamago` runtime internals:

- added support for `GOOS=arm64` (#13)
- `testing`: relocated testing runtime initialization to fix test run detection for all platforms (`631b797`)

Full Changelog: `tamago-go1.25.1...tamago-go1.25.2`
Upstream comparison: `go1.25.2...tamago-go1.25.2`

Release binaries

The release assets include the pre-compiled modified Go distribution required to compile applications using the TamaGo framework on amd64 and armv7l (such as the USB armory Mk II) Linux hosts.

Installation:

```
sudo tar -xvf tamago-go<version>.linux-<arch>.tar.gz -C /
```

You should now be able to compile the example application by setting the `TAMAGO` environment variable as follows:

```
export TAMAGO=/usr/local/tamago-go/bin/go
```

Assets 4

tamago-go1.25.2.linux-amd64.tar.gz	sha256:21957c40ac09fc…	58 MB	1 minute ago
tamago-go1.25.2.linux-armv7l.tar.gz	sha256:79b94a2adaa345…	56.6 MB	1 minute ago
Source code (zip)			4 minutes ago
Source code (tar.gz)			4 minutes ago

https://github.com/usbarmory/tamago-go/releases

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 ~450 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.

On i.MX6UL a full CAAM driver is available.

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

[1] Use cgo `-godefs`.

type **CAAM**

CAAM represents the Cryptographic Acceleration and Assurance Module instance.

func (*CAAM) DeriveKey

```
func (hw *CAAM) DeriveKey(diversifier []byte, key []byte) (err error)
```

DeriveKey derives a hardware unique key in a manner equivalent to NXP Symmetric key diversifications guidelines (AN10922 - Rev. 2.2) for AES-256 keys.

The diversifier is used as message for AES-256-CMAC authentication using a blob key encryption key (BKEK) derived from the hardware unique key (internal OTPMK, when SNVS is enabled, through Master Key Verification Blob).

WARNING: when SNVS is not enabled a default non-unique test vector is used and therefore key derivation is *unsafe*, see snvs.Available().

The unencrypted BKEK is used through DeriveKeyMemory. An output key buffer previously created with DeriveKeyMemory.Reserve() can be used to avoid external RAM exposure, when placed in iRAM, as its pointer is directly passed to the CAAM without access by the Go runtime.

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 ~80 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

```
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.

```
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 ~1200 LOC !

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

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 = 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
}
```

```
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)
}
```

```
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
}
```

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

Developed in less than 2 hours and ~400 LOC.

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

i.MX6ULL driver: Ethernet networking

```
func (ring *bufferDescriptorRing) init(rx bool, n int) uint32 {
        ring.size = n
        ring.bds = make([]*bufferDescriptor, n)

        descSize := len((&bufferDescriptor{}).Bytes())
        ptr, desc := dma.Reserve(n*descSize, bufferAlign)

        dataSize := MTU + (bufferAlign - (MTU % bufferAlign))
        addr, data := dma.Reserve(n*dataSize, bufferAlign)

        for i := 0; i < n; i++ {
                off := dataSize * i

                bd := &bufferDescriptor{
                        Addr: uint32(addr) + uint32(off),
                        data: data[off : off+dataSize],
                }

...

                off = descSize * i
                bd.desc = desc[off : off+descSize]
                copy(bd.desc, bd.Bytes())

                ring.bds[i] = bd
        }

        return uint32(ptr)
}
```

```
func (hw *ENET) Rx() (buf []byte) {
        hw.Lock()
        defer hw.Unlock()

        buf = hw.rx.pop()
        reg.Set(hw.rdar, RDAR_ACTIVE)

        return
}

func (hw *ENET) Tx(buf []byte) {
        hw.Lock()
        defer hw.Unlock()

        if len(buf) > MTU {
                return
        }

        hw.tx.push(buf)
        reg.Set(hw.tdar, TDAR_ACTIVE)
}
```

```
// enable IRQs, start interface
eth.EnableInterrupt(enet.IRQ_RXF)
eth.Start(false)

// hook interface into Go runtime
net.SocketFunc = iface.Socket
```

```
// interrupt handling
func handleEthernetInterrupt(eth *enet.ENET) {
        for buf := eth.Rx(); buf != nil; buf = eth.Rx() {
                eth.RxHandler(buf)
                eth.ClearInterrupt(enet.IRQ_RXF)
        }
}

func isr() {
        if irq := imx6ul.GIC.GetInterrupt(true); irq == eth.IRQ {
                handleEthernetInterrupt(eth)
        }
}

func StartInterruptHandler(eth *enet.ENET) {
        imx6ul.GIC.Init(true, false)
        imx6ul.GIC.EnableInterrupt(eth.IRQ, true)
        arm.ServiceInterrupts(isr)
}
```

The physical Ethernet driver for the PoE model consists of 400 LOC.

It features full IRQ handling support.

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

```
$ make qemu

tamago/arm (go1.22.1) • 5e1f6aa lcars@lambda on 2024-03-20 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)
ntp                     <host>                      # change runtime date and time w/ NTP
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  |................|

> ntp time.google.com
2024-03-20T08:02:11Z

> info
Runtime ......: go1.22.1 tamago/arm
RAM ..........: 0x80000000-0x9f600000 (502 MiB)
Board ........: UA-MKII
SoC ..........: i.MX6ULZ 1188 MHz (emulated)
```

```
$ ssh 10.0.0.1

tamago/arm (go1.22.1) • 5e1f6aa lcars@lambda on 2024-03-20 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
[142.251.215.238 2607:f8b0:400a:805::200e]


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

> info
Runtime ......: go1.22.1 tamago/arm
RAM ..........: 0x80000000-0x9f600000 (502 MiB)
Board ........: UA-MKII-γ
SoC ..........: i.MX6ULZ 900 MHz
SSM Status ...: state:0b1101 clk:false tmp:false vcc:false hac:4294967295
Boot ROM hash : 1727a0f46dbde555b583e9a138ae359389974b7be4369ffd4a252a8730f7e59b
Secure boot ..: true
Unique ID ....: FE186D5AB312430B
SDP ..........: true
Temperature ..: 48.333332
```

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

```
> tailscale $YOURKEY

tsnet --- [v1] using fake (no-op) tun device
tsnet --- [v1] using fake (no-op) OS network configurator
tsnet --- [v1] using fake (no-op) DNS configurator
tsnet --- dns: using dns.noopManager
tsnet --- link state: interfaces.State{defaultRoute= ifs={} v4=false v6=false}
tsnet --- magicsock: disco key = d:xxxxxxxxxxxxxxx
tsnet --- Creating WireGuard device...
tsnet --- Bringing WireGuard device up...
tsnet --- wg: [v2] UDP bind has been updated
tsnet --- wg: [v2] Interface state was Down, requested Up, now Up
tsnet --- Bringing router up...
tsnet --- [v1] warning: fakeRouter.Up: not implemented.
tsnet --- Clearing router settings...
tsnet --- [v1] warning: fakeRouter.Set: not implemented.
tsnet --- Starting network monitor...
tsnet --- Engine created.
tsnet --- tsnet running state path /tsnet-tamago/tailscaled.state
tsnet --- pm: migrating "_daemon" profile to new format
tsnet --- [vJSON]1{"Hostinfo":{"IPNVersion":"1.49.0-dev20230906-t7a0be7f2c-dirty","OS":"tamago","Package":"tsnet","Hostname":"tamago","GoArch":"arm","GoArchVar":"7","GoVersion":"go1.21.0"}}
tsnet --- logpolicy: using UserCacheDir, "/Tailscale"
tsnet --- [v1] netmap packet filter: (not ready yet)
tsnet --- tsnet starting with hostname "tamago", varRoot "/tsnet-tamago"
tsnet --- Start
tsnet --- generating new machine key
…
netmap: self: [0xGGm] auth=machine-authorized u=xxxxxxxxx@gmail.com [100.xxx.xx.82/32 fd7a:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx/128]
tsnet --- control: [v1] mapRoutine: netmap received: state:synchronized
tsnet --- control: [v1] sendStatus: mapRoutine-got-netmap: state:synchronized
tsnet --- active login: xxxxxxxxx@gmail.com
tsnet --- [v1] netmap packet filter: 1 filters
tsnet --- [v1] magicsock: got updated network map; 3 peers
tsnet --- [v2] netstack: registered IP 100.xxx.xx.82/32
tsnet --- [v2] netstack: registered IP fd7a:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx/128
…
tsnet --- peerapi: serving on http://100.xxx.xx.82:63151
tsnet --- peerapi: serving on http://[fd7a:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx]:63151
tsnet --- netcheck: UDP is blocked, trying ICMP
tsnet --- control: [v1] HostInfo:
{"IPNVersion":"1.49.0-dev20230906-t7a0be7f2c-dirty","BackendLogID":"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx9b2efd","OS":"tamago","Package":"tsnet","Hostname":"tamago","GoArch":"arm","GoArchVar":"7","GoVersion":"go1.21.0","Services":[{"Proto":"peerapi4","Port":63151},{"Proto":"peerapi6","Port":63151}],"Userspace":true,"UserspaceRouter":true}
tsnet --- control: [v1] PollNetMap: stream=false ep=[]

starting web server at 100.117.90.82:80
tsnet --- control: [v1] successful lite map update in 316ms
starting ssh server (SHA256:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx) at :22
```

Machine Details

Information about this machine's network. Used to debug connection issues.

Creator	[redacted]	Created	Jun 15, 2023 at 1:06 PM GMT+2
Machine name	tamago Copy	Last seen	1:06 PM GMT+2
OS hostname	tamago	Key expiry	6 months from now
OS	Tamago	CLIENT CONNECTIVITY	
Tailscale version	1.38.4-dev20230612	Varies	—
Tailscale IPv4	100.[redacted] Copy	Hairpinning	—
Tailscale IPv6	fd7a:[redacted] Copy	IPv6	—
ID	nf4[redacted]	UDP	—
Endpoints	—	UPnP	—
Relays	—	PCP	—
		NAT-PMP	—

```
$ ping 100.xxx.xx.82

PING 100.xxx.xx.82 (100.xxx.xx.82) 56(84) bytes of data.
64 bytes from 100.xxx.xx.82: icmp_seq=1 ttl=64 time=170 ms
64 bytes from 100.xxx.xx.82: icmp_seq=2 ttl=64 time=161 ms
64 bytes from 100.xxx.xx.82: icmp_seq=3 ttl=64 time=126 ms
```

https://github.com/usbarmory/tamago-example/blob/master/cmd/tailscale.go

GoKey - The bare metal Go smart card

The GoKey application implements a composite USB device implementing an OpenPGP 3.4 smartcard, a FIDO U2F token, an age plugin and a PKCS#11 over RPC server, written in pure Go (~2700[1] LOC).

It allows to implement a radically different security model for smartcards, taking advantage of Go 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

https://github.com/usbarmory/gokey

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

Demo: GoKey

https://github.com/usbarmory/gokey

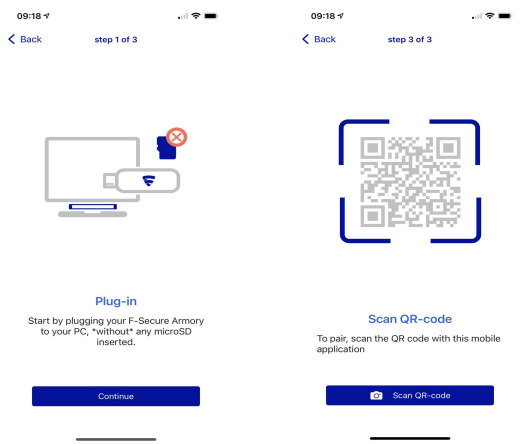
https://youtu.be/WeO2eiYSeWM

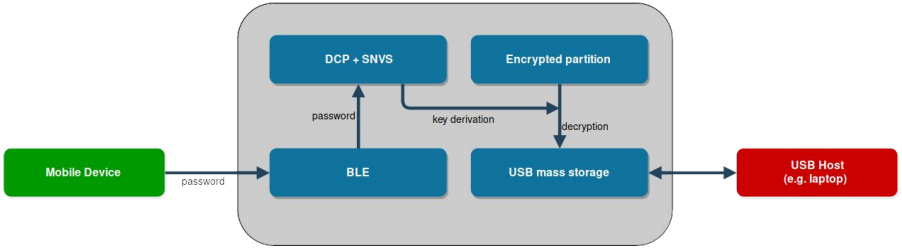
Armory Drive - Encrypted USB Mass Storage

Armory Drive implements the **easiest to use encrypted drive solution** allowing secure access to any microSD card.

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 **~3000 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

https://youtu.be/watch?v=Sv2wdVkp03A

armory-boot - USB armory boot loader

A primary signed boot loader (~700 LOC) to launch authenticated Linux kernel images on secure booted[1] USB armory boards, replacing U-Boot.

```
func boot(kernel uint, params uint, cleanup func(), region *dma.Region) (err error) {
	table := arm.SystemVectorTable()
	table.Supervisor = exec

	imx6ul.ARM.SetVectorTable(table)

	_kernel = uint32(kernel)
	_params = uint32(params)
	_mmu = (region != nil)

	cleanup()

	if region != nil {
		imx6ul.ARM.SetAttribute(
			uint32(region.Start()),
			uint32(region.End()),
			arm.TTE_EXECUTE_NEVER, 0)
	} else {
		imx6ul.ARM.FlushDataCache()
		imx6ul.ARM.DisableCache()
	}

	svc()

	return errors.New("supervisor failure")
}
```

```
func verifySignature(buf []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(buf, sig)
}

func verifyHash(buf []byte, s string) bool {
	// use hardware acceleration
	sum, _ := imx6ul.DCP.Sum256(buf) {

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

	return bytes.Equal(sum[:], hash)
}
```

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

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

type ELFImage

ELFImage represents a bootable bare-metal ELF image.

```
type ELFImage struct {
        // Region is the memory area for image loading.
        Region *dma.Region
        // ELF is a bootable bare-metal ELF image.
        ELF []byte
        // contains filtered or unexported fields
}
```

func (*ELFImage) Boot

```
func (image *ELFImage) Boot(cleanup func()) (err error)
```

Boot calls a loaded bare-metal ELF image.

func (*ELFImage) Entry

```
func (image *ELFImage) Entry() uint
```

Entry returns the image entry address.

func (*ELFImage) Load

```
func (image *ELFImage) Load() (err error)
```

Load loads a bare-metal ELF image in memory.

```
import (
          “github.com/usbarmory/armory-boot/exec”
)

image := &exec.ELFImage{
          Region: mem,
          ELF:    os.Firmware,
}

if err = image.Load(); err != nil {
          panic(fmt.Sprintf("load error, %v\n", err))
}

log.Printf("armored-witness-boot: starting kernel@%.8x\n", image.Entry())

if err = image.Boot(preLaunch); err != nil {
          panic(fmt.Sprintf("armored-witness-boot: load error, %v\n", err))
}
```

go-boot - UEFI boot manager

A TamaGo unikernel implementing a **UEFI Shell** and **OS loader** for AMD64 platforms.

It supports loading of arbitrary EFI images, Linux kernels and Windows UEFI boot manager.

It provides a clean pure Go API to UEFI Boot and Runtime services in ~2000 LOC.

It results in a 3MB EFI application which performs as fast, if not faster, than equivalent functionality Implemented in other C/Rust based efforts.

It supports **UEFI networking** through SNP, bringing full Go network I/O in your pre-boot environment.

```
initializing EFI services
initializing console (text)

go-boot • tamago/amd64 (go1.24.2) • UEFI x64

.               <path>                  # load and start EFI image
build                                   # build information
cat             <path>                  # show file contents
clear                                   # clear screen
cpuid           <leaf> <subleaf>        # show CPU capabilities
date            (time in RFC339 format)? # show/change runtime date and time
exit,quit                               # exit application
halt,shutdown                           # shutdown system
info                                    # device information
linux,l,\r      (loader entry path)?    # boot Linux kernel image
log                                     # show runtime logs
lspci                                   # list PCI devices
memmap          (e820)?                 # show UEFI memory map
mode            <mode>                  # set screen mode
peek            <hex offset> <size>     # memory display (use with caution)
poke            <hex offset> <hex value> # memory write   (use with caution)
protocol        <registry format GUID>  # locate UEFI protocol
reset           (cold|warm)?            # reset system
stack                                   # goroutine stack trace (current)
stackall                                # goroutine stack trace (all)
stat            <path>                  # show file information
uefi                                    # UEFI information
uptime                                  # show system running time
windows,win,w                           # launch Windows UEFI boot manager

> uefi
UEFI Revision ......: 2.70
Firmware Vendor ....: EDK II
Firmware Revision ..: 0x10000
Runtime Services  ..: 0xbf9ecb98
Boot Services ......: 0xbfea5720
Frame Buffer .......: 1280x800 @ 0xc0000000
Configuration Tables: 0xbf9ecc98
  ee4e5898-3914-4259-9d6e-dc7bd79403cf (0xbf8ebe98)
  05ad34ba-6f02-4214-952e-4da0398e2bb9 (0xbfea58a0)
  7739f24c-93d7-11d4-9a3a-0090273fc14d (0xbf8e7018)
  4c19049f-4137-4dd3-9c10-8b97a83ffdfa (0xbfea54e0)
  49152e77-1ada-4764-b7a2-7afefed95e8b (0xbfea6180)
  060cc026-4c0d-4dda-8f41-595fef00a502 (0xbf940018)
  eb9d2d31-2d88-11d3-9a16-0090273fc14d (0xbf93f000)
  eb9d2d30-2d88-11d3-9a16-0090273fc14d (0xbfb7e000)
  8868e871-e4f1-11d3-bc22-0080c73c8881 (0xbfb7e014)
  dcfa911d-26eb-469f-a220-38b7dc461220 (0xbebb6018)
```

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

https://github.com/usbarmory/go-boot/wiki/

```
initializing EFI services
initializing console (text)

go-boot • tamago/amd64 (go1.25.1) • UEFI x64

.                                   # `. \efi\boot\bootx64.efi`
.               <path>              # load and start EFI image
build                               # build information
cat             <path>              # show file contents
clear                               # clear screen
cpuid           <leaf> <subleaf>    # show CPU capabilities
date            (time in RFC339 format)? # show/change runtime date and time
dns             <host>              # resolve domain
exit,quit                           # exit application
gemini                              # ask gemini
halt,shutdown                       # shutdown system
help                                # this help
info                                # device information
linux,l         (loader entry path)?    # boot Linux kernel image
linux,l,\r                          # `l \loader\entries\arch.conf`
log                                 # show runtime logs
ls              (path)?             # list directory contents
lspci                               # list PCI devices
memmap          (e820)?             # show UEFI memory map
mode            <mode>              # set screen mode
net             <ip> <mac> <gw> (debug)? # start UEFI networking
peek            <hex offset> <size>     # memory display (use with caution)
poke            <hex offset> <hex value> # memory write   (use with caution)
protocol        <registry format GUID>  # locate UEFI protocol
reset           (cold|warm)?        # reset system
stack                               # goroutine stack trace (current)
stackall                            # goroutine stack trace (all)
stat            <path>              # show file information
uefi                                # UEFI information
uptime                              # show system running time
windows,win,w                       # launch Windows UEFI boot manager

> net 10.0.0.1/24 : 10.0.0.2
network initialized (10.0.0.1/24 56:6d:ab:80:04:5b)
> gemini I am talking to you from an UEFI bootloader, isn't that amazing? Shall I
boot Linux or Windows today?
That *is* amazing! It's cool to think I'm communicating with you directly from the UEFI environment.

As for your question, whether to boot Linux or Windows is entirely up to you! It depends on what you want to do today. Here's a little thought process to help you decide:

*   **What tasks do you have planned?**
    *   **Linux:** If you're planning on doing software development, system administration, using specific Linux-only tools, or prefer its command-line environment, Linux might be the better choice.
    *   **Windows:** If you're planning on gaming, using Windows-specific applications (like Microsoft Office), or prefer its user interface, Windows might be the better choice.

*   **What are you in the mood for?**
    *   Sometimes, it's just about which operating system you feel like using!

Ultimately, the decision is yours. Have fun with whichever OS you choose! Let me know if you need any more help or have any other questions.

>
```

type Services

Services represents the UEFI services instance.

```
type Services struct {
    // EFI System Table instance
    SystemTable *SystemTable

    // UEFI services
    Console *Console
    Boot    *BootServices
    Runtime *RuntimeServices
    // contains filtered or unexported fields
}
```

```
import (
       “github.com/usbarmory/go-boot/uefi”
)

UEFI := &uefi.Services{}

if err = UEFI.Init(imageHandle, systemTable); err != nil {
       panic(fmt.Sprintf("could not initialize EFI services, %v\n", err))
}

root, _ = UEFI.Root()
UEFI.Boot.LoadImage(0, root, “EFIShell.efi”)
```

func (*Services) Address

```
func (s *Services) Address() uint64
```

Address returns the EFI System Table pointer.

func (*Services) Root

```
func (s *Services) Root() (root *FS, err error)
```

Root returns an EFI Simple File System instance for the current EFI image root volume.

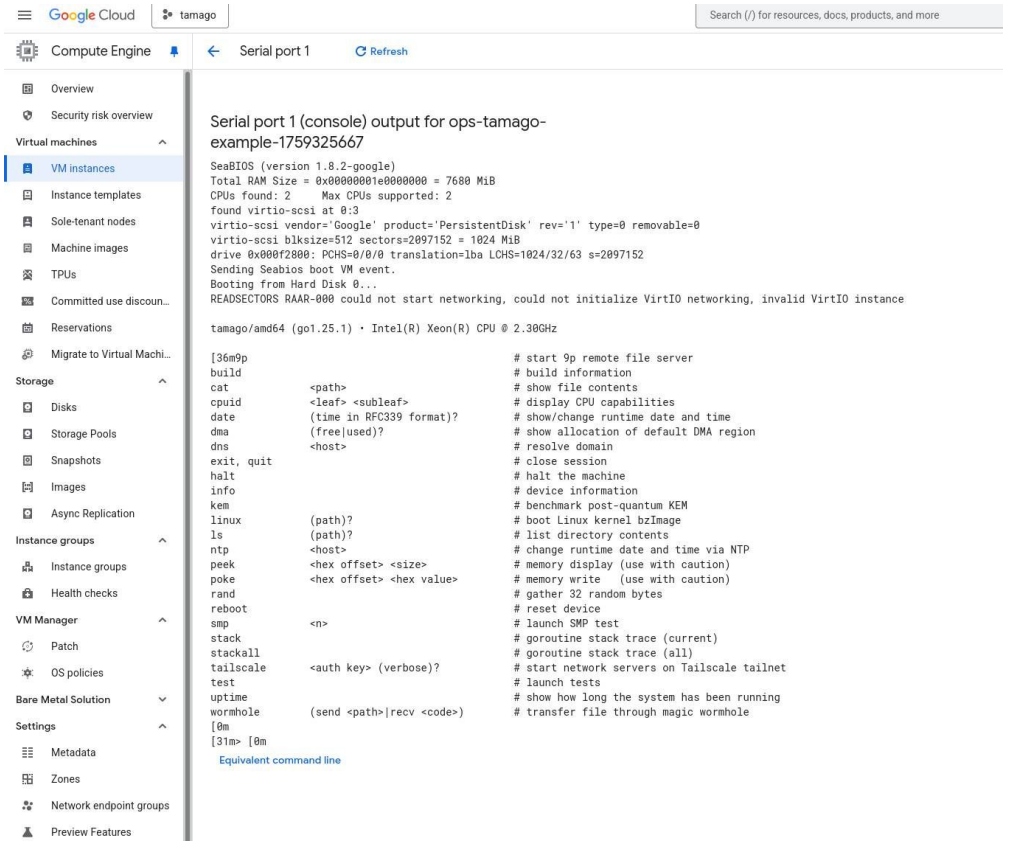
Cloud deployments

TamaGo unikernels can be deployed in cloud environments.

Unikernels can be converted to raw MBR disk images to be launched as VMs on **Google Compute Engine** or **Amazon EC2**.

Tools like ops from NanoVMs can be used to launch Go unikernels directly on arbitrary cloud providers without requiring an orchestration layer.

On compute platforms with UEFI support it is also possible to simply boot as UEFI binary.



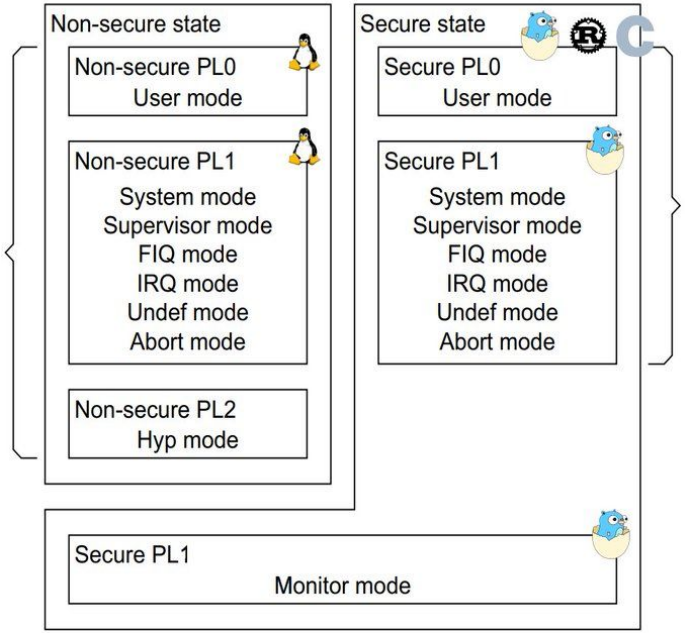
https://github.com/usbarmory/tamago-example/tree/master/tools#google-compute-engine

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).



https://github.com/usbarmory/GoTEE

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

Package monitor

```
type ExecCtx struct {
    R0  uint32
    ...
    R15 uint32 // PC

    // Memory is the executable allocated RAM
    Memory *dma.Region
    // Handler, if not nil, handles user syscalls
    Handler func(ctx *ExecCtx) error
    // Server, if not nil, serves RPC calls over syscalls
    Server *rpc.Server
}
```

```
import (
          "github.com/usbarmory/armory-boot/exec"
          "github.com/usbarmory/GoTEE/monitor"
)

image := &exec.ELFImage{
          Region: appletRegion,
          ELF:    elf,
}

if err = image.Load(); err != nil {
          return
}

if ta, err = monitor.Load(image.Entry(), image.Region, true); err != nil {
          return nil, fmt.Errorf("SM could not load applet: %v", err)
}

ta.Handler = handler
ta.Server.Register(ctl.RPC)
ta.Run()
```

func Load

```
func Load(entry uint, mem *dma.Region, secure bool) (ctx *ExecCtx, err error)
```

Load returns an execution context initialized for the argument entry point and memory region, the secure flag controls whether the context belongs to a secure partition (e.g. TrustZone Secure World) or a non-secure one (e.g. TrustZone Normal World).

func (*ExecCtx) Run

```
func (ctx *ExecCtx) Run() (err error)
```

Run starts the execution context and handles system or monitor calls. The execution yields back to the invoking Go runtime only when exceptions are caught. The function invokes the context Handler() and returns when an unhandled exception, or any other error, is raised.

```
> 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)
```

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

```
$ 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
  lockstep <% fault>                    # tandem applet example w/ fault injection

  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-example

Opportunistic soft lockstep

```
> lockstep 0.01
SM loading applet in lockstep shadow memory
SM loaded applet addr:0x10000000 entry:0x10082658 size:5002440
SM starting mode:USR sp:0x12000000 pc:0x10082658 ns:false
tamago/arm • TEE user applet
applet obtained 16 random bytes from monitor: 2c4337ed2da3d3570d7120d7d52dbda5
applet requests echo via RPC: hello
applet received echo via RPC: hello
applet will sleep for 5 seconds
applet says 1 mississippi
applet says 2 mississippi
!! injecting register fault !!
SM stopped mode:USR sp:0x11fffdd0 lr:0x101accb4 pc:0x101acbc0 ns:false err:lockstep failure
   r0:00000002  r1:11ff02a0  r2:00000000  r3:00000002
   r4:00000000  r5:00000000  r6:00000000  r7:00000001
   r8:00000001  r9:10365178 r10:10363e00 r11:00000007 cpsr:200001d3 (SVC)
  r12:584875b0  sp:11fffdd0  lr:101accb4  pc:101acbc0 spsr:200001d0 (USR)
shadow context:
   r0:00000003  r1:11ff02a0  r2:00000000  r3:00000002
   r4:00000000  r5:00000000  r6:00000000  r7:00000001
   r8:00000001  r9:10365178 r10:10363e00 r11:00000007 cpsr:200001d3 (SVC)
  r12:584875b0  sp:11fffdd0  lr:101accb4  pc:101acbc0 spsr:200001d0 (USR)
stack trace:
  github.com/usbarmory/GoTEE/syscall/syscall_arm.s:37
  github.com/usbarmory/GoTEE/applet/nanotime_syscall.go:21

>
```

GoTEE supports lockstep execution of any deterministic execution context, such a Trusted Applet.

The tandem progression of a Trusted Applet executable allows detection of injected or naturally occurred faults.

To achieve this a shadow execution context is paired to the primary one for comparison at each monitor call.

All it takes is the definition of a function that re-configures virtual memory addressing so that the same applet can be allocated in two separate physical memory areas.

```
ta, _ := monitor.Load(taEntry, taMemory, true)
ta.Shadow = ta.Clone()

flags := arm.MemoryRegion|arm.TTE_AP_011<<10

ta.MMU = func() {
        imx6ul.ConfigureMMU(start, end, primaryStart, flags)
}

ta.Shadow.MMU = func() {
        imx6ul.ConfigureMMU(start, end, shadowStart, flags)
}
```

https://github.com/usbarmory/GoTEE/wiki/Examples

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

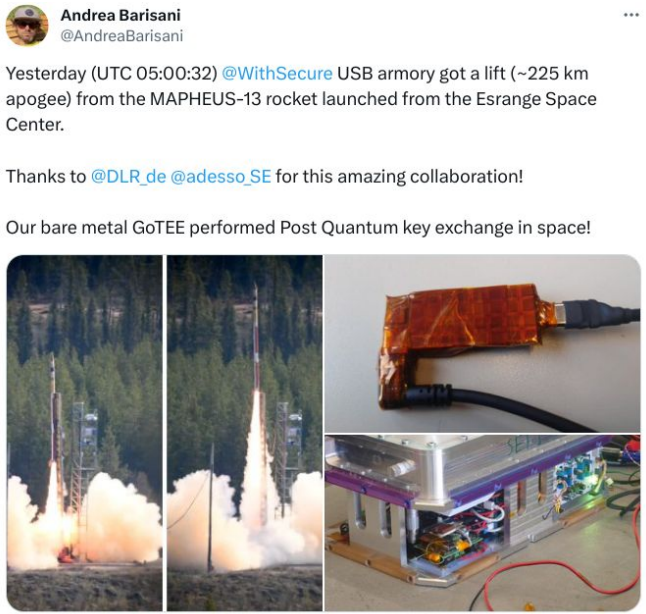
GoTEE in the wild: space!

On May 22nd 2023 (UTC 05:00:32) the USB armory Mk II got a lift to space!

On February 27th 2024 (UTC 07:27:00) the same unit went back to space.

GoTEE supervised a TamaGo based unikernel (acting as Trusted Applet) and a full Linux instance isolated in NonSecure World to test Post Quantum Key exchanges.

For this occasion TamaGo and GoTEE have been updated with full watchdog and interrupt support.



The payload remained operation for the entire flight duration performing exactly 400 PQC key exchanges. As far as we know this is the first time bare metal Go executed in space.

https://www.withsecure.com/en/whats-new/pressroom/usb-armory-post-quantum-encryption-put-to-test-in-space

GoTEE in the wild: Armored Witness

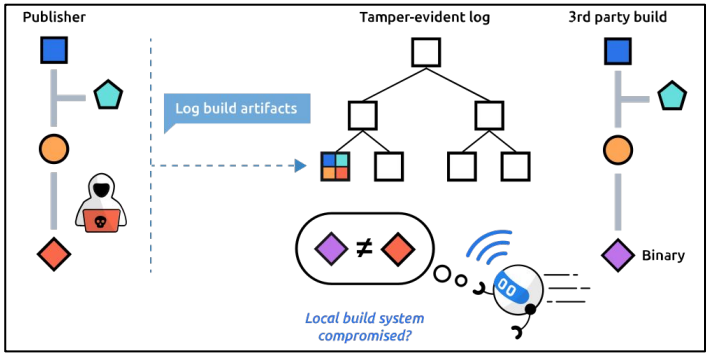
The applet observes public transparency logs verifying that they're operating in an append-only fashion, and counter-signing those checkpoints which it has determined are consistent with all previous checkpoints its seen from the same log.

The counter-signed checkpoints are sent to a **distributor**, which then collates counter-signatures for a given checkpoint from one or more Armored Witness devices, and serves them via a public API.

The benefit of this system comes through **removing trust from log operators** to behave honestly, and **placing some of that trust in the witnesses**.

Splitting the trust across multiple parties in this way means that **a larger number of parties must collude to hide malfeasance**, and as other witness implementations/networks start to appear, the number of parties required to collude increases correspondingly.

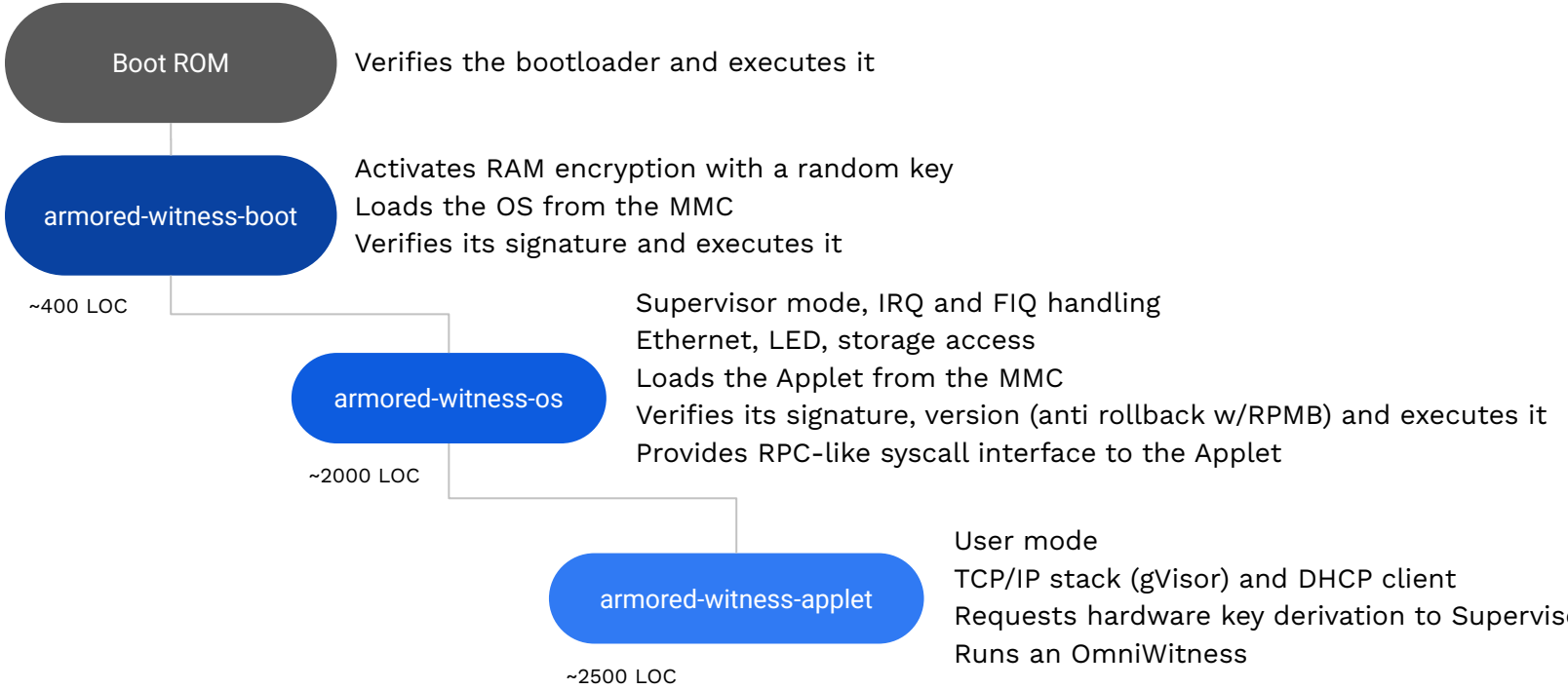
However, **we can minimise the amount of trust required** to be placed in the Armored Witness by having it be **as transparent as possible** too.



https://github.com/transparency-dev/armored-witness

https://blog.transparency.dev/can-i-get-a-witness-network

Armored Witness architecture



All firmware verification past the Boot ROM stage verify, through FT proof bundles, the presence of one or more trusted signatures (multi party signing) on the release manifest. The release manifest includes the binary hash, log checkpoint, the index of the manifest in the log and its corresponding inclusion proof.

The OS and Applet must be published on the log to be usable on the device.

https://github.com/transparency-dev/armored-witness

Minimising trust

All firmware is open source, written in TamaGo, and is build-reproducible by anyone. All firmware is logged to a Firmware Transparency log at build and release time.

The provision tool will only use firmware artefacts discovered in the FT log in order to program devices. The on-device self-update process requires that updated firmware is hosted in the FT log.

The verify tool can be used by *custodians* to inspect the device, extract the firmware components from it, and verify that they are present in the FT log.

The verify_build command continuously monitors the contents of the FT log, and tests that every logged firmware is indeed reproducibly built.

Google is able to quickly become aware of misuse of their signing identities to release unauthorised firmware updates.

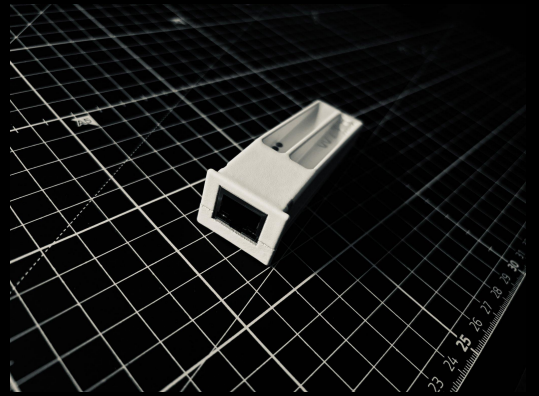

```
{
  "schema_version": 0,
  "component": "TRUSTED_APPLET",
  "git": {
    "tag_name": "0.3.1709910063-incompatible",
    "commit_fingerprint": "9651fc25839d9937acc041057cf3906f26fc1ae5"
  },
  "build": {
    "tamago_version": "1.22.0",
    "envs": [
      "FT_LOG_URL=https://api.transparency.dev/armored-witness-firmware/ci/log/2",
      "FT_BIN_URL=https://api.transparency.dev/armored-witness-firmware/ci/artefacts/2",
      "LOG_ORIGIN=transparency.dev/armored-witness/firmware_transparency/ci/2",
      "LOG_PUBLIC_KEY=transparency.dev-aw-ftlog-ci-2+f77c6276+AZXqiaARpwF4MoNOxx46kuiIRjrML0PDTm+c7BLaAMt6",
      "APPLET_PUBLIC_KEY=transparency.dev-aw-applet-ci+3ff32e2c+AV1fgxtByjXuPjPfi0/7qTbEBlPGGCyxqr6ZlppoLOz3",
      "OS_PUBLIC_KEY1=transparency.dev-aw-os1-ci+7a0eaef3+AcsqvmrcKIbs21H2Bm2fWb6oFWn/9MmLGNc6NLJty2eQ",
      "OS_PUBLIC_KEY2=transparency.dev-aw-os2-ci+af8e4114+AbBJk5MgxRB+68KhGojhUdSt1ts5GAdRIT1Eq9zEkgQh",
      "REST_DISTRIBUTOR_BASE_URL=https://api.transparency.dev/ci",
      "BEE=1",
      "DEBUG=1",
      "SRK_HASH=b8ba457320663bf006accd3c57e06720e63b21ce5351cb91b4650690bb08d85a"
    ]
  },
  "output": {
    "firmware_digest_sha256": "lLPLT5TO2+Ln71cByKhVvNFyAL47IzOOSGoXNKVSCvU="
  }
}

-- transparency.dev-aw-applet-ci
P/MuLOfW8473+PNMa58SZA2/rw1aEaIaLTw/aNfdawSiyFEcDjGksYqCTFMnHHGAhhbfnITkkktL1...
```


Anyone can verify that each firmware can be rebuilt consistently with what is running, logged and its source.

The entire Software Bill of Materials (SBOM) can be managed with Go ecosystem tools (e.g. `go.mod` + `go.sum`, `go mod graph`).

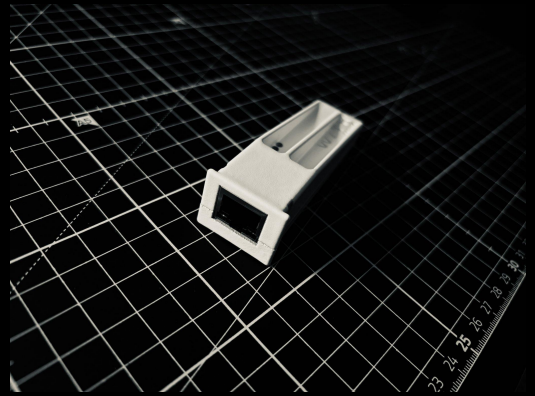
https://github.com/transparency-dev/armored-witness/blob/main/docs/transparency.md

```
boot: tamago/arm • i.MX6UL
boot: starting kernel@1007dee8
os: tamago/arm • TEE security monitor (Secure World system/monitor)
os: loading applet from MMC storage
os: SM applet verification pub:RWQiFth4tAgsVQT5caaZGJGgUzZFnwCdeVHe5XpobGWc9XzCJmjJ56tO
os: SM applet loaded addr:0x20000000 entry:0x20081700 size:15323136
os: SM applet started mode:USR sp:0x30000000 pc:0x20081700 ns:false
ta: tamago/arm • TEE user applet
ta: SM starting network
ta: SNVS - Deriving hardware key
ta: Opening storage - CardInfo: {BlockSize:512 Blocks:8388608}
os: SM registering applet event handler g:0x21803900 p:0x21826000
ta: MAC:26:76:04:d4:4d:db IP:10.0.0.1/24 GW:10.0.0.0/24 DNS:8.8.8.8:53
ta: Starting witness...
ta: TA starting ssh server (SHA256:IH0WyXV66dvixxOPDhtAalAvWg+sQC…) at 10.0.0.1:22
ta: Feeder "go.sum database tree" goroutine started
ta: Feeder "Armory Drive Prod 2" goroutine started
ta: Feeder "rekor.sigstore.dev - 3904496407287907110" goroutine started
ta: Feeder "rekor.sigstore.dev - 2605736670972794746" goroutine started
ta: Feeder "developers.google.com/android/binary_transparency/0" goroutine started
ta: "sum.golang.org: go.sum database tree" grew - @0:  → @19895786: 3c9b8f49f56cb…
ta: "armory-drive-log: Armory Drive Prod 2" grew - @0:  → @2: 02a14ca4a7313d868a4…
ta: "rekor.sigstore.dev: 2605736670972794746" grew - @0:  → @36541297: 0d491271b9…
ta: "pixel_transparency_log: …/binary_transparency/0" grew - @0:  → @235: f98458…
ta: "rekor.sigstore.dev: 3904496407287907110" grew - @0:  → @4163431: 4d006aa46ef…
ta: "lvfs: lvfs" grew - @0:  → @12749: 6ac429c550b8b28b7c65b6e61c99c9c76303d14012…
ta: No checkpoint
```



```
$ ssh 10.0.0.1

TA tamago/arm (go1.22.1) • TEE user applet (User Mode)

date            (time in RFC339 format)?              # show/change runtime date and time
dns             <fqdn>                                # resolve domain (requires routing)
exit, quit                                            # close session
hab             <hex SRK hash>                        # secure boot activation (*irreversible*)
help                                                  # this help
led             (white|blue|yellow|green) (on|off)    # LED control
mmc             <hex offset> <size>                   # MMC card read
reboot                                                # reset device
stack                                                 # stack trace of current goroutine
stackall                                              # stack trace of all goroutines
status                                                # status information

> status
------------------------------------------------------- Trusted Applet ----
Runtime .............: go1.22.1 tamago/arm
------------------------------------------------------------ Trusted OS ----
Serial number .......: 3bee6cda358f0c33
Secure Boot .........: false
Revision ............: 70ffeda
Build ...............: lcars@lambda on 2024-03-21 08:50:01
Version .............: 1696495801 (2024-03-21 08:50:01 +0000 UTC)
Runtime .............: go1.22.1 tamago/arm
Link ................: true
Witness/Identity ....: DEV:ArmoredWitness-still-tree+e1f17ea8+AZvDSL1C0…
Witness/IP ..........: 10.0.0.1
```

```
$ ssh armory
tamago/arm (go1.22.1) • 4112c8d lcars@lambda on 2024-03-08 12:06:34 • i.MX6UL 528 MHz
ntp             <host>                                  # change runtime date and time via NTP
tailscale       <auth key> (verbose)?                   # start network servers on Tailscale tailnet
witness                                                 # start/inspect transparency.dev omniwitness
wormhole        (send <path>|recv <code>)               # transfer file through magic wormhole
> ntp time.google.com
2024-03-08T12:06:58Z
> witness
starting omniwitness on :8080 (tamago-example-ephemeral-witness+599e290c+Afszq5oPlHBvi/cyjEkGGGHS+r96hhedJK4C71BdqPlG)

I0308 12:07:04.123431 Feeder "lvfs" goroutine started
I0308 12:07:04.135195 Feeder "go.sum database tree" goroutine started
I0308 12:07:04.142646 Feeder "Armory Drive Prod 2" goroutine started
I0308 12:07:04.148453 Feeder "rekor.sigstore.dev - 3904496407287907110" goroutine started
I0308 12:07:04.157244 Feeder "developers.google.com/android/binary_transparency/0" goroutine started

> witness
Armory Drive Prod 2 2 AqFMpKcxPYaKTmihsFbQvb758iSzJvvJBX5thVJ7r/k=                                  // log checkpoint root hash
— armory-drive-log FlQbj/vNC0bZUS8GUCMAw4A3GOMRU+ZkhVTPmGXTuFST75f2v92/02ilu6eukoSbFTlzQmCNDT6wor5VB/X5z91bMAA=  // log signature
— tamago-example-ephemeral-witness 6SKM5Gj/6mUAAAAALhy5ei/8YVJBYTxZWSP8p2ST+868EDQkXUtz40JRSwtzlYOqQ4W6o4g+vneP9rNkiEAN/gQGXGQd9Jz2HQ5JDw==

rekor.sigstore.dev - 3904496407287907110 4163431 TQBqpG78tgfdUdkAsSE3VMUMySUcNAXGwlYdnWovMjk=
— rekor.sigstore.dev wNI9ajBGAiEA8NoGSE0tPoD0tk5kNKQdM4Sxv4L5551vMsbvavFkD1ICIQDW1QsPAS1jGQAqjwOqpWft0m+Iw5P/Kd2ImoUdMgez4g==
— tamago-example-ephemeral-witness 6SKM5Gj/6mUAAAAApV8+I7roAQgSLMOXxxIJ7jZ32mATtz1mlZfiLyqogUjnij59h478pruCKgJbK5sDzQL9JFeVyGO3G4te5bkHDg==

developers.google.com/android/binary_transparency/0 324 vSoSuFDnfUfaKNJne2AtZjQD1CPORv+BLqnSXJE4phE=
— pixel_transparency_log csh42zBFAiEAk7GYrVxnXVZW9UDGMk3vdEOwbHBi2EMUZ0XQ0zz7e3MCIDHOpuLOuvx3O5pyHWl32jJdlldkClzcS/VUVTbMogsE
— tamago-example-ephemeral-witness 6SKM5Gj/6mUAAAAAodGRDGrZ5UQwnanHsDWhUqpzXY8+6RqTvQczrdMtHr1pJP8nE6G6NvKzWJhcnFcOBkCDITSYWbSAqAaA6EEHDg==
go.sum database tree 23470203 fA/vQxRCMYYyCzwGMoSaYipBC0tT0NEv4IpQgL8yCr8=
— sum.golang.org Az3grl+gARGNedLjWIGz62fX334EBbUFiPdkuoEZIvQgNKWccGOL9GXKzNBUNgK6MAUx4b5/f3ogAfftTCRVO5W54QA=
— tamago-example-ephemeral-witness 6SKM5Gn/6mUAAAAADpA/9FTe0cuJtPoztsjc+ksmieeezmxhzF+y1+8TMgCbMjBEumBuraML92PLu1OTNriB6ocf/pPkyWbzszIAA==

Lvfs 14770 TVcAhxR0NKIkoKwGAsVSouDSiQ1o2lA9LT1ZPZYMnSY=
— lvfs eQjRQuZh1vVt9r1JRlxYMhTbZz9xtL/0tiRdrOnkakyycwJkKLVooAA2PGUsWi0wuwFXdxTgbll5VijHQX3DdTos2Q4=
— tamago-example-ephemeral-witness 6SKM5Gn/6mUAAAAAZ/uaA9ptWcQKdrt3esieSL9XjAQQmKFVolEomVTKD+gUYXxVQY6GHKOKbuQdeIAGEyBIgGuQF2xJyJih7NvoBA==
```



```
$ sudo ./provision --template=ci
…
Fetching TRUSTED_OS bin from "f2a54c9ff38f27b92afe9f0db6794d528e34cf508e60f90d7399f87f6f8143b1"
Fetching TRUSTED_APPLET bin from "a2012b90c44e8e4e48aa04f41f005813342e9f461cdf9f705e72fa1f4dd0f870"
Fetching BOOTLOADER bin from "1285e82d785723a054e2f43d19af3edf8d7f1092ef5dee58c5010753ec6f039b"
Fetching RECOVERY bin from "8271e2a8ccefb6c4df48889fcbb35343511501e3bcd527317d9e63e2ac7349e3"
🙋 OPERATOR: please ensure boot switch is set to USB, and then connect unprovisioned device
Recovery firmware is 1924096 bytes + 16384 bytes HAB signature
Waiting for device to be detected...
found device 15a2:007d Freescale SemiConductor Inc  SE Blank 6UL
Attempting to SDP boot device /dev/hidraw0
Loading DCD at 0x00910000 (976 bytes)
Loading imx to 0x8000f400 (1940480 bytes)
Serial download on /dev/hidraw0 complete
Witness device booting recovery image
Waiting for block device to appear
Waiting for block device to settle...
✅ Detected device "/dev/hidraw0"
✅ Detected blockdevice /dev/disk/by-id/usb-F-Secure_USB_armory_Mk_II_720A9DEAD439211E-0:0
Bootloader firmware is 2976768 bytes + 16384 bytes HAB signature
Flashing images...
  ✅ os @ 0x5000
  ✅ bootloader @ 0x2
  ✅ boot config @ 0x4fb0
  ✅ applet @ 0x200000
✅ Flashed images
🙋 OPERATOR: please change boot switch to MMC, and then reboot device 🙏
Waiting for device to boot...
Waiting for armored witness device to be detected...
✅ Detected device "/dev/hidraw0"
✅ Witness serial number 720A9DEAD439211E found
✅ Witness serial number 720A9DEAD439211E is not HAB fused
🙋 OPERATOR: please reboot device 🙏
Waiting for device to boot...
✅ Witness ID DEV:ArmoredWitness-nameless-rain+192be1c1+AY5ob1kU0v3w4obdEBXVC0ygvNhco8wDMkOMIk1YGZdv provisioned
✅ Device provisioned!
```

```
$ docker run armored-witness-build-verifier continuous \
  --log_origin=transparency.dev/armored-witness/firmware_transparency/ci/3  \
  --log_url=https://api.transparency.dev/armored-witness-firmware/ci/log/3/ \
  --log_pubkey=transparency.dev-aw-ftlog-ci-3+3f689522+Aa1Eifq6rRC8qiK+bya07yV1fXyP156pEMsX7CFBC6gg

No previous checkpoint, starting at 0
Running Monitor.From (0, 5]
Downloading and installing tamago 1.22.0
Installed tamago 1.22.0 at /usr/local/tamago-go/1.22.0
Leaf index 0: ✅ reproduced build TRUSTED_APPLET@0.3.1710338359-incompatible (fc52bc11b0d543de847eed44b285acfe7eabed03) => 4f36a18f64014ee9f7c56568c8aeec3bc07d9b05849d3b96c9ff3ad3e8aa721f
Leaf index 1: ✅ reproduced build TRUSTED_OS@0.3.1710338980-incompatible (90eb1cb61f0981fe1bdcf5b23b00ae8bf44bbbbe) => 7f562e45ac78487679d422ec6a7adffe9e0bbbc8d4d44d3622a4978bf4c68075
Leaf index 2: ✅ reproduced build TRUSTED_APPLET@0.3.1710339913-incompatible (7a5de51228e9299b7185440c73b54c86baefb117) => f226de72b0c56c7f8443fb90112c9d5169165e038f7eb9e7bbe2a21951ce3097
Leaf index 3: ✅ reproduced build RECOVERY@0.3.1710340256-incompatible (850baf54809bd29548d6f817933240043400a4e1) => 8271e2a8ccefb6c4df48889fcbb35343511501e3bcd527317d9e63e2ac7349e3
Leaf index 4: ✅ reproduced build BOOTLOADER@0.0.1710340266-incompatible (86aefd7e97bc3a96814e52b8c01bdd67e26cc837) => 1285e82d785723a054e2f43d19af3edf8d7f1092ef5dee58c5010753ec6f039b
No known backlog, switching mode to poll log for new checkpoints. Current size: 5
```

```
$ sudo ./verify --template=ci
…
Fetching RECOVERY bin from "8271e2a8ccefb6c4df48889fcbb35343511501e3bcd527317d9e63e2ac7349e3"
--------------------------------------------------------------------------------------------
🙏 Operator, please ensure boot switch is set to USB, and then connect device 🙏
--------------------------------------------------------------------------------------------
Recovery firmware is 1924096 bytes + 16384 bytes HAB signature
Waiting for device to be detected...
found device 15a2:007d Freescale SemiConductor Inc  SE Blank 6UL
Attempting to SDP boot device /dev/hidraw0
Loading DCD at 0x00910000 (976 bytes)
Loading imx to 0x8000f400 (1940480 bytes)
Sending jump address to 0x8000f400
Serial download on /dev/hidraw0 complete
Witness device booting recovery image
Waiting for block device to appear
Waiting for block device to settle...
✅ Detected device "/dev/hidraw0"
✅ Detected blockdevice /dev/disk/by-id/usb-F-Secure_USB_armory_Mk_II_720A9DEAD4390E1F-0:0
Found config at block 0x4fb0
Reading 0x2d6c00 bytes of firmware from MMC byte offset 0x400
Found config at block 0x5000
Reading 0xdbd965 bytes of firmware from MMC byte offset 0xa0a000
Found config at block 0x200000
Reading 0x102a51a bytes of firmware from MMC byte offset 0x4000a000
  ✅ Bootloader: proof bundle is self-consistent
  ✅ Bootloader: proof bundle checkpoint(@10) is consistent with current view of log(@10)
  ✅ TrustedOS: proof bundle is self-consistent
  ✅ TrustedOS: proof bundle checkpoint(@10) is consistent with current view of log(@10)
  ✅ TrustedApplet: proof bundle is self-consistent
  ✅ TrustedApplet: proof bundle checkpoint(@10) is consistent with current view of log(@10)
✅ Device verified OK!
--------------------------------------------------------------------------------------------
🙏 Operator, please ensure boot switch is set to MMC, and then reboot device 🙏
--------------------------------------------------------------------------------------------
```

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/go1.32.0/src/crypto/ecdsa/ecdsa_test.go#L76

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 or BIOS jumps directly to Go runtime.

USB armory
Repository: https://github.com/usbarmory/usbarmory
Documentation: https://github.com/usbarmory/usbarmory/wiki
HAB/OTP tool: https://github.com/usbarmory/crucible

TamaGo
Repository: https://github.com/usbarmory/tamago
Documentation: https://github.com/usbarmory/tamago/wiki
API: https://pkg.go.dev/github.com/usbarmory/tamago
Example: https://github.com/usbarmory/tamago-example

GoTEE
Repository: https://github.com/usbarmory/GoTEE
Documentation: https://github.com/usbarmory/GoTEE/wiki
Example: https://github.com/usbarmory/GoTEE-example

Armored Witness
Repository: https://github.com/transparency-dev/armored-witness
Bootloader: https://github.com/transparency-dev/armored-witness-boot
OS: https://github.com/transparency-dev/armored-witness-os
Applet: https://github.com/transparency-dev/armored-witness-applet

go-boot
Repository: https://github.com/usbarmory/go-boot
HCL: https://github.com/usbarmory/go-boot/wiki

TamaGo

Bare metal Go

Secure embedded unikernels
with drastically reduced attack surface

Andrea Barisani

https://andrea.bio

andrea@inversepath.com

andrea.barisani@reversec.com