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 

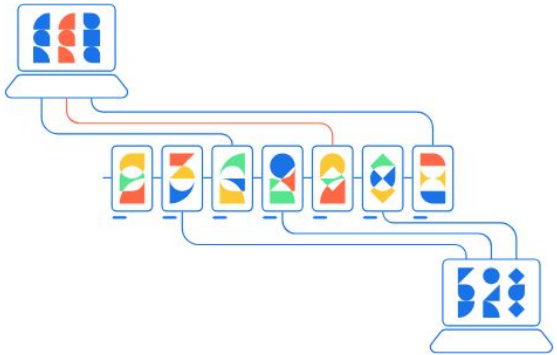
Spoke at too many conferences...

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

@AndreaBarisani | @lcars@infosec.exchange

https://andrea.bio

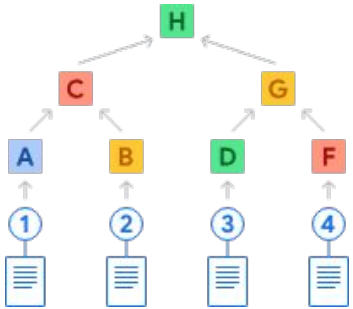
Google Transparency Project



An open-source append only ledger designed to discourage insider threats.

Based on Trillian, an open-source verifiable log.

Merkle trees are used, a log record cannot be modified without cascading the change to the tree head hash.



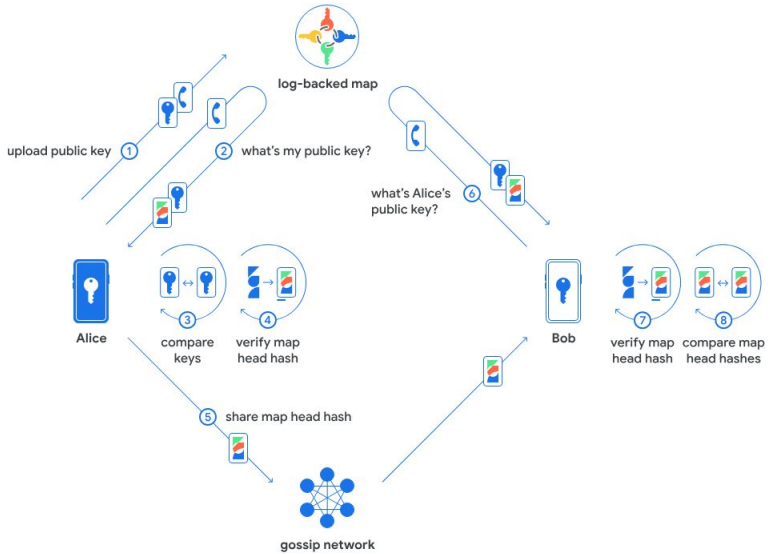
https://transparency.dev/

https://transparency.dev/verifiable-data-structures/

Certificate Transparency

Certificate Transparency allows to see which CAs have issued which certificates, when, and for which domains.

Detect maliciously or mistakenly issued certificates.



https://certificate.transparency.dev/

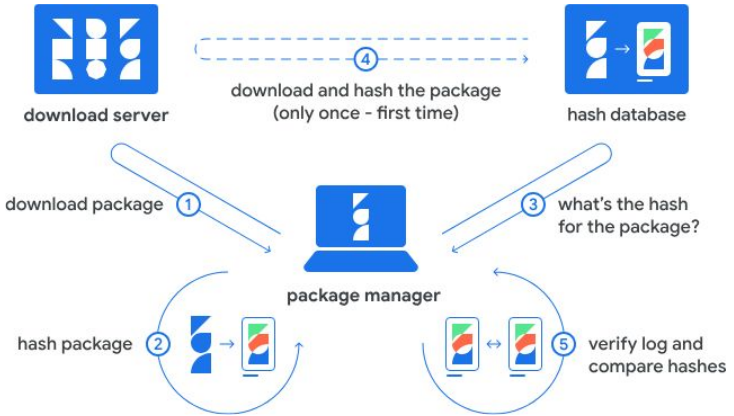
Tamper checking: package managers

A transparent log is hosted on a server and made accessible to clients which are able to verify that a particular log record really is in the log and also that the server never removes any log record from the log.

The log server is not trusted to store the log properly, nor is it trusted to put the right records into the log.

Instead, clients and **auditors interact skeptically with the server**, able to verify for themselves in each interaction that the server really is behaving correctly.

The go command performs “inclusion” proofs (that a specific record exists in the log) and “consistency” proofs (that the tree hasn’t been tampered with) before adding new go.sum lines to the main module’s go.sum file.



go.mod

```
module github.com/usbarmory/GoTEE

go 1.22.0

require github.com/usbarmory/tamago v0.0.0-20240104082716-7fdd041b36ef
```

go.sum

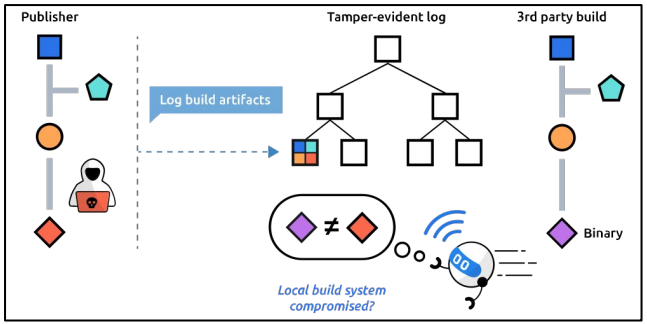
```
github.com/usbarmory/tamago v0.0.0-20240104082716-7fdd041b36ef h1:/4FEy+WnOsA06twAR0UWFLEBU2KRhYzkh+jmvlsI4f4=
github.com/usbarmory/tamago v0.0.0-20240104082716-7fdd041b36ef/go.mod h1:uCPXcPo8SZulhZPz8irfVqzwVlPZ45w7CTJxkfxueGA=
```

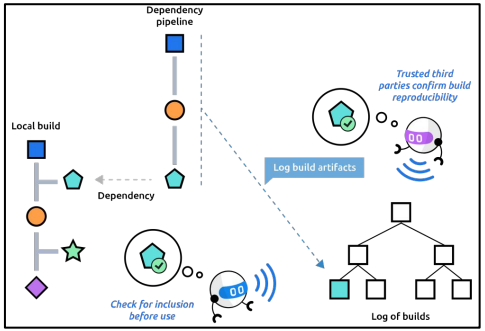
https://go.googlesource.com/proposal/+/master/design/25530-sumdb.md

https://go.dev/ref/mod#authenticating

Binary Transparency

A software supply chain can be protected by an immutable and tamper-evident record which can be viewed and verified by others.





https:/binary.transparency.dev

Build reproducibility, in combination with tamper evident records, allows verification of opaque firmware binaries required for third party locked down hardware (e.g. secure booted systems).

Release manifest

A JSON *release manifest* exists for every published Armory Drive firmware. This manifest contains information about the release, including the firmware's SHA256 checksum:


```
{
  "description": "armory-drive v2021.09.22",
  "platform_id": "UA-MKII-ULZ",
  "revision": "v2021.09.22",
  "artifact_sha256": {
    "armory-drive.csf": "/f636l9GHbwub5gKdaWKP7/DA6/vPoa1JXDfGxfb6vk=",
    "armory-drive.imx": "MVorRrFrzXT89Gu+FE4/aeX12QiGPcRoHiLuR6Kq+6g=",
    "armory-drive.sdp": "+G9bzK+phhIXb5786VW79poRvLg4T7EnoggDV0E0tL8="
  },
  "source_url": "https://github.com/usbarmory/armory-drive/tarball/v2021.09.22",
  "source_sha256": "X6rkBnT+Ht3kBcJXJ7YK4RvIXEZ16VnkT5xy6iNVvBc=",
  "tool_chain": "tamago version go1.17.1 linux/amd64",
  "build_args": {
    "REV": "efeb733"
  }
}
```

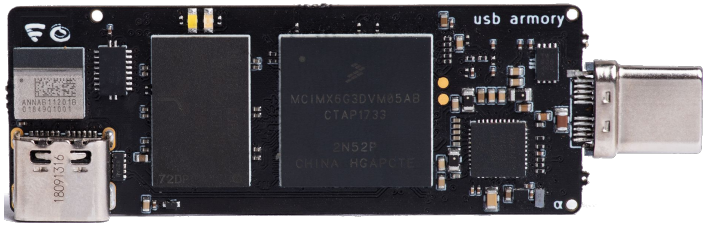

The manifest is signed with the F-Secure Armory Drive Firmware private key. The signature, in Go sumdb signed note format, is appended to the release manifest:


```
— armory-drive WpWkG+RY85Ha0j+iWxLeWI+GV8eRBnxOttH7wb3Pu4g5v2E2zCHnop0+UkR77l3YH1O8+tVwdLiTvVDTasagTsUE
```


These signed release manifests are generated with the create_release tool.

https://github.com/usbarmory/armory-drive/wiki/Firmware-Transparency

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

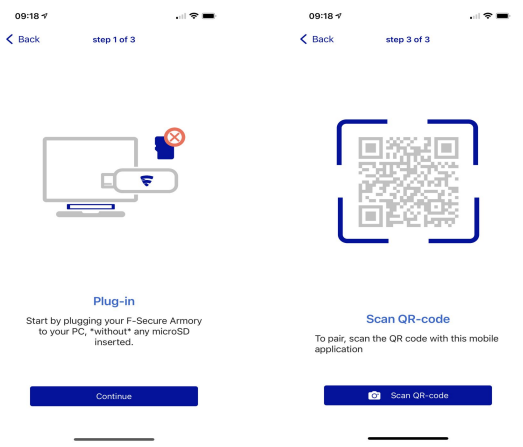
https://github.com/usbarmory/usbarmory/wiki

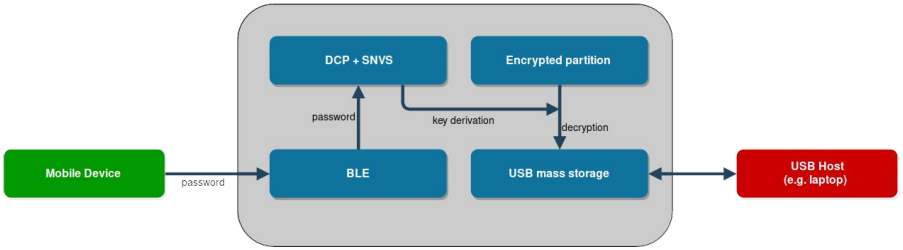
Armory Drive - Encrypted USB Mass Storage

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 (more on this later) code and an iOS app.





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

https://www.youtube.com/watch?v=Sv2wdVkp03A

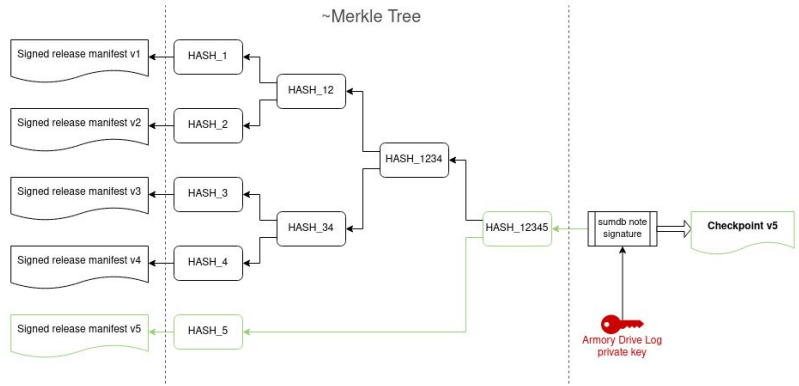
Armory Drive - Encrypted USB Mass Storage

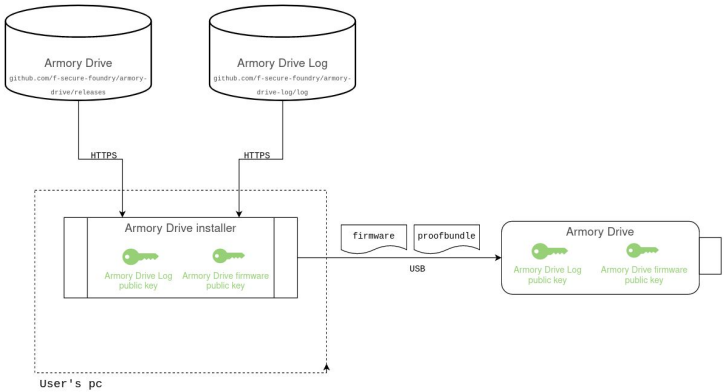
Armory Drive leverages on **Firmware Transparency** to enable advanced firmware update authentication on the installer as well as the device itself.

The installer and firmware work together through a combination of Secure Boot and Firmware Transparency frameworks to authenticate firmware updates.

Secure boot allows firmware authentication with burned in read-only public keys, as well as confidential configuration storage with device specific hardware keys.

Users can choose Secure Boot with own keys, manufacturer keys or none at all (with reduced security).





https://github.com/usbarmory/armory-drive/wiki/Firmware-Transparency

```
$ armory-drive-install -I

Welcome to the Armory Drive installer!

For more information or support on Armory Drive see:
  https://github.com/usbarmory/armory-drive/wiki

This program will install or upgrade Armory Drive on your USB armory.

████████████████████████████████████████████████████████████████████████

This installer supports installation of unsigned or signed Armory Drive
releases on the USB armory.

                ***     Option #1: signed releases     ***

The installation of signed releases activates Secure Boot on the target USB
armory, fully converting the device to exclusive operation with signed
executables.

If the signed releases option is chosen you will be given the option of using
F-Secure signing keys or your own.

                ***    Option #2: unsigned releases    ***

The installation of unsigned releases does not leverage on Secure Boot and does
not permanently modify the USB armory security state.

Unsigned releases however cannot guarantee device security as hardware bound
key material will use default test keys, lacking protection for stored armory
communication keys and leaving data encryption key freshness only to the mobile
application.

Unsigned releases are recommended only for test/evaluation purposes and are not
recommended for protection of sensitive data where device tampering is a risk.

████████████████████████████████████████████████████████████████████████


Would you like to use unsigned releases, *without enabling* Secure Boot on the USB armory? (y/N): N

Would you like to *permanently enable* Secure Boot on the USB armory? (y/N): y

Would you like to use F-Secure signed releases, enabling Secure Boot on the USB armory with permanent fusing of F-Secure public keys? (y/N): y
```

WITH secure

```
Found HAB signature
  Tag:    v2021.10.08
  Author: andrejro
  Date:   2021-10-08 12:27:18 +0000 UTC
  URL:    https://github.com/usbarmory/armory-drive/releases/download/v2021.10.08/armory-drive.csf

Found binary release
  Tag:    v2021.10.08
  Author: andrejro
  Date:   2021-10-08 12:27:20 +0000 UTC
  URL:    https://github.com/usbarmory/armory-drive/releases/download/v2021.10.08/armory-drive.imx

Found proof bundle
  Tag:    v2021.10.08
  Author: andrejro
  Date:   2021-10-08 12:27:32 +0000 UTC
  URL:    https://github.com/usbarmory/armory-drive/releases/download/v2021.10.08/armory-drive.proofbundle

Found recovery signature
  Tag:    v2021.10.08
  Author: andrejro
  Date:   2021-10-08 12:27:32 +0000 UTC
  URL:    https://github.com/usbarmory/armory-drive/releases/download/v2021.10.08/armory-drive.sdp

Found SRK table hash
  Tag:    v2021.10.08
  Author: andrejro
  Date:   2021-10-08 12:27:33 +0000 UTC
  URL:    https://github.com/usbarmory/armory-drive/releases/download/v2021.10.08/armory-drive.srk

Downloaded verified release assets
Downloading manifest authentication key from usbarmory/armory-drive-log/keys/armory-drive.pub
Downloading transparency log authentication key from usbarmory/armory-drive-log/keys/armory-drive-log.pub

████████████████████████████████████████████████████████████████████

               ***  Armory Drive Programming Utility  ***
               ***           READ CAREFULLY           ***

This will provision F-Secure signed Armory Drive firmware on your USB armory. By
doing so, secure boot will be activated on the USB armory with permanent OTP
fusing of F-Secure public secure boot keys.

Fusing OTP's is an **irreversible** action that permanently fuses values on the
device. This means that your USB armory will be able to only execute F-Secure
signed Armory Drive firmware after programming is completed.

In other words your USB armory will stop acting as a generic purpose device and
will be converted to *exclusive use of F-Secure signed Armory Drive releases*.

████████████████████████████████████████████████████████████████████
```

WITH secure

Armory Drive - device verification of previous checkpoint

```
imx, csf, proof, err := extract(buf)

if err != nil {
        return fmt.Errorf("could not extract archive, %v", err)
}

if len(proof) > 0 {
        var pb *api.ProofBundle

        pb, err = verifyProof(imx, csf, proof, keyring.Conf.ProofBundle)

        if err != nil {
                err = fmt.Errorf("could not verify proof, %v", err)
                return
        }

        keyring.Conf.ProofBundle = pb
        keyring.Save()
}

// append HAB signature
imx = append(imx, csf...)

if err = usbarmory.MMC.WriteBlocks(2, imx); err != nil {
        return fmt.Errorf("could not write to MMC, %v", err)
}

log.Println("firmware update complete")
```

To ensure the firmware is valid, clients must verify that it is present in the transparency log. The **inclusion verification is performed by both the Armory Drive installer and the Armory Drive firmware** itself.

The installer can check the presence of the relevant hash directly against the published log.

The firmware, not having network access, is unable verify the inclusion directly with the log but instead does so via a *proof bundle* passed from the installer.

This *proof bundle* contains a new checkpoint covering the new firmware release, the firmware manifest, and all the leaf hashes from the log (i.e. HASH_1, HASH_2, ... , etc.).

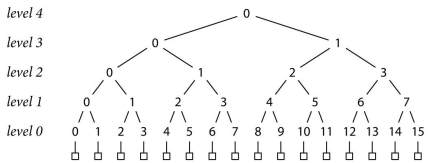
The presence of the leaf hashes allows the firmware to prove to itself that the checkpoint provided when the currently-running firmware was installed is consistent with (i.e. an ancestor of) the new one, ensuring that the tree history has not been compromised.

https://github.com/usbarmory/armory-drive/blob/master/internal/ota/ota.go

Witnesses

Transparency works if:

- There is a log at the centre which offers efficient cryptographically verifiable proofs of inclusion and append-only operation.
- Anyone preparing to rely on an artefact checks that it is present in the log before doing so (i.e. artefacts must be present in a log before anyone will trust them).
- There are one or more entities who are able to *verify* the correctness of artefacts in the log (i.e. "bad" artefacts will be spotted).
- Everyone listed above sees the same list of entries in the log.



A witness is an entity that:

- Verifies append-only operation of one or more logs, countersigning checkpoints if and only if the witness is convinced that a given checkpoint is consistent with all checkpoints previously issued by the same log.
- Makes these countersigned checkpoints publicly available.
- **Ensures there is only one view of the log.**

"In addition to verification done by the go command, third-party auditors can hold the checksum database accountable by iterating over the log looking for bad entries. **They can work together and gossip about the state of the tree as it grows to ensure that it remains uncompromised**, and we hope that the Go community will run them."
https://go.dev/blog/module-mirror-launch/

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 how to apply firmware transparency.

Device goals:

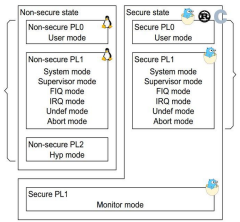
- Full transparency, open hardware and software.
- Reduced attack surface.
- Plug-and-go.

Combining our skills and projects:

USB armory

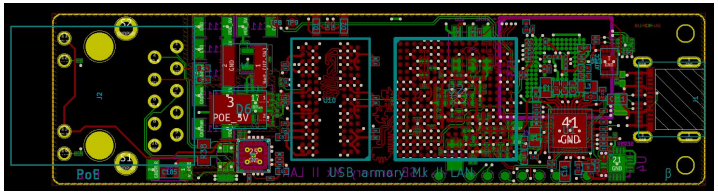
TamaGo

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.

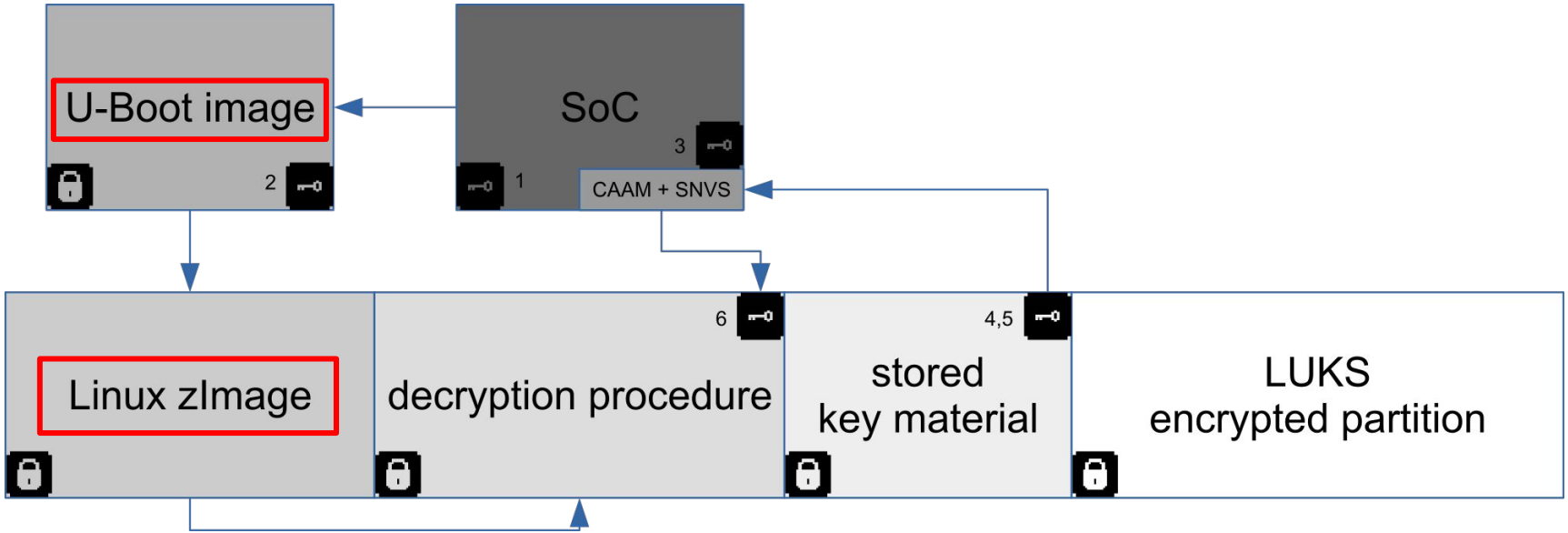
This means that less constrained hardware (e.g. SoCs in favor of MCUs) and higher level code are perfectly acceptable.

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**.

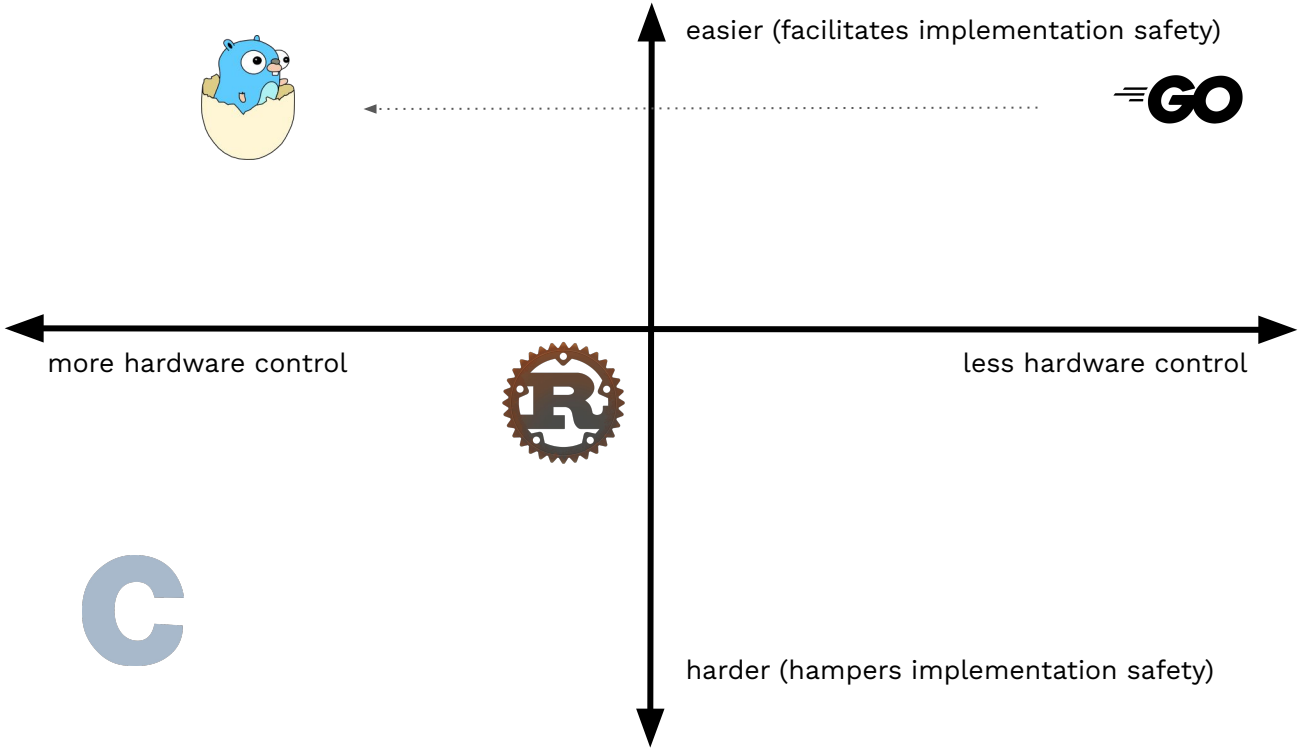
Reducing the attack surface



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.

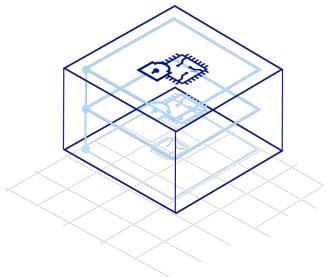
The focus is reducing the attack surface, carrying only strictly necessary code.

“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.

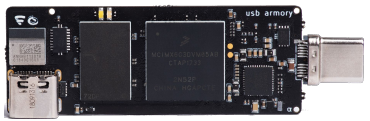
- A **minimally**[1] patched Go distribution to enable `GOOS=tamago` support, which provides freestanding execution on `GOARCH=arm` and `GOARCH=riscv64` bare metal.

- 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 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.

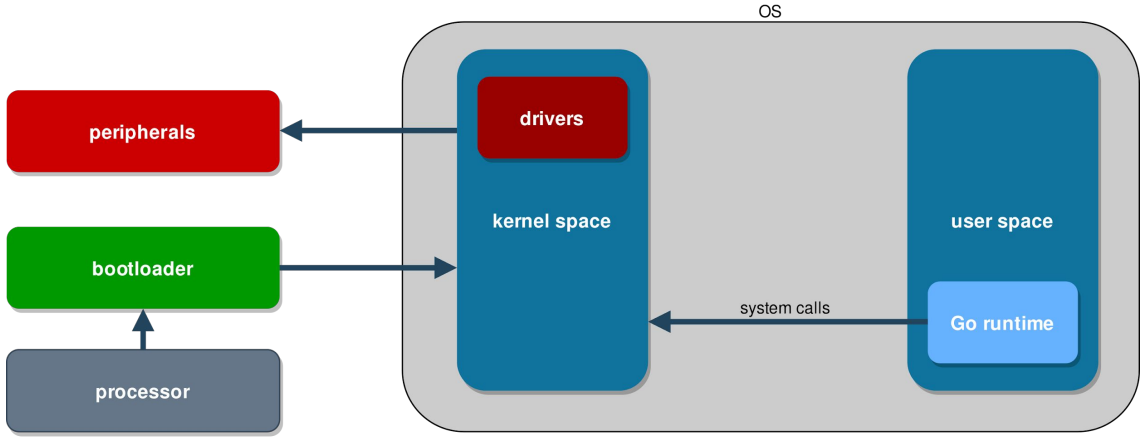
Applications are never required to use any `unsafe` function.

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

https://github.com/usbarmory/tamago

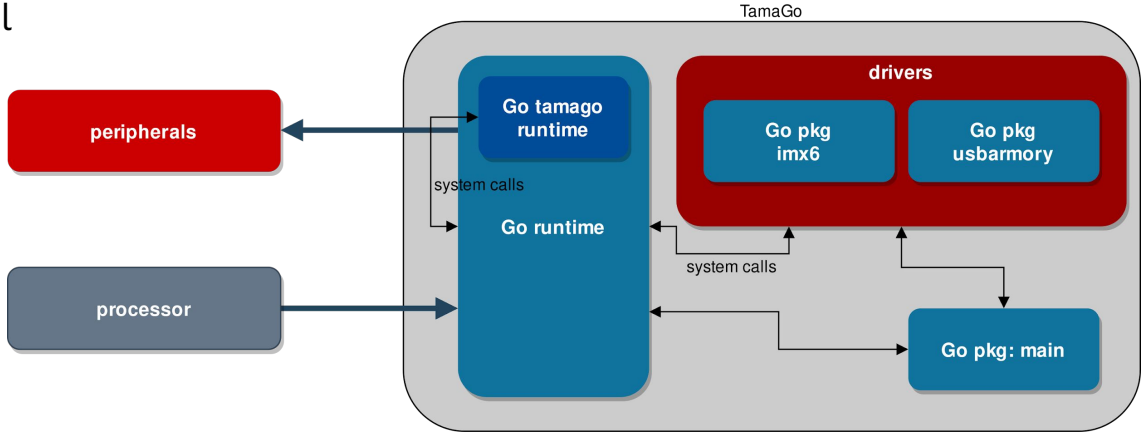
TamaGo

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!).

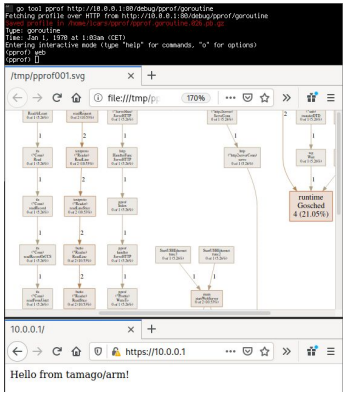
Firmware can be compiled just as easily on Linux, Windows, macOS.

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

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

Reducing the attack surface

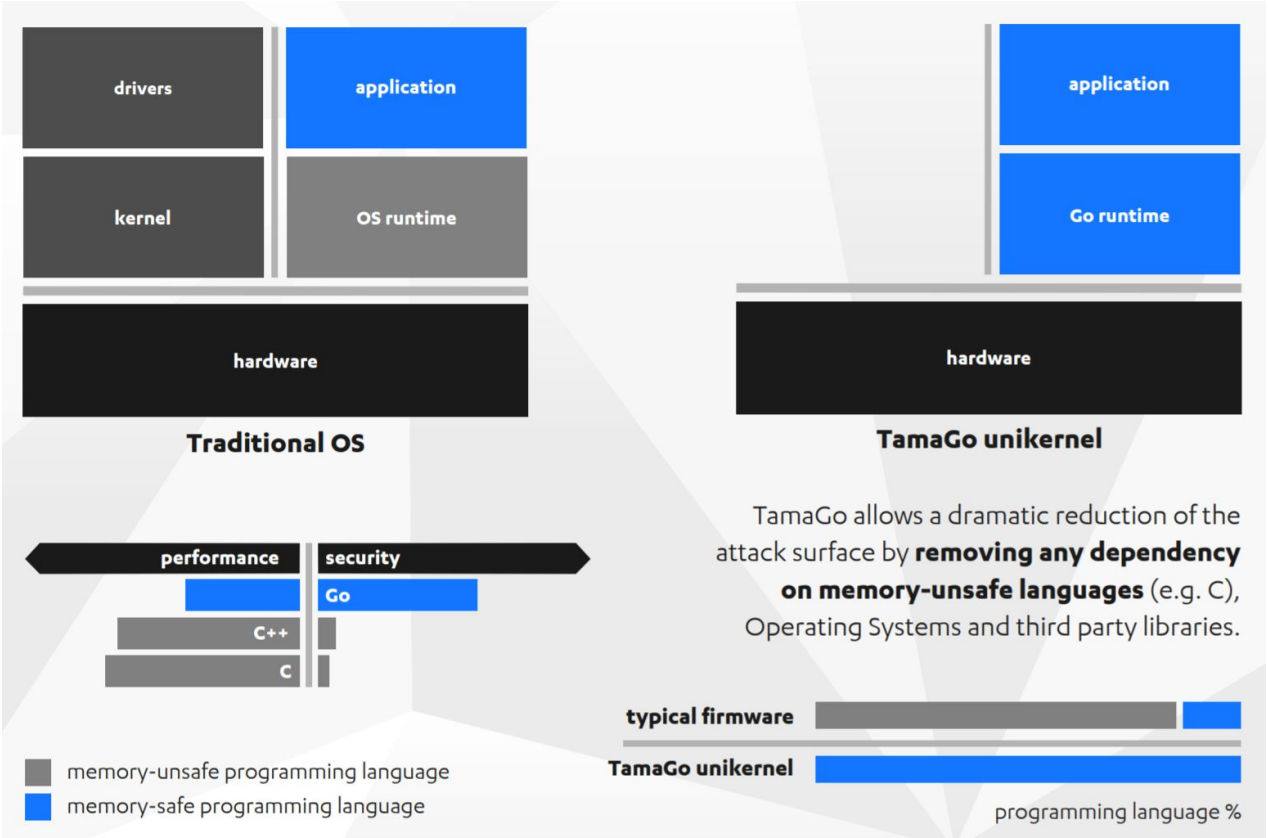
Block	LOCs	Driver support
ARM	900	CPU MMU, timer, exceptions, IRQ handling
BEE	130	OTF AES RAM encryption/decryption
CAAM	840	accel. AES/ECC/CMAC/SHA/TRNG, HUK derivation
DCP	450	accel. AES/SHA, HUK derivation
ENET	370	10/100-Mbps Ethernet driver, MII support
RNGB	80	True Random Number Generator
RPMB	230	Replay Protected Memory Block
SNVS	180	tamper proof sensors
USB	1200	USB 2.0 in device mode
USDHC	1100	eMMC (up to HS200 speed) / SD (up to SDR104 speed)
MK2	680	USB armory Mk II board support package



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 debugging.

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

Improving memory safety



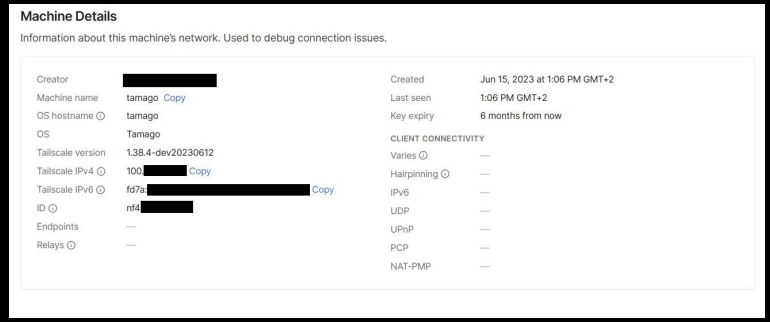
```
$ make qemu

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

```
> 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":"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx9b2efd","OS":"tamago","Package":"tsnet","Hostname":"tamago","GoArch":"a
rm","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
```




```
$ 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

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 (~400 LOC) to launch authenticated Linux kernel images on secure booted[1] USB armory boards, replacing U-Boot.

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

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

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

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

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

		imx6ul.ARM.DisableInterrupts()
		imx6ul.ARM.FlushDataCache()
		imx6ul.ARM.Disable()

		exec(image, params)
	})

	svc()
}
```

```
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

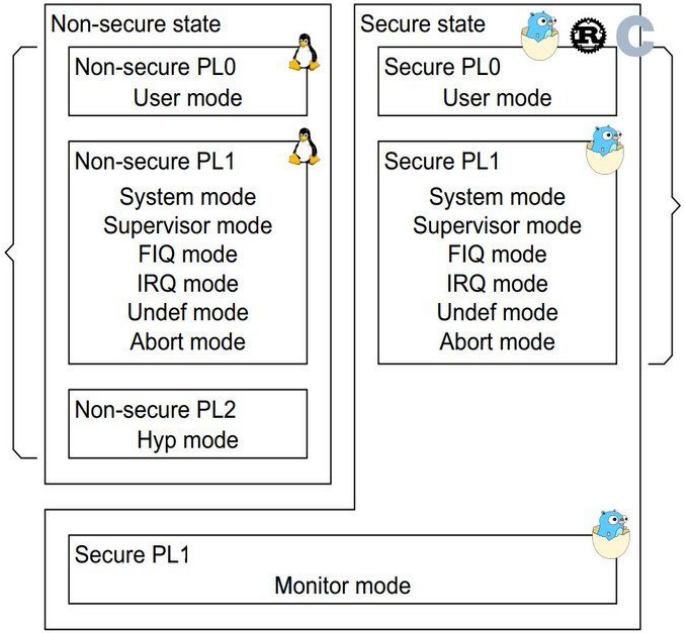
[1] https://github.com/usbarmory/usbarmory/wiki/Secure-boot-(Mk-II)

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/

```
> 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

  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

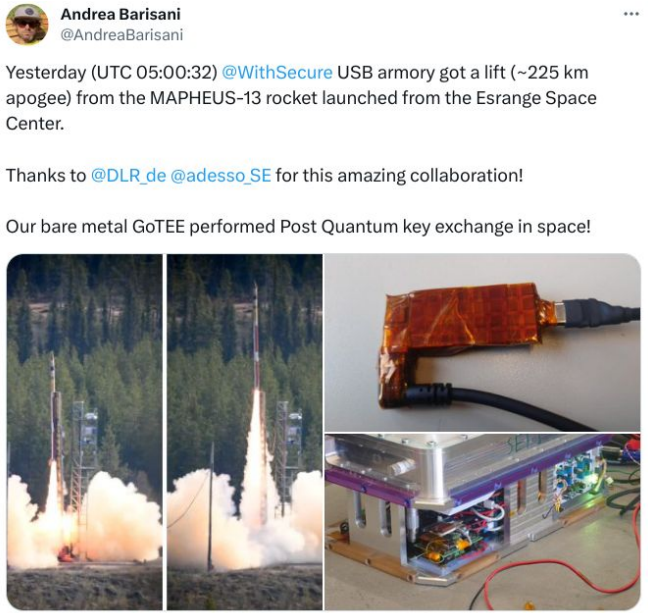
GoTEE in the wild: space!

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.

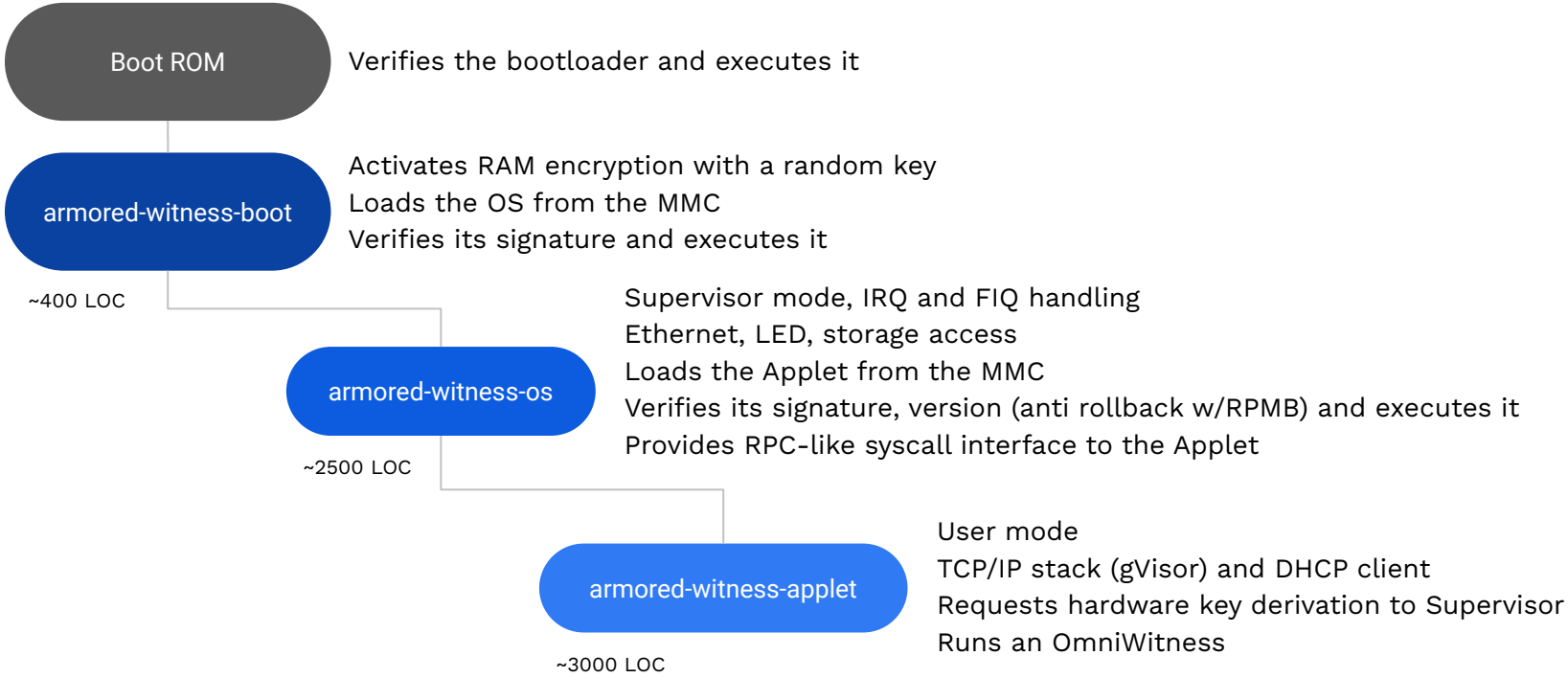
For this occasion TamaGo and GoTEE have been updated with full watchdog and interrupt support.



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

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

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 and WithSecure are 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

Code snippet - bootloader AES OTF RAM activation

```
func init() {
	// 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)
	}

	imx6ul.ARM.ConfigureMMU(
		bee.AliasRegion0,
		bee.AliasRegion1 + bee.AliasRegionSize,
		0,
		arm.TTE_CACHEABLE | arm.TTE_BUFFERABLE | arm.TTE_SECTION | arm.TTE_AP_001<<10,
	)
}
```

https://github.com/transparency-dev/armored-witness-boot/blob/main/mem_bee.go

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.

Code snippet - Trusted OS IRQ/syscall handling

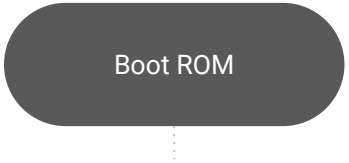
```go
func handler(ctx *monitor.ExecCtx) (err error) {
	switch ctx.ExceptionVector {
	case arm.FIQ:
		// service Ethernet IRQs for incoming packets
		return fiqHandler(ctx)
	case arm.SUPERVISOR:
		 // service system calls
		switch ctx.A0() {
		case syscall.SYS_WRITE:
			return bufferedStdoutLog(byte(ctx.A1()))
		case RX:
			return rxFromApplet(ctx)
		case TX:
			imx6ul.WDOG2.Service(watchdogTimeout)
			return txFromApplet(ctx)
		case FIQ:
			// re-activate Fast Interrupts
			bits.Clear(&ctx.SPSR, CPSR_FIQ)
		case FREQ:
			return imx6ul.SetARMFreq(uint32(ctx.A1()))
		default:
			// handle RPC
			return monitor.SecureHandler(ctx)
		}
	default:
		log.Fatalf("unhandled exception %x", ctx.ExceptionVector)
	}

	return
}
```

https://github.com/transparency-dev/armored-witness-os/blob/main/trusted_os/handler.go

```go
func eventHandler() {
	var handler rpc.Handler

	handler.G, handler.P = runtime.GetG()

	if err := syscall.Call("RPC.Register", handler, nil); err != nil {
		log.Fatalf("TA event handler registration error, %v", err)
	}

	n := 0
	out := make([]byte, enet.MTU)

	for {
		// To avoid losing interrupts, re-enabling must happen only
		// after we are sleeping.
		go syscall.Write(FIQ, nil, 0)

		// sleep indefinitely until woken up by runtime.WakeG
		time.Sleep(math.MaxInt64)

		// check for Ethernet RX event
		for n = rxFromEth(out); n > 0; n = rxFromEth(out) {
			rx(out[0:n])
		}
	}
}
```

https://github.com/transparency-dev/armored-witness-applet/blob/main/trusted_applet/handler.go

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

Putting it all together

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.

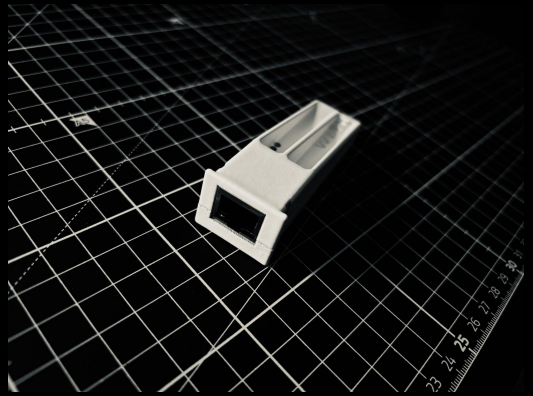
The benefit of this system comes through **removing trust from log operators** to behave honestly, and **placing some of that trust in the witnesses**.

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.

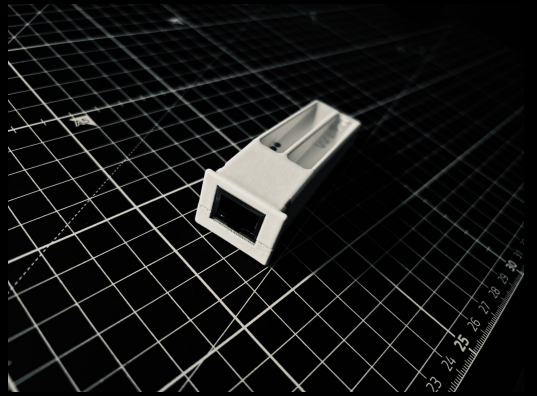
https://github.com/transparency-dev/armored-witness

```
$ 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==
```



WITH secure

```
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
```

WITH secure

```
$ 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!
```

WITH secure

```
$ 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
```

WITH secure

```
$ 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 🙏
--------------------------------------------------------------------------------
```

WITH secure

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

❤️ Thanks!

Andrej Rosano @ WithSecure

Al Cutter, Ryan Hurst, Martin Hutchinson
and the rest of Google TrustFabric Team

Join the transparency-dev Slack!

Andrea Barisani

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

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