ArmoredWitness

Putting verifiable transparency in your verifiable transparency

Al Cutter
Martin Hutchinson
9th October 2024

Google

Logs

A log provides a verifiable *globally consistent*, *append-only* list of data.

Witnesses

A witness:

- cosigns consistent checkpoints with what it last signed

Quorum of witnesses:

- ensures no split view

A witness only verifies hashes!

Omniwitness

TrustFabric witness + policy

- Go SumDB
- WithSecure Armory Drive
- Sigstore
- Android BT
- LVFS
- Armored Witness !!
- Bastion support

Armored Witness

Omniwitness + armor

- open source hardware & software
- reproducible builds installed from a log
- full boot-chain verification from mask ROM to running software

Reputation

Armored Witness = Our Reputation

But... why?!

A fleet of devices provides:

- Resilience
- Geographic coverage
- Awareness & buy-in

Transparency provides:

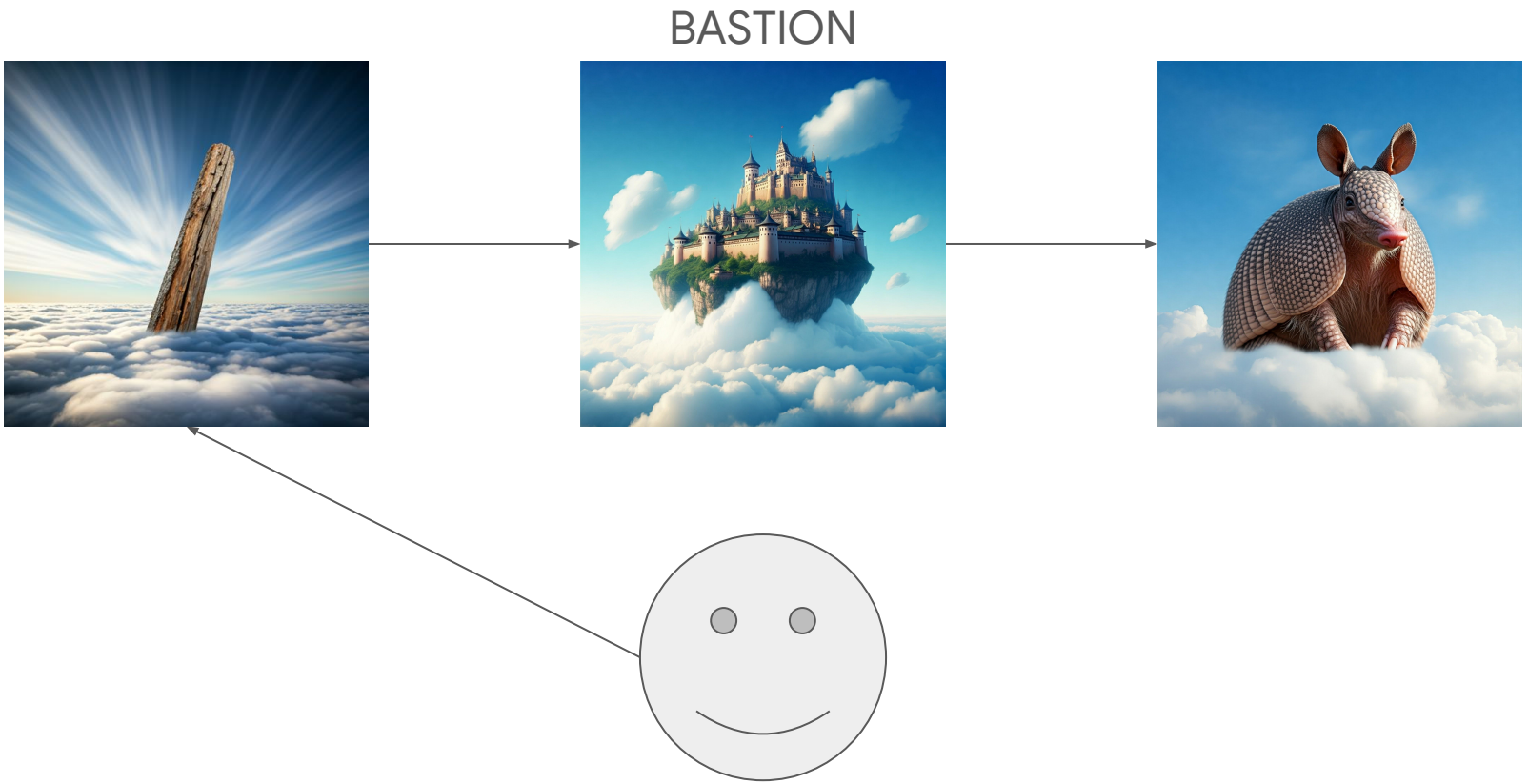
- Accountability
- Showcase

How does it work?

Two ways:

- Synchronous
- Asynchronous

Sync Flow



Async Flow

DISTRIBUTOR

Distributor

Makes witnessed checkpoints easily available via a simple API

- Witnesses stay hidden
- Clients get as many sigs as they request

Demo: Distributor

```
> go run github.com/transparency-dev/distributor/cmd/client@main --n=3

...
Log "sum.golang.org" (a32d071739c062f4973f1db8cc1069f517428d77105962b285bbf918c4062591)
✅ Got checkpoint:

go.sum database tree
30446855
09K73A2OPhyKqqJBdeHiQ89Mr8YBQ3MZv6CWzMKys+0=

— sum.golang.org Az3gru0N2RlF6cTv4gDgb9jL/41foh2RRfRBQjiWk0eAgfyK6JXDt1g1uTjATqGrJrVNUyyr7SLRwEFYQHe0N+JjgAE=
— ArmoredWitness-snowy-sound
iEWJ76Sm+mYAAAAAomo5F9A/j9+FaiYhZ127gAj0DOO92KHRMSEsZvJqVy9Be/99mV9HmBFO+j21he61mouNXFbVukaFdf9PN7DyBg==
— ArmoredWitness-small-breeze
nRq5Ubam+mYAAAAAuByf3Fn6LWZ++Mo6qfwERcBKoW+gxFUztpzzhft75EAAIKUfOD9ssQers/izj7BWjYYpAICDdq5DE1ylUxViCA==
— ArmoredWitness-dry-sunset
uS6j96Gm+mYAAAAAwfmOma3668VmgGCaGSvldlr+92VyqK7nUycbY0RYm9oHfaRATpCV9fngl8EqA9kOOx/orB8VBRhLOqzXIXKXCg==

Witness timestamps:
— ArmoredWitness-snowy-sound: 3 minutes ago (2024-09-30T14:24:52+01:00)
— ArmoredWitness-small-breeze: 2 minutes ago (2024-09-30T14:25:10+01:00)
— ArmoredWitness-dry-sunset: 3 minutes ago (2024-09-30T14:24:49+01:00)

...
```

Recap

A lightweight "plug and play" witness implementation.

Goals:

1. Help kick-start a diverse **witness network**
2. Be reasonably **resistant to tampering** and team coercion
3. Serve as a reference **example of firmware transparency**

Armored Witness

Building a Trusted Notary unikernel

Kick-starting a cross-ecosystem
witness network

Andrea Barisani

@AndreaBarisani - @lcars@infosec.exchange - https://andrea.bio

andrea@inversepath.com | andrea.barisani@withsecure.com

$ whoami

Andrea Barisani

Information Security Engineer and Researcher

Founder: INVERSE PATH (acquired in 2017)

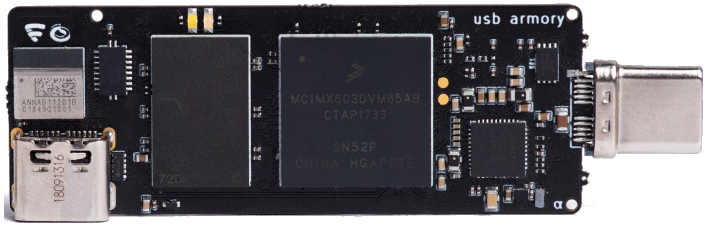
Head of Hardware Security: WITH secure

USB armory  and TamaGo 

Spoke at too many conferences...

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

USB armory



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.

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

Building a hardware witness

The key goal is to build a device for *custodians* to:

- Help transparency-enabled ecosystems to further tighten their security properties (Go's sum DB, Sigstore, Pixel BT, LVFS, SigSum, Amory Drive).
- Allow **low-touch and maintenance-free** operation.
- Demonstrate and promote **firmware transparency**.

Device goals:

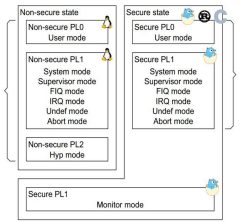
- Full transparency, **open hardware and software**.
- **Reduced attack surface**.
- **Plug-and-go**.

Combining our skills and projects:

USB armory

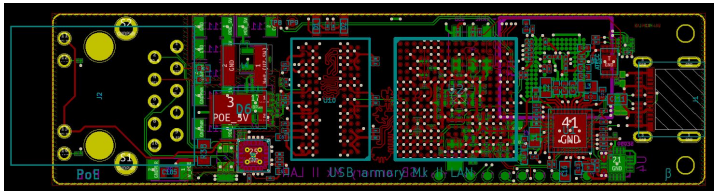
TamaGo

GoTEE



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

USB armory Mk II LAN



A new bespoke variant created specifically for the Armored Witness project.

- RAM: 512 MB or 1 GB DDR3
- Internal storage: 16 GB eMMC
- External secure element: NXP SE050
- SoC: NXP i.MX6UL/i.MX6ULL
  (ARM® Cortex™-A7 528/900 MHz)
- Ethernet: 10/100-Mbps with IEEE 802.3af Power over Ethernet
- USB 2.0 over USB-C: DRP plug

It can be powered by either USB or PoE, acts as USB host or device depending on power mode.

Provides the same security features of the USB only model, full OSS tooling (no NXP blobs).

open hardware

https://github.com/usbarmory/usbarmory/wiki/Mk-II-LAN

Hardware security features

High Assurance Boot (HAB)
SoC Boot ROM authentication of initial bootloader (i.e. Secure Boot).

CAAM (i.MX6UL) | **DCP+RNGB** (i.MX6ULZ)
SoC cryptographic accelerators and TRNG.

Secure Non-Volatile Storage (SNVS)
Encrypted storage of arbitrary data using unique keys, voltage, temperature, clock tamper sensors.

Bus Encryption Engine (BEE)
On i.MX6UL SoC it provides on-the-fly (OTF) AES-128-CTR RAM encryption/decryption.

NXP SE050
External SE with hardware acceleration for elliptic-curve cryptography as well as hardware based key storage.

Replay Protected Memory Block (RPMB)
The internal eMMC allows replay protected authenticated access to flash memory partition areas, using a shared secret between the host and the eMMC.

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

TamaGo: Killing C

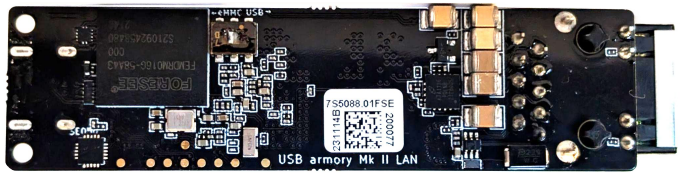
When security matters software and hardware optimizations matter less.

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

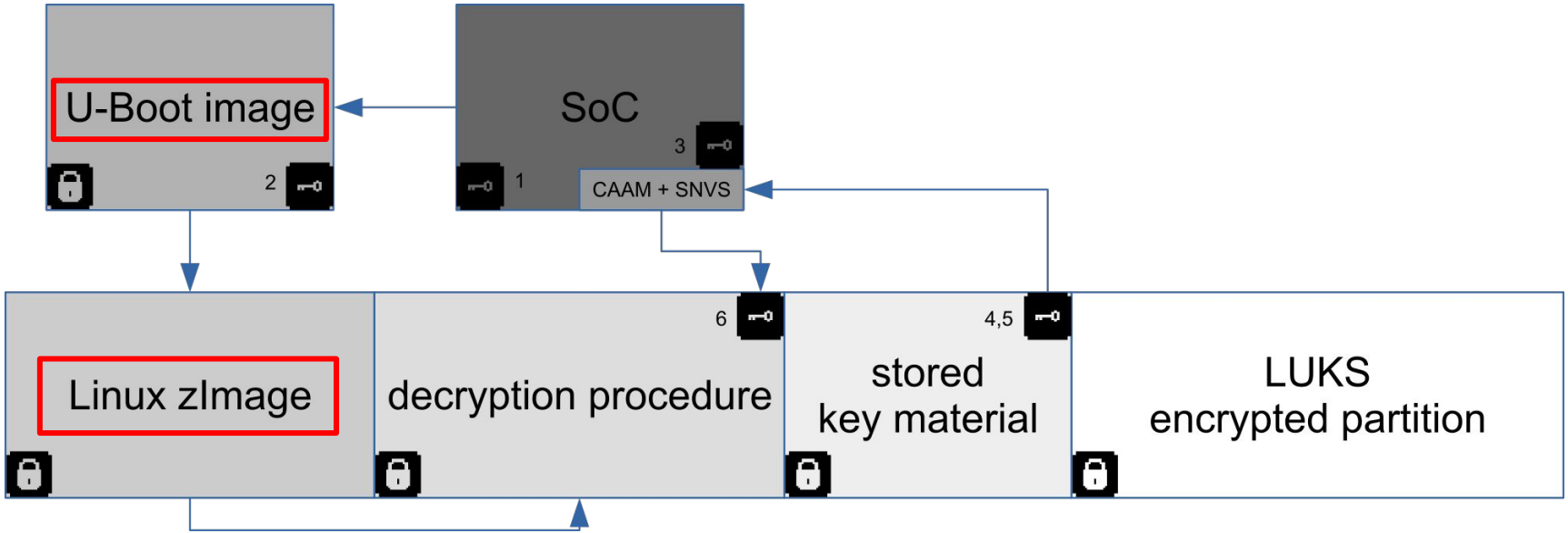
However high level programming typically entails several layers (e.g. OS, libraries) to serve runtime execution.

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**.



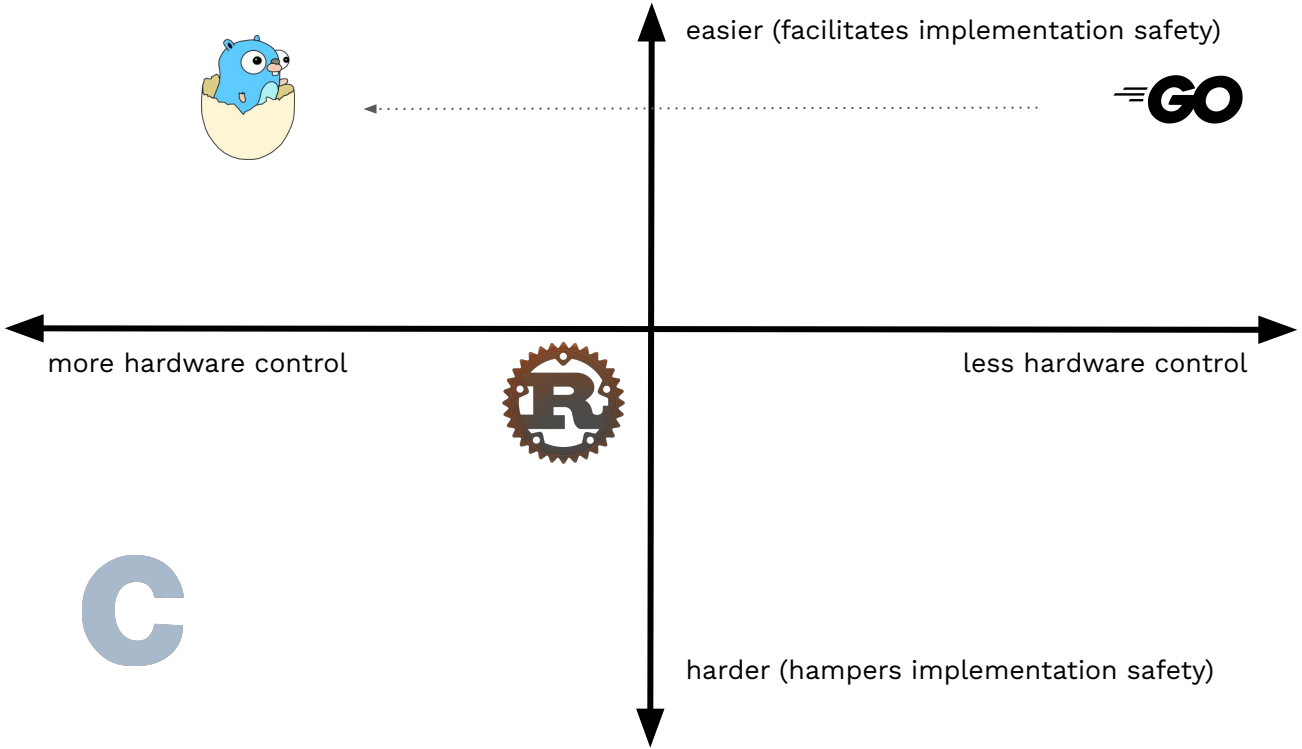
Reducing the attack surface



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

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

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.

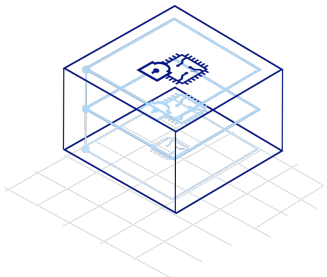
The focus is reducing the attack surface, carrying only strictly necessary code.

“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

TamaGo in a nutshell

TamaGo is made of two main components.

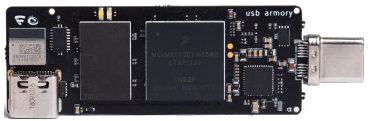
- A **minimally**[1] patched Go distribution to enable `GOOS=tamago` support, which provides freestanding execution on `GOARCH=arm` and `GOARCH=riscv64` bare metal.

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

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

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

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



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

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

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.
- ★ ~5000 LOC of changes against Go distribution with clean separation from other `GOOS` support.
- ★ Strong emphasis on code reuse from existing architectures of standard Go runtime, see Internals[1].
- ★ Requires only one import (“library OS”) on the target Go application.
- ★ Supports unencumbered Go applications with nearly full runtime availability.
- ★ In addition to the compiler, aims to provide a complete set of peripheral drivers for SoCs.

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

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

Go distribution modifications[1]

Glue code - 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` vs simple pointer)
- `js,wasm` locking is re-used identically (with JS VM hooks removed)
- `nacl` in-memory filesystem is re-used (raw SD/MMC access implemented in `imx6`)

New code - basic syscall and memory layout support:

```
rt0_tamago_arm.s
sys_tamago_arm.s
os_tamago_arm.go
```

```
rt0_tamago_riscv64.s
sys_tamago_riscv64.go
os_tamago_riscv64.go
```

https://github.com/golang/go/compare/go1.23.0...usbarmory:tamago1.23.0

[1] As of `tamago1.23.0` against `go1.23.0`

```
$ git diff go1.23.0 tamago1.23.0 --stat --stat-width "$(tput cols)" --color=always -- ":(exclude)*tamago_test.go" | sort -t '|' -n -k2 "$@"

315 files changed, 6468 insertions(+), 325 deletions(-)

 src/cmd/go/internal/imports/build.go                            |   1 +
 ...
 src/runtime/proc.go                                             |  14 +-
 src/syscall/asm_tamago_arm.s                                    |  14 ++
 src/syscall/asm_tamago_riscv64.s                                |  14 ++
 src/internal/syscall/unix/nonblocking_tamago.go                 |  15 ++
 src/os/sys_tamago.go                                            |  15 ++
 src/os/pipe_tamago.go                                           |  16 ++
 src/runtime/malloc.go                                           |  16 +-
 src/runtime/sigqueue_tamago.go                                  |  16 ++
 src/time/zoneinfo_tamago.go                                     |  17 ++
 src/os/signal/signal_tamago.go                                  |  23 +++
 src/runtime/mem_tamago.go                                       |  24 +++
 src/syscall/zsyscall_tamago_arm.go                              |  25 +++
 src/syscall/zsyscall_tamago_riscv64.go                          |  25 +++
 src/internal/goos/zgoos_tamago.go                               |  27 +++
 src/crypto/rand/rand_tamago.go                                  |  29 +++
 src/os/exec/lp_tamago.go                                        |  29 +++
 src/os/dirent_tamago.go                                         |  30 +++
 src/os/user/lookup_tamago.go                                    |  35 ++++
 src/net/sockopt_tamago.go                                       |  37 ++++
 src/runtime/rt0_tamago_riscv64.s                                |  42 ++++
 src/internal/syscall/unix/net_tamago.go                         |  44 ++++
 src/os/stat_tamago.go                                           |  51 +++++
 src/time/sys_tamago.go                                          |  54 +++++
 src/testing/testing_tamago.go                                   |  55 +++++
 src/runtime/rt0_tamago_arm.s                                    |  56 +++++
 src/testing/testing_tamago.s                                    |  64 ++++++
 src/testing/run_example_tamago.go                               |  76 +++++++
 src/runtime/sys_tamago_riscv64.s                                | 122 +++++++++++
 src/runtime/lock_tamago.go                                      | 169 +++++++++++++++
 src/net/net_tamago.go                                           | 210 +++++++++++++++++++
 src/runtime/sys_tamago_arm.s                                    | 220 ++++++++++++++++++++
 src/runtime/os_tamago_riscv64.go                                | 226 ++++++++++++++++++++
 src/runtime/os_tamago_arm.go                                    | 251 +++++++++++++++++++++++
 src/syscall/fd_tamago.go                                        | 261 +++++++++++++++++++++++
 src/syscall/syscall_tamago.go                                   | 328 +++++++++++++++++++++++++++++
 src/syscall/tables_tamago.go                                    | 494 ++++++++++++++++++++++++++++++++++++++++++++++
 src/syscall/fs_tamago.go                                        | 872 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/syscall/net_tamago.go                                       | 954 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Go low level access

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

	hw.ctrl = hw.Base + BEE_CTRL
	hw.addr0 = hw.Base + BEE_ADDR_OFFSET0
	hw.addr1 = hw.Base + BEE_ADDR_OFFSET1
	hw.key = hw.Base + BEE_AES_KEY0_W0
	hw.nonce = hw.Base + BEE_AES_KEY1_W0

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

	// disable
	reg.Clear(hw.ctrl, CTRL_BEE_ENABLE)
}

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: BEE initialization

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.

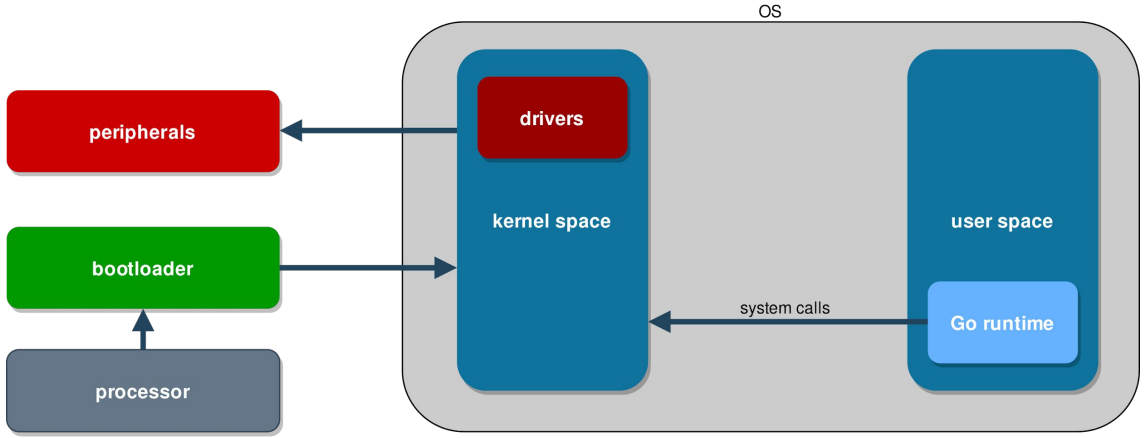
Applications are never required to use any `unsafe` function.

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

https://github.com/usbarmory/tamago

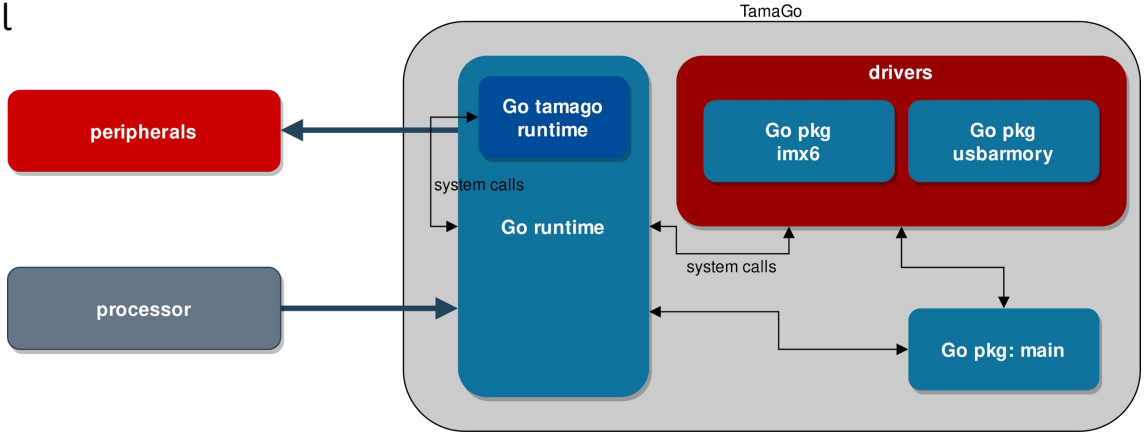
TamaGo

W / T H secure



Traditional OS

TamaGo unikernel



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

Developing, building and running

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

```
package main

import (
        _ "github.com/usbarmory/tamago/board/usbarmory/mk2"
)

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

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

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

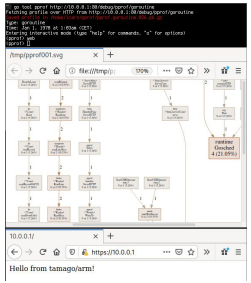
Firmware can be compiled just as easily on Linux, Windows, macOS.

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

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

Reducing the attack surface

Block	LOCs	Driver support
ARM	900	CPU 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)
MK2	680	USB armory Mk II board support package

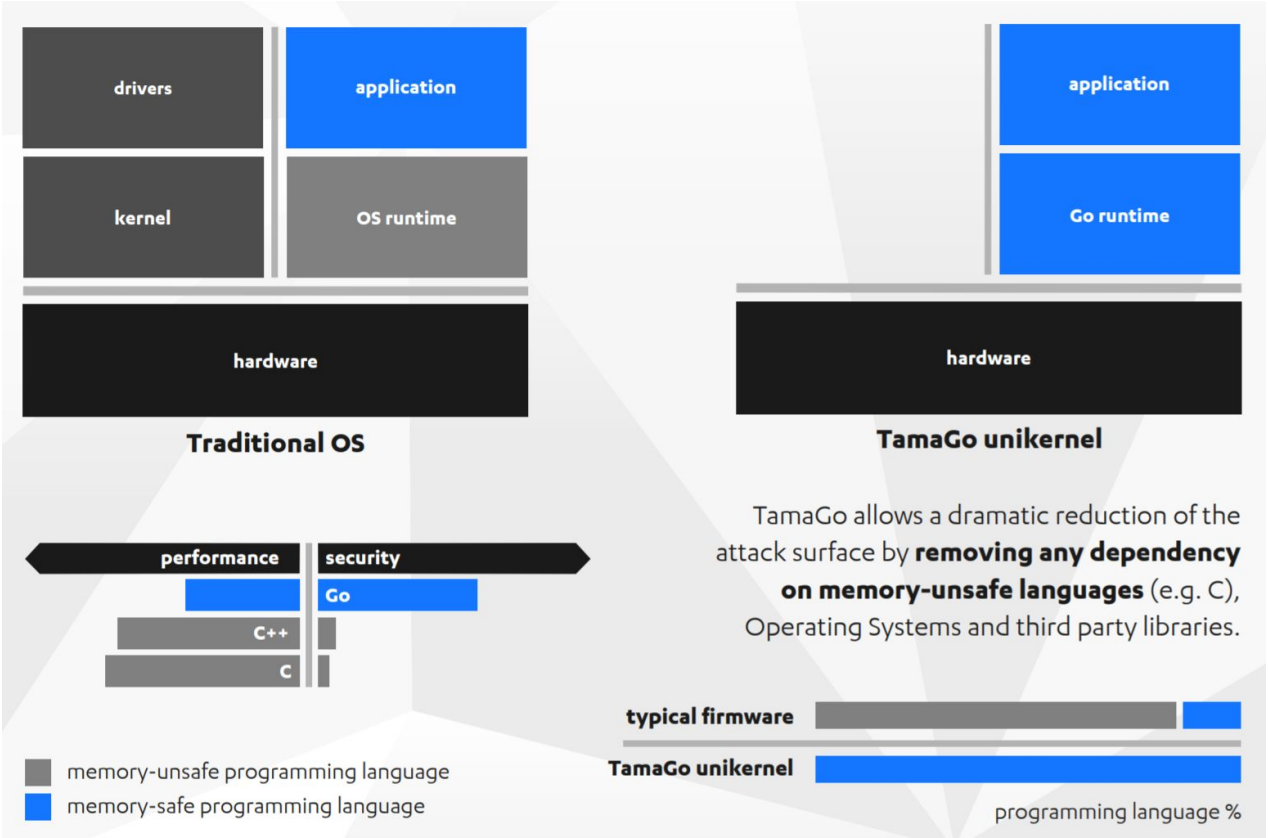


```
module github.com/usbarmory/GoTEE                                    go.mod

go 1.23.0

require github.com/usbarmory/tamago v0.0.0-20240924114619-273d67cd811d
```

The TamaGo firmware 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

Improving memory safety



```
$ 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 |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

WITH secure

GoKey - The bare metal Go smart card

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

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

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

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

https://github.com/usbarmory/gokey

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

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

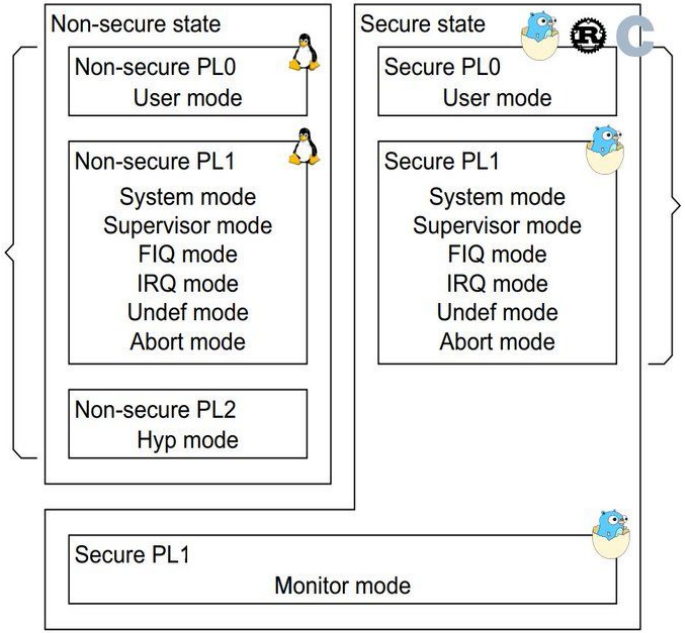
[1] https://github.com/usbarmory/usbarmory/wiki/Secure-boot-(Mk-II)

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.

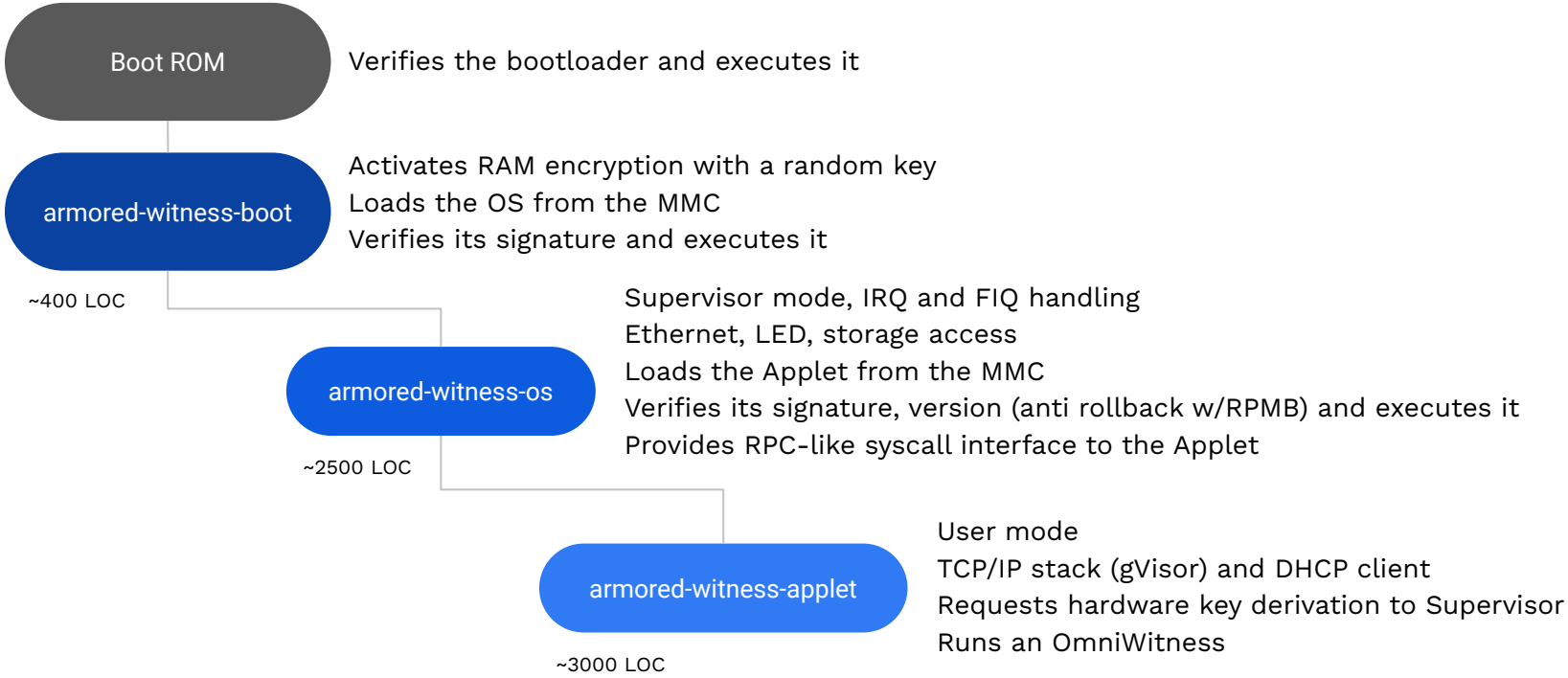
It supports any freestanding user mode applets (e.g. TamaGo, C, Rust) and any “rich” OS running in NonSecure World (e.g. Linux).



https://github.com/usbarmory/GoTEE

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

Building a hardware witness



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

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)
}
```

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(pre 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))
}
```

type **RPMB**

RPMB defines a Replay Protected Memory Block partition access instance.

func Init

```
func Init(card *usdhc.USDHC, key []byte, dummyBlock uint16, writeDummy bool) (p *RPMB, err error)
```

Init returns a new RPMB instance, dummyBlock argument is an unused sector, required for CVE-2020-13799 mitigation to invalidate uncommitted writes.

func (*RPMB) Counter

```
func (p *RPMB) Counter(auth bool) (n uint32, err error)
```

Counter returns the RPMB partition write counter, the argument boolean indicates whether the read operation should be authenticated.

func (*RPMB) ProgramKey

```
func (p *RPMB) ProgramKey() (err error)
```

ProgramKey programs the RPMB partition authentication key.

func (*RPMB) Read

```
func (p *RPMB) Read(offset uint16, buf []byte) (err error)
```

Read performs an authenticated data transfer from the card RPMB partition, the input buffer can contain up to 256 bytes of data.

func (*RPMB) Write

```
func (p *RPMB) Write(offset uint16, buf []byte) (err error)
```

Write performs an authenticated data transfer to the card RPMB partition, the input buffer can contain up to 256 bytes of data.

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.

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.

type RPC

RPC represents an example receiver for user/system mode RPC over system calls.

```
type RPC struct {
     RPMB        *RPMB
     Storage     Card
     Ctx         *monitor.ExecCtx
     // sha256.Sum256([]byte(AppletManifestVerifier))
     Diversifier [32]byte
}
```

```
import (
          “github.com/usbarmory/GoTEE/applet”
          “github.com/usbarmory/GoTEE/syscall”
)

// underlying implementation uses Go stdlib net/rpc/jsonrpc
if err := syscall.Call("RPC.Status", nil, &status); err != nil {
                    return fmt.Errorf("failed to fetch Status: %v", err)
}
```

func (*RPC) DeriveKey

```
func (r *RPC) DeriveKey(diversifier [aes.BlockSize]byte, key *[sha256.Size]byte) (err error)
```

DeriveKey derives a hardware unique key ,the diversifier is AES-CBC encrypted using the internal OTPMK key.

func (*RPC) HAB

```
func (r *RPC) HAB(srk []byte, _ *bool) error
```

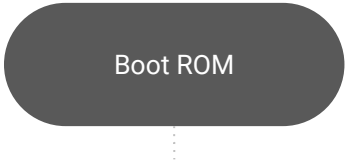
HAB activates secure boot.

func (*RPC) ReadRPMB

```
func (r *RPC) ReadRPMB(buf []byte, n *uint32) error
```

ReadRPMB performs an authenticated data transfer from the card RPMB partition sector allocated to the Trusted Applet. The input buffer can contain up to 256 bytes of data, n can be set to retrieve the partition write counter.

Trust Anchors



Secure Boot keys are created and fused by the **device owner** using open source tools. A low level USB protocol (SDP) allows to interact with the Boot ROM to perform register read, write and firmware load, this allows **inspection** of a unit provisioning state and contents **without its firmware interference**. The Boot ROM remains the only blob (can be read for RE).

func **BuildDCDWriteReport**

```
func BuildDCDWriteReport(dcd []byte, addr uint32) (r1 []byte, r2 []byte)
```

BuildDCDWriteReport generates USB HID reports (IDs 1 and 2) that implement the SDP DCD_WRITE command sequence, used to load a DCD binary payload (p327, 8.9.3.1.5 DCD_WRITE, IMX6ULLRM).

func **BuildFileWriteReport**

```
func BuildFileWriteReport(imx []byte, addr uint32) (r1 []byte, r2 [][]byte)
```

BuildFileWriteReport generates USB HID reports (IDs1 and 2) that implement the SDP FILE_WRITE command sequence, used to load an imx binary payload (p325, 8.9.3.1.3 FILE_WRITE, IMX6ULLRM).

func **BuildJumpAddressReport**

```
func BuildJumpAddressReport(addruint32) (r1 []byte)
```

BuildJumpAddressReport generates the USB HID report (ID 1) that implements the SDP JUMP_ADDRESS command, used to execute an imx binary payload (p328, 8.9.3.1.7 JUMP_ADDRESS, IMX6ULLRM).

func **BuildReadRegisterReport**

```
func BuildReadRegisterReport(addruint32, size uint32) (r1 []byte)
```

BuildReadRegisterReport generates USB HID reports (ID 1) that implement the SDP READ_REGISTER command for reading a single 32-bit device register value (p323, 8.9.3.1.1 READ_REGISTER, IMX6ULLRM).

https://github.com/usbarmory/crucible

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

WITH secure

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

Firmware Transparency

100% open source*

Firmware: Bootloader & Recovery, OS, and Applet.

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

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

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

Tooling, docs, build pipeline config, witness IDs, public keys, etc.:

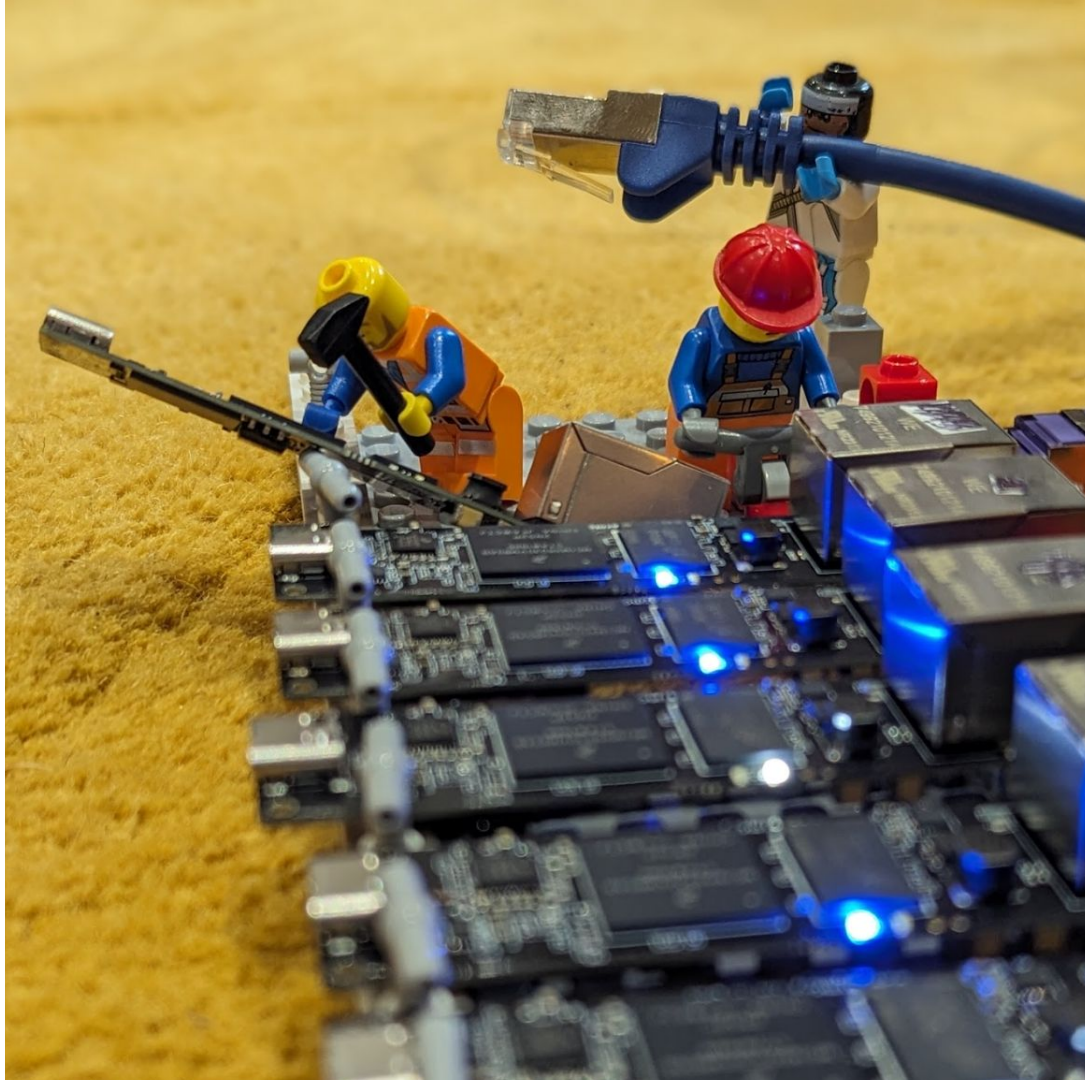
https://github.com/transparency-dev/armored-witness

Firmware Builds

4 images:
Bootloader, **Monitor**, **Applet**, **Recovery**

All are 100% bare-metal Go (TamaGo).

Automatically **built, signed, and transparency logged** by build pipeline.



Firmware Transparency: logged metadata


```
{
  "schema_version": 0,
  "component": "TRUSTED_APPLET",
  "git": {
    "tag_name": "0.2.1",
    "commit_fingerprint": "b91543149d15d0166cd9470f0547d6022f108eb2"
  },
  "build": {
    "tamago_version": "1.22.4",
    "envs": [
      "FT_LOG_URL=https://api.transparency.dev/armored-witness-firmware/prod/log/1",
      "FT_BIN_URL=https://api.transparency.dev/armored-witness-firmware/prod/artefacts/1",
      "LOG_ORIGIN=transparency.dev/armored-witness/firmware_transparency/prod/1",
      "LOG_PUBLIC_KEY=transparency.dev-aw-ftlog-prod-1+3e6d87ee+Aa3qdhefd2cc/98jV3blslJT2L+iFR8WKHeGcgFmyjnt",
      "APPLET_PUBLIC_KEY=transparency.dev-aw-applet-prod+d45f2a0d+AZSnFa8GxH+jHV6ahELk6peqVObbPKrYAdYyMjrzNF35",
      "OS_PUBLIC_KEY1=transparency.dev-aw-os1-prod+985bdfd2+AV7mmRamQp6VC9CutzSXzqtNhYNyNmQQRcLX07F6qlC1",
      "OS_PUBLIC_KEY2=transparency.dev-aw-os2-prod+662add8c+AebLJIKJhx57T3mWmHKe0sasFnXmtIQNTGRaoj2PQLrY",
      "REST_DISTRIBUTOR_BASE_URL=https://api.transparency.dev/prod",
      "BASTION_ADDR=bastion.glasklar.is:443",
      "BEE=1",
      "DEBUG=1",
      "SRK_HASH=77e021cc51b5547fb0c2192fb32710bfa89b4bbaa7dab5f97fc585f673b0b236"
    ]
  },
  "output": {
    "firmware_digest_sha256": "PcM1yItuFrBxRgo5CaLIufmDTDREqMfbi/nh30nQVhg="
  }
}

— transparency.dev-aw-applet-prod 1F8qDWHhg7KdKDGEbHRY9ugcKEaINmPWrwIBgha9YvpywRW9IpFqbfedLAxrkMJ6rUc1QF/G9uIDrkFtKEvyyOjLNQE=
```

But is that claim true?

Yes [*hopefully* 🤞] ... but don't trust us; try it for yourself!

https://github.com/transparency-dev/armored-witness/tree/main/cmd/verify_build

```
$ docker build . -t armored-witness-build-verifier -f ./cmd/verify_build/Dockerfile
$ docker run armored-witness-build-verifier
...
I0219 12:23:00.470332       1 verify.go:162] Leaf index 125: ✅ reproduced build BOOTLOADER@0.0.1707929407-incompatible
(373ce9ef15cc7937e1dc024a7288e4d4b1c33eab) => 7ac229b8c166d26c93006586ffb4e46a0f13c31d881fda85b09816f88c1ebc31
E0219 12:23:01.898909       1 verify.go:158] Leaf index 126: ❌ failed to reproduce build RECOVERY@0.0.0
(74060722c9aa92bbdcf3725ed0d0be4ebe8f8687) => (got b7e32298bb284c92f1cdceaffa39fa8840d21b19e4b08167023985a4a60206b2,
wanted d59b4eaf94b1a895264b2a11eaee60c2a5a89c4a5b115fe29d78171187fb4df1)
I0219 12:23:01.898934       1 verify.go:94] 🔎 Evidence of failed build: /tmp/armored-witness-build-verify2690175259
(126: RECOVERY@74060722c9aa92bbdcf3725ed0d0be4ebe8f8687)
I0219 12:23:02.815427       1 verify.go:162] Leaf index 127: ✅ reproduced build RECOVERY@0.0.0
(850baf54809bd29548d6f817933240043400a4e1) => b7e32298bb284c92f1cdceaffa39fa8840d21b19e4b08167023985a4a60206b2
I0219 12:23:03.857585       1 verify.go:162] Leaf index 128: ✅ reproduced build BOOTLOADER@0.0.1707998563-incompatible
(6062287365c4d7bab79532940d70d1bab846ef78) => 70d2fc2cc57de625f9a04e8923e2e0a99060c7694c3df758def16ca0f030aa4c
```

Provisioning

Turns "blank" device into an ArmoredWitness:

https://github.com/transparency-dev/armored-witness/tree/main/cmd/provision

- Uses a compiled-in template for configuration: "CI" or "PROD"
  - Which FT log to use for firmware images to install
  - Which keys should be used to verify images & checkpoint signatures during installation
- Downloads firmware images from FT log, verifies inclusion, signatures etc.
- Flashes images + offline proofs onto device.
- Permanently fuses device into Secure Boot

Custodian verification

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

```
$ sudo ${PWD}/verify --template=prod
I0412 10:40:28.663867 2193170 main.go:98] Using template flag setting --firmware_log_url=https://api.transparency.dev/armored-witness-firmware/prod/log/1/
... <More template flags elided>
I0412 10:40:28.663982 2193170 main.go:98] Using template flag setting --os_verifier_2=transparency.dev-aw-os2-prod+662add8c+AebLJIKJhx57T3mWmHKe...
I0412 10:40:29.681000 2193170 fetcher.go:88] Fetching RECOVERY bin from "8271e2a8ccefb6c4df48889fcbb35343511501e3bcd527317d9e63e2ac7349e3"
I0412 10:40:29.879505 2193170 main.go:217] Successfully fetched and verified recovery image
I0412 10:40:29.879519 2193170 main.go:218] ----------------------------------------------------------------------------------------
I0412 10:40:29.879523 2193170 main.go:219] 🔷🔷🔷 🙋 OPERATOR: please ensure boot switch is set to USB, and then connect device 🙏
I0412 10:40:29.879526 2193170 main.go:220] ----------------------------------------------------------------------------------------
I0412 10:40:29.879530 2193170 main.go:223] Recovery firmware is 1924096 bytes + 16384 bytes HAB signature
I0412 10:40:29.879540 2193170 recovery.go:64] Waiting for device to be detected...
I0412 10:46:13.033524 2193170 sdp.go:85] found device 15a2:007d Freescale SemiConductor Inc  SE Blank 6UL
I0412 10:46:13.092825 2193170 sdp.go:111] Attempting to SDP boot device /dev/hidraw0
I0412 10:46:13.092912 2193170 sdp.go:123] Loading DCD at 0x00910000 (976 bytes)
I0412 10:46:13.096387 2193170 sdp.go:128] Loading imx to 0x8000f400 (1940480 bytes)
I0412 10:46:14.288277 2193170 sdp.go:133] Sending jump address to 0x8000f400
I0412 10:46:14.288651 2193170 sdp.go:138] Serial download on /dev/hidraw0 complete
I0412 10:46:15.289152 2193170 recovery.go:51] Witness device booting recovery image
I0412 10:46:15.289210 2193170 recovery.go:106] Waiting for block device to appear
I0412 10:46:18.876369 2193170 recovery.go:118] Waiting for block device to settle...
I0412 10:46:19.897030 2193170 main.go:230] ✅ Detected device "/dev/hidraw0"
I0412 10:46:19.897079 2193170 main.go:231] ✅ Detected blockdevice /dev/disk/by-id/usb-F-Secure_USB_armory_Mk_II_720A9DEAD4413E39-0:0
I0412 10:46:19.900369 2193170 main.go:370] Found config at block 0x4fb0
I0412 10:46:19.900394 2193170 main.go:375] Reading 0x2d6c00 bytes of firmware from MMC byte offset 0x400
I0412 10:46:20.045080 2193170 main.go:370] Found config at block 0x5000
I0412 10:46:20.045122 2193170 main.go:375] Reading 0xdcfe65 bytes of firmware from MMC byte offset 0xa0a000
I0412 10:46:20.765940 2193170 main.go:370] Found config at block 0x200000
I0412 10:46:20.765988 2193170 main.go:375] Reading 0xf09521 bytes of firmware from MMC byte offset 0x4000a000
I0412 10:46:21.695851 2193170 main.go:291]   ✅ Bootloader: proof bundle is self-consistent
I0412 10:46:21.695923 2193170 main.go:314]   ✅ Bootloader: proof bundle checkpoint(@7) is consistent with current view of log(@7)
I0412 10:46:21.714944 2193170 main.go:291]   ✅ TrustedOS: proof bundle is self-consistent
I0412 10:46:21.715029 2193170 main.go:314]   ✅ TrustedOS: proof bundle checkpoint(@7) is consistent with current view of log(@7)
I0412 10:46:21.735579 2193170 main.go:291]   ✅ TrustedApplet: proof bundle is self-consistent
I0412 10:46:21.735666 2193170 main.go:314]   ✅ TrustedApplet: proof bundle checkpoint(@7) is consistent with current view of log(@7)
I0412 10:46:21.735672 2193170 main.go:128] ✅ Device verified OK!
```

Boot security

ROM	Bootloader	Trusted OS	Trusted Applet

SecureBoot fuses blown.
 	
Bootloader is authenticated via key fused at provision time.
	Bootloader:
	- Enables bus encryption
	- Loads/verifies OS+proof
	  - metadata signature
	  - hash(OS)
	  - Checkpoint signature
	  - Inclusion proof of meta
	- Jumps to OS
		OS:
		- Manages H/W
		- Provides services to applet
		- Loads/verifies Applet+proof
		  - metadata signature
		  - hash(Applet)
		  - Checkpoint signature
		  - Inclusion proof of meta
		- Launches Applet in TEE
			Applet (TEE):
			- Runs witness code
			- Runs TCP/IP stack
			- Responsible for f/w updates

Witness IDs & Keys

Derived on-device from per-device secrets in H/W security module (CAAM).

Corresponding public keys are here:
https://github.com/transparency-dev/armored-witness/tree/main/devices

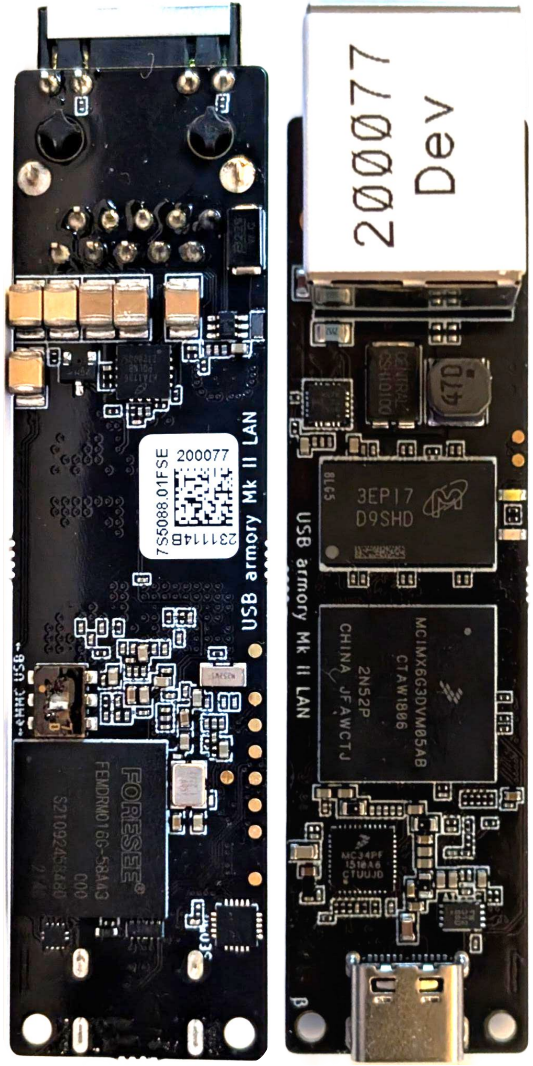
So... what?

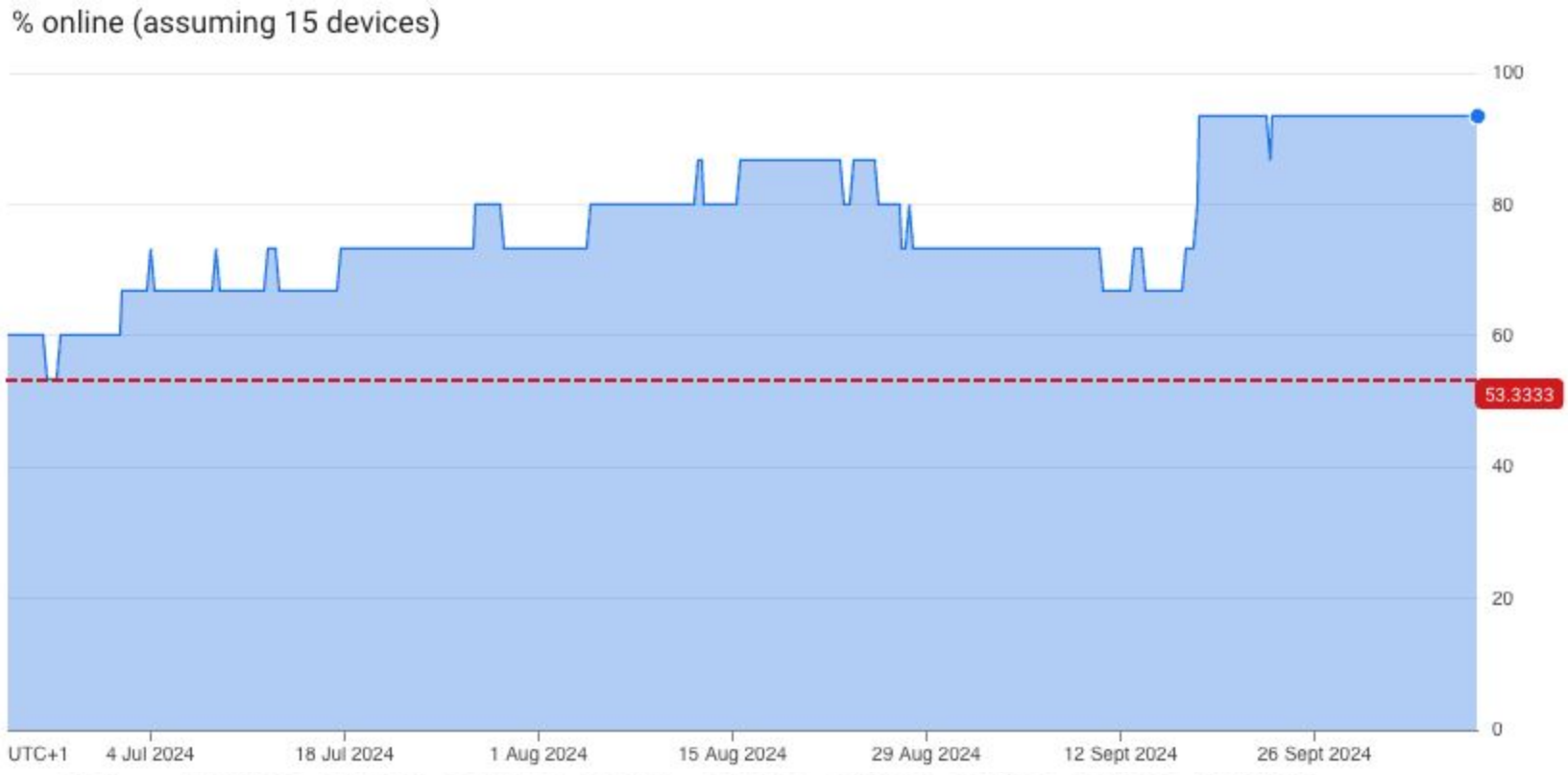
In essence:

It makes it hard for us, or anyone else, to **covertly** tamper with a device.

- Custodians can at any time **verify** and know *the totality* of what's running.
  - ⇒ Custodians can **inspect source@<commit>** and see *exactly* what it's programmed to do.
  - ⇒ They **call us out** if anything looks wrong.
  - ⇒ TrustFabric can't ship devices out with **dodgy covert firmware pre-installed**.
- TrustFabric **cannot sneak dodgy firmware onto devices** via updates.
  - ⇒ We'd have to do it in the open, and leave a trail of **easily discoverable evidence.**
- Anyone with **physical access can't boot inauthentic firmware**.
  - ⇒ Nor older authentic firmware versions.
- Impenetrable and uncrackable?
  - ⇒ No, of course not.
  - ⇒ But to do much you'd probably need to physically get hold of at least half of them without anyone noticing...

Thanks!


200077
Dev
USB armory Mk II LAN
7S5088.01FSE
200077
231114B
~eMMC USB~
FORESEE
FEMDRM016-58A43
3EP17
D9SHD
MCIMX6G3DVM05AB
CTAW1806
2N52P
CHINA JFAWCTJ
USB armory Mk II LAN


% online (assuming 15 devices)
100
80
60
53.3333
40
20
0
UTC+1
4 Jul 2024
18 Jul 2024
1 Aug 2024
15 Aug 2024
29 Aug 2024
12 Sept 2024
26 Sept 2024