Precursor

Secure Bootloader and Self-Provisioning

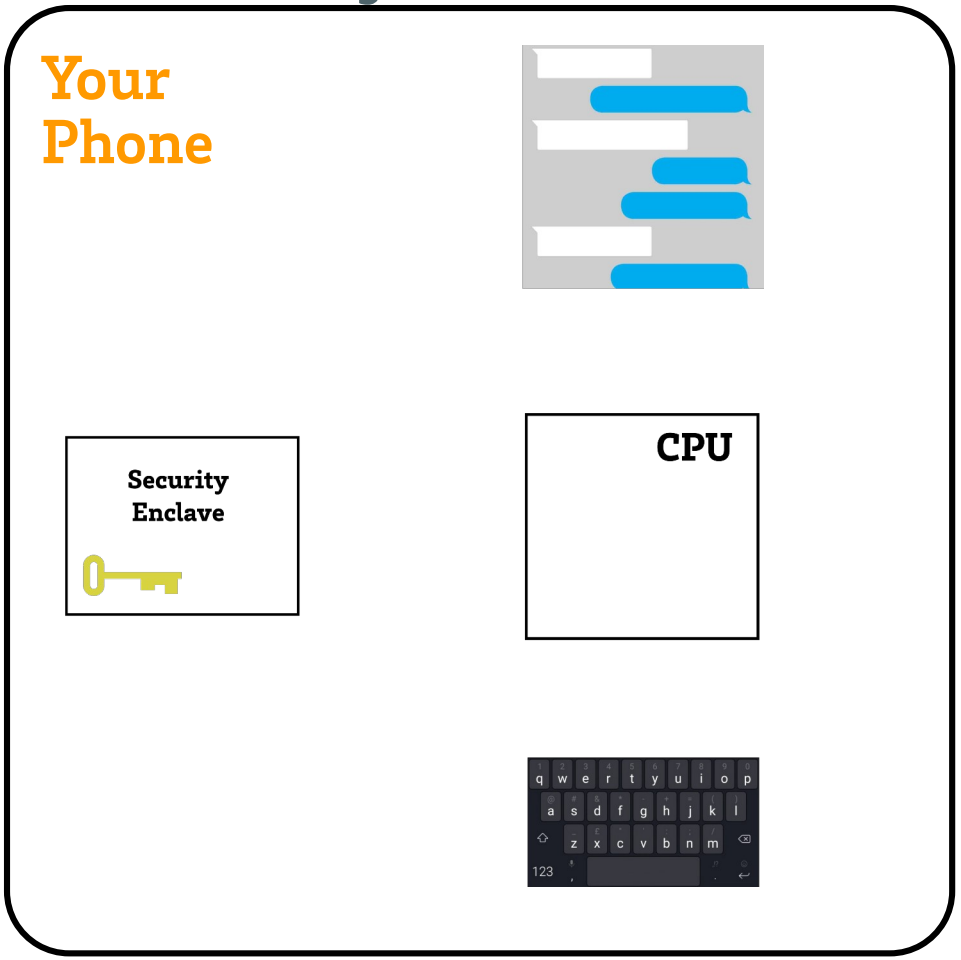
bunnie (@bunniestudios / twitter)

Silicon Salon - 2022

Precursor

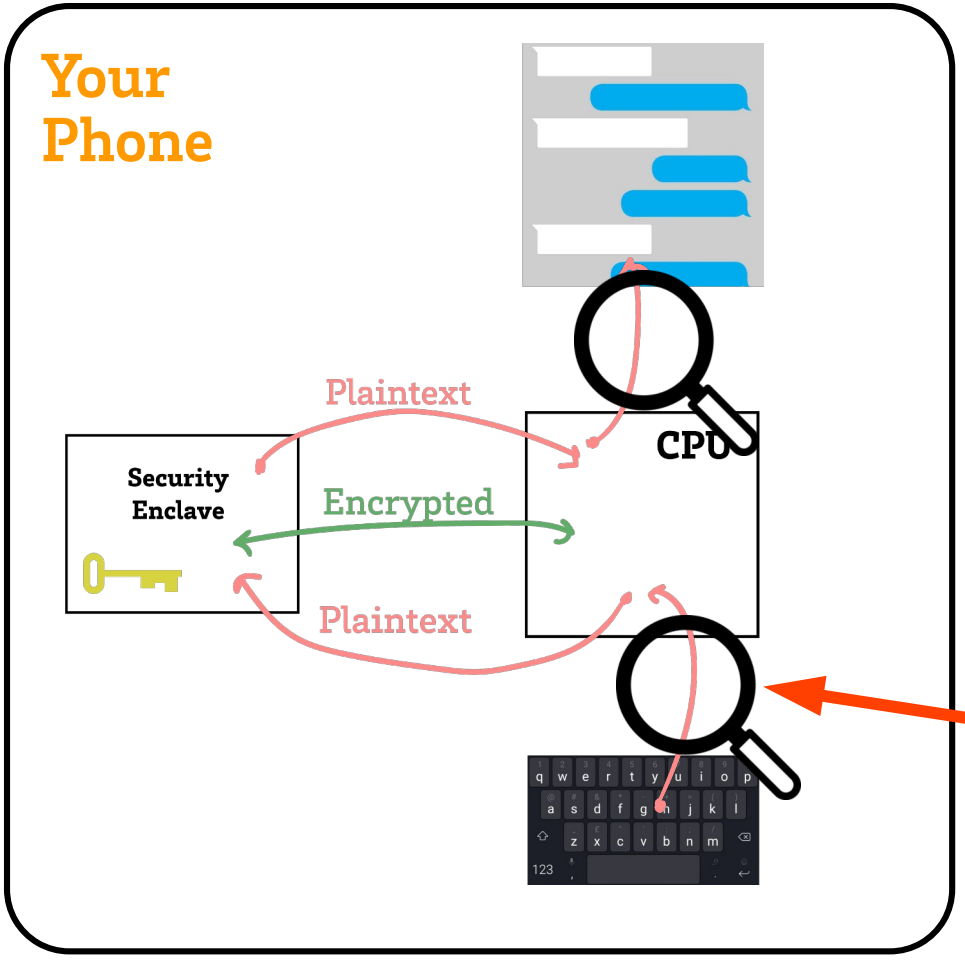
- What:
  - Mobile device
- Why:
  - Communication, authentication, wallet
- Who:
  - "At risk" end users: high-value targets either politically or financially; devs/enthusiasts
  - Global demographic (e.g., not just English-speaking)

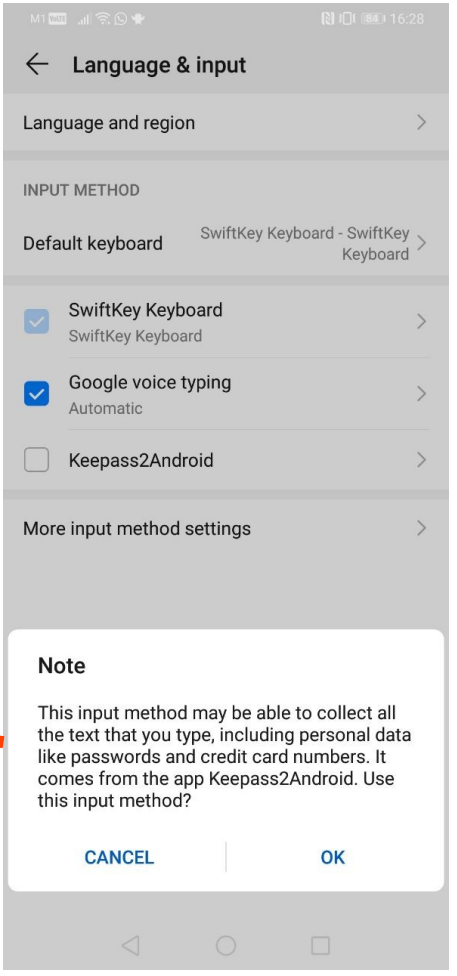
Why a Device, and Not Just a Chip?



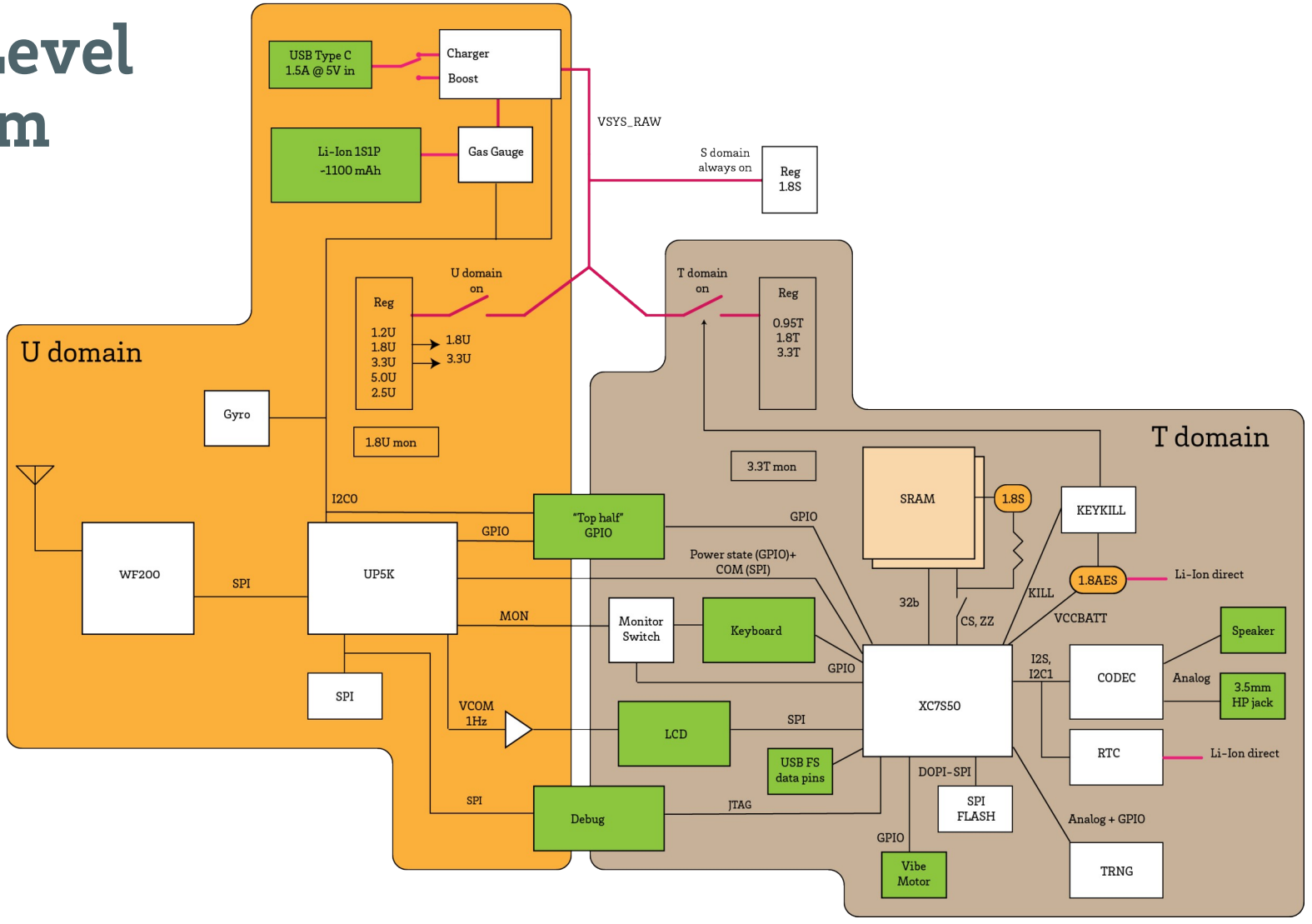
- Private keys are not your private matters
  - Screens can be scraped, keyboards can be logged

The Secure I/O Problem





System-Level Diagram

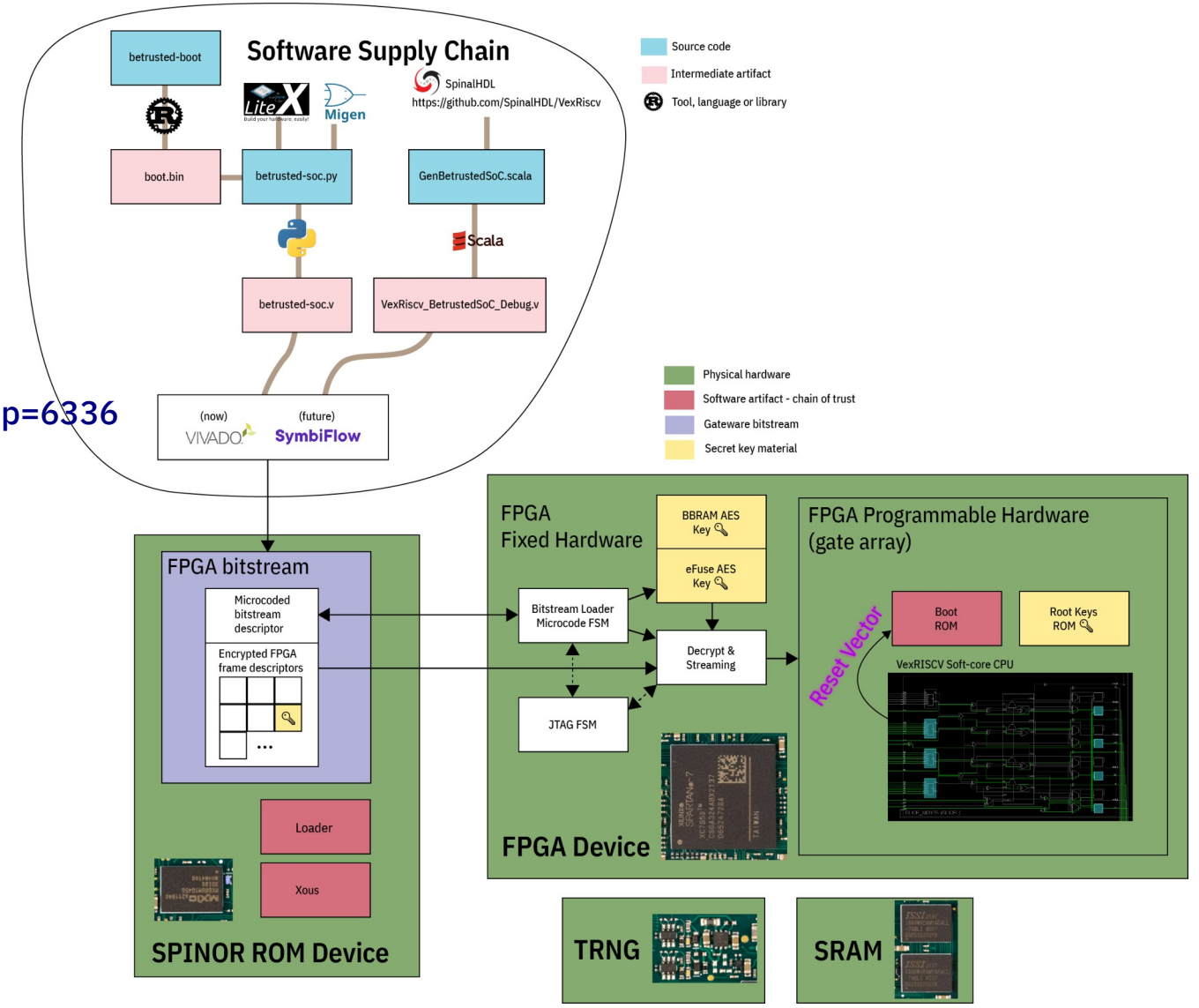


Long-Term Arc

- Use the FPGA-based system to:
  - Vet use cases
  - Develop apps
  - Test IP blocks
  - Hammer out kernel integration
  - Test things like the secure bootloader
- Eventually:
  - This gets taped out into an ASIC

Security is a System, not a Component

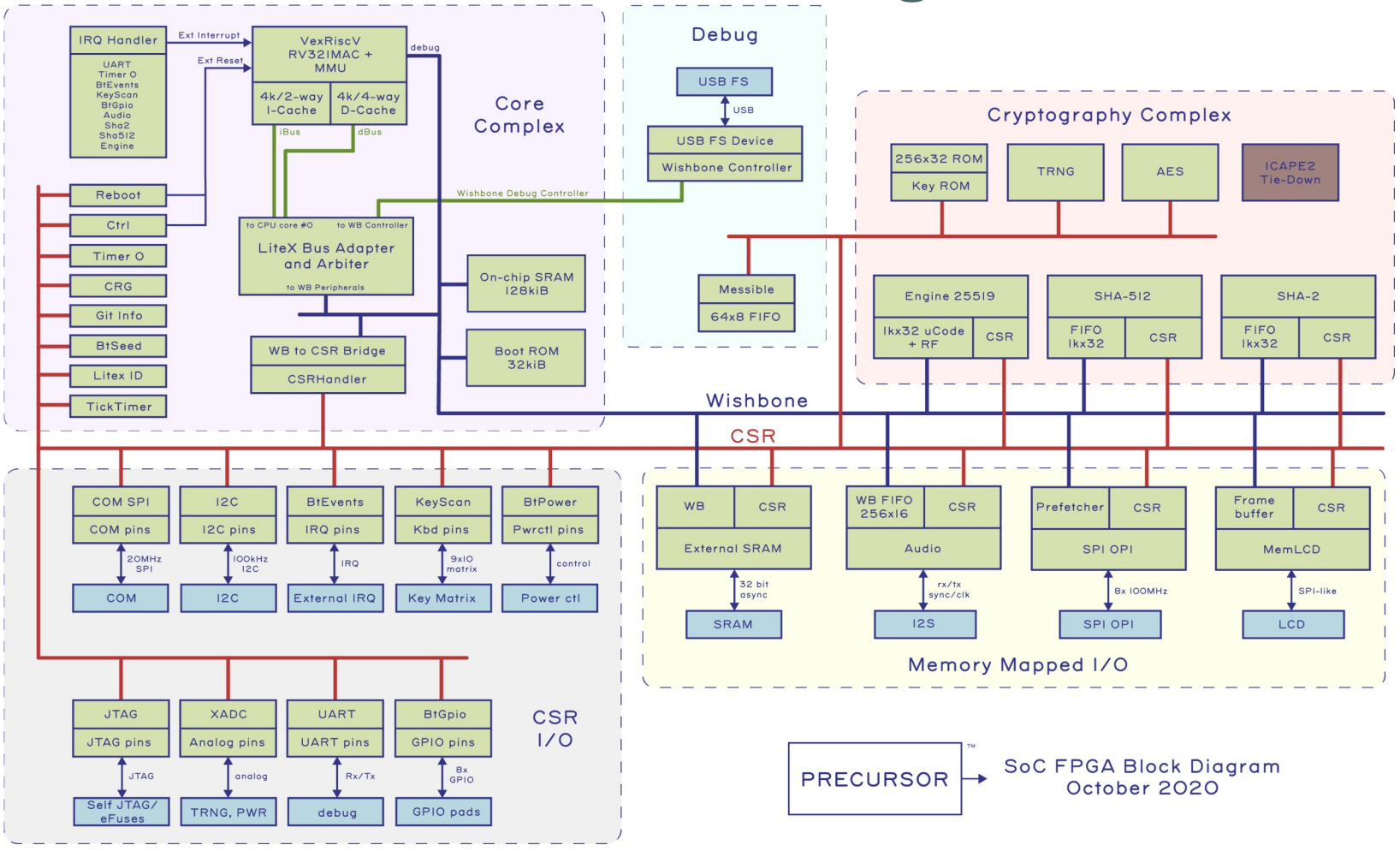
The software supply chain matters.
See the “full talk”:
https://www.bunniestudios.com/blog/?p=6336
“From Boot to Root in One Hour”



But, We Only Have a Few Minutes:

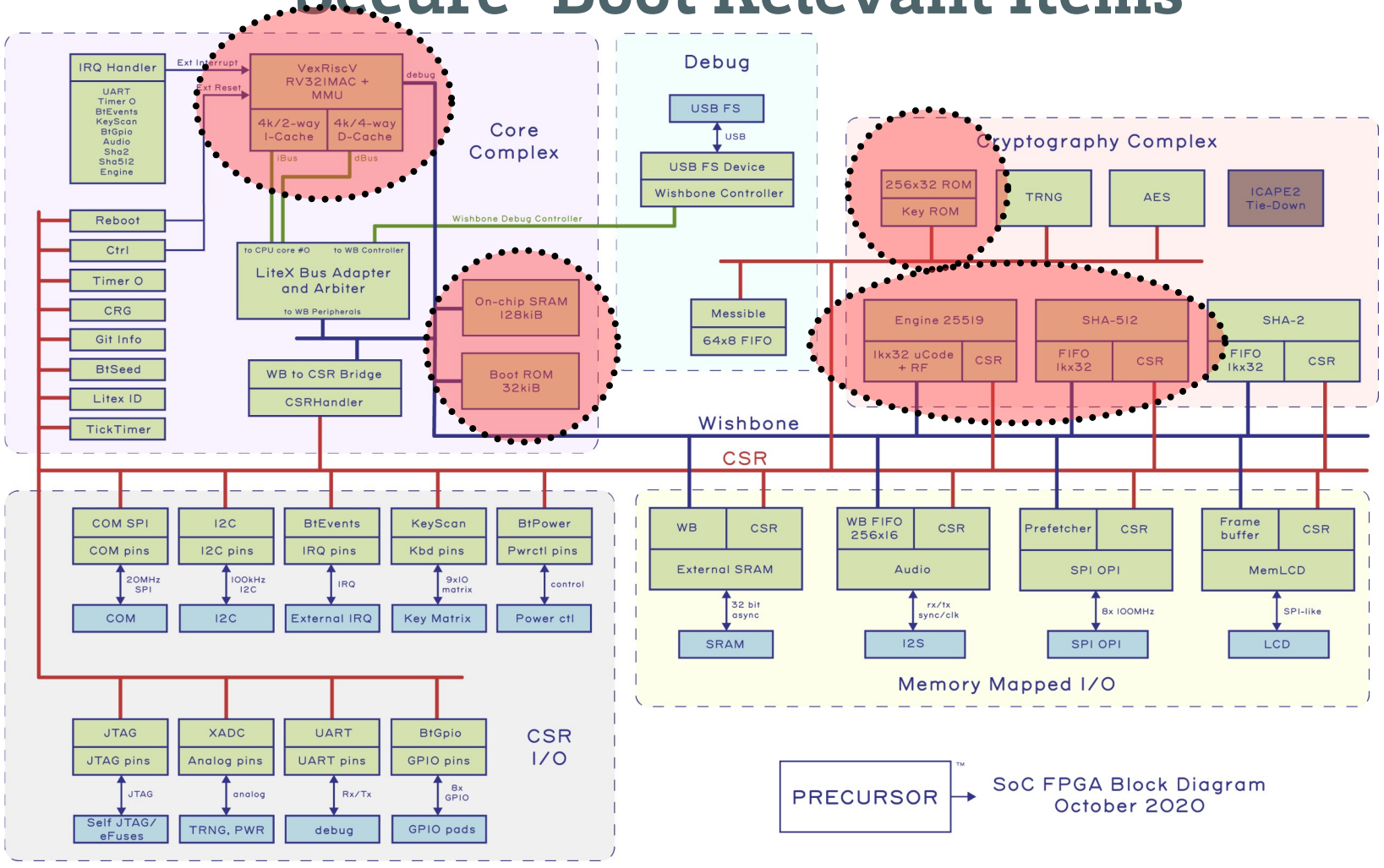
To the SoC!

SoC-Level Diagram



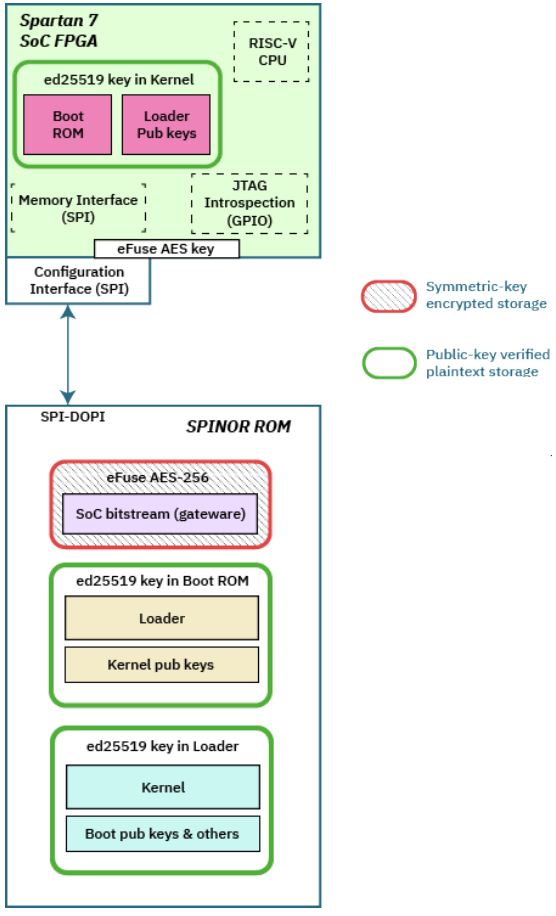
PRECURSOR™ → SoC FPGA Block Diagram
October 2020

Secure-Boot Relevant Items



PRECURSOR™ → SoC FPGA Block Diagram
October 2020

Layout of Artifacts



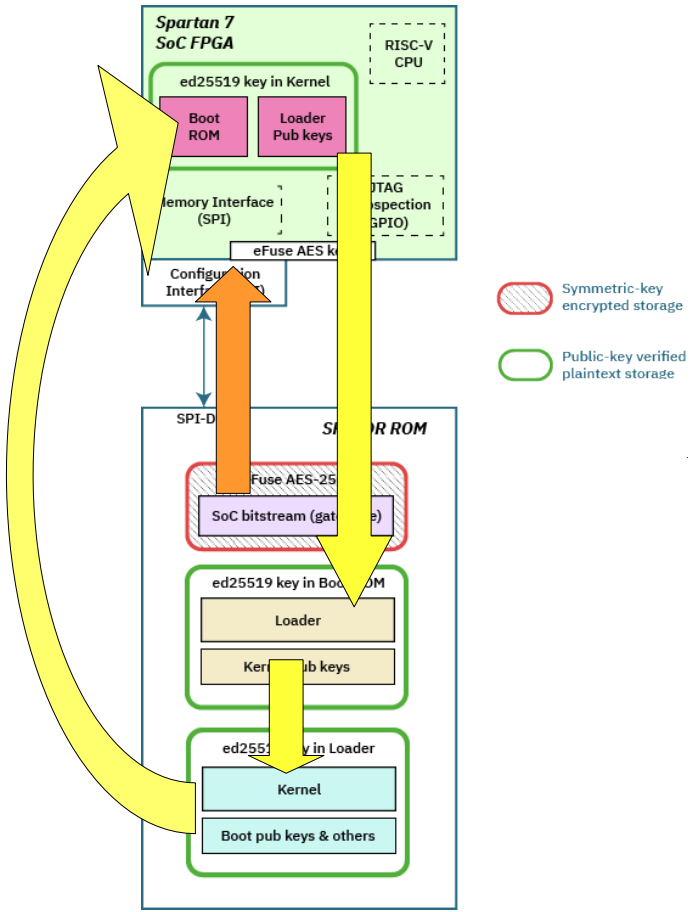
- Bootloader sigcheck code:
  https://github.com/betrusted-io/betrusted-soc/tree/main/boot/betrusted-boot/src

- Kernel loader sigcheck code:
  https://github.com/betrusted-io/xous-core/blob/main/loader/src/secboot.rs

bootloader

kernel

Layout of Artifacts



- Bootloader sigcheck code:
  https://github.com/betrusted-io/betrusted-soc/tree/main/boot/betrusted-boot/src
- Kernel loader sigcheck code:
  https://github.com/betrusted-io/xous-core/blob/main/loader/src/secboot.rs

bootloader

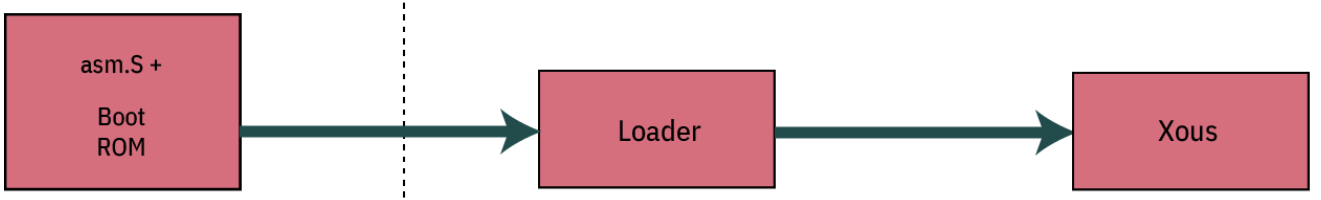
kernel

The Assembly Stub; then Pure Rust

```
4   _start:
5       // decorate boot area with a canary pattern
6       li          t1, 0xDEADC0DE
7       li          t0, 0x40FFE01C  // currently allowed stack extent - 8k - (7 words) - 7 words for kernel backup args
8       li          t2, 0x41000000
9   fillstack:
10      sw          t1, 0(t0)
11      addi        t0, t0, 4
12      bltu        t0, t2, fillstack
13
14      // flush any stale pages/cache in case of WDT reset by reading data out of the ROM
15      li          t0, 0x20500000
16      li          t2, 0x20508000
17  clearcache:
18      lw          t1, 0(t0)
19      addi        t0, t0, 4
20      bltu        t0, t2, clearcache
21
22      // Place the stack pointer at the end of RAM
23      li          t0, 0x40000000 // SRAM start
24      li          t1, 0x01000000 // SRAM length
25      add         sp, t0, t1
26
27      // Install a machine mode trap handler - just go back to the boot vector if we hit any issues
28      la          t0, _start
29      csrw        mtvec, t0
30
31      // Start Rust
32      j   rust_entry
```

Chain of Trust

Inside SoC



- Set up stack guard
- Clear cache
- Set stack pointer

- Check loader Ed25519 signature against known public keys
- Rudimentary graphics
- Consume < 8kiB of stack

Externally Loadable Code

- Check kernel Ed25519 signature against known public keys
  THEN
- Check for clean suspend marker & resume
  OR
- Clear RAM
- Map MiniElf segments to process RAM pages
- Copy .text (code); copy .data, zero .bss for each process
- Set up initial stack for the first thread in each process
- Set up Runtime Page Tracker (map physical RAM pages to process virtual pages)
- Map kernel superpage into userspaces
- Set all processes as runnable
- Set up PID 1 to run next
- Switch on MMU
- Jump to kernel init
- Rudimentary graphics
- Consume < 8kiB of stack

- Init memory manager
- Init system services
- Arch setup: interrupts & kernel TRNG
- Find runnable threads and run them

- Microkernel architecture
- Isolated processes provide services such as fs, net, io, etc.
- Root key server is one such process
- Applications are baked into a monolithic ROM image

A Brief Commentary on Threat Model Before Getting into Key ROM Layout

- Most ASIC Secure Boots:
  - Don't trust the user
  - Ultimately trust the manufacturer and the supply chain
  - Aim to enforce manufacturer or service provider-oriented policies upon the user
  - Aim to prevent users from running arbitrary code on their devices
- Precursor:
  - Doesn't trust the manufacturer
  - Doesn't trust the supply chain
  - Aims to empower users to control and protect their hardware
  - Aims to complicate tampering and remote exploit persistence

Key ROM Layout

offset	function	type	notes
0x00-0x07	eFuse key	AES256	this is necessary for bitstream updates. Erasing this makes the gateware immutable. Defaults to `0`
0x08-0x0F	self-signing privkey	ed25519 private key	Defaults to `0` (not used). Erased upon disable, regenerated from TRNG when enabled. Never disclosed to user.
0x10-0x17	self-signing pubkey	ed25519 public key	Defaults to `0` (not used). Derived from privkey.
0x18-0x1F	developer pubkey	ed25519 public key	`Well-known` key. When used, signatures are still validated, but UX is defaced with a strikethrough in the status bar.
0x20-0x27	third party pubkey	ed25519 public key	Reserved for users to provide a third-party public key for Boot ROM verification of loader packages.
0x28-0x2F	user root key	AES256	Root key for user secrets
0x30-0xF7	unallocated	TBD	Unallocated key store (space for ~25 additional 256-bit keys)
0xF8-0xFB	pepper [2]	128 bits pepper	128 bits of pepper, unique per device, used for password hashes
0xFC	anti-rollback [3]	u8.u8.u8.u8	min version code for FPGA gateware (maj.min.rev.ext) (unimplemented)
0xFD	anti-rollback	u8.u8.u8.u8	min version code for loader firmware (maj.min.rev.ext) (unimplemented)
0xFE	anti-rollback	x.x.x.u8	global anti rollback code
0xFF	config flags	config data	32 bits for config flags. See table below for config flags.

- Docs at https://github.com/betrusted-io/betrusted-wiki/wiki/Secure-Boot-and-KEYROM-Layout
- Size set by S7 LUTROM granularity
- Global anti-rollback by repeatedly hashing keys (255-code) times
- Private keys protected by password

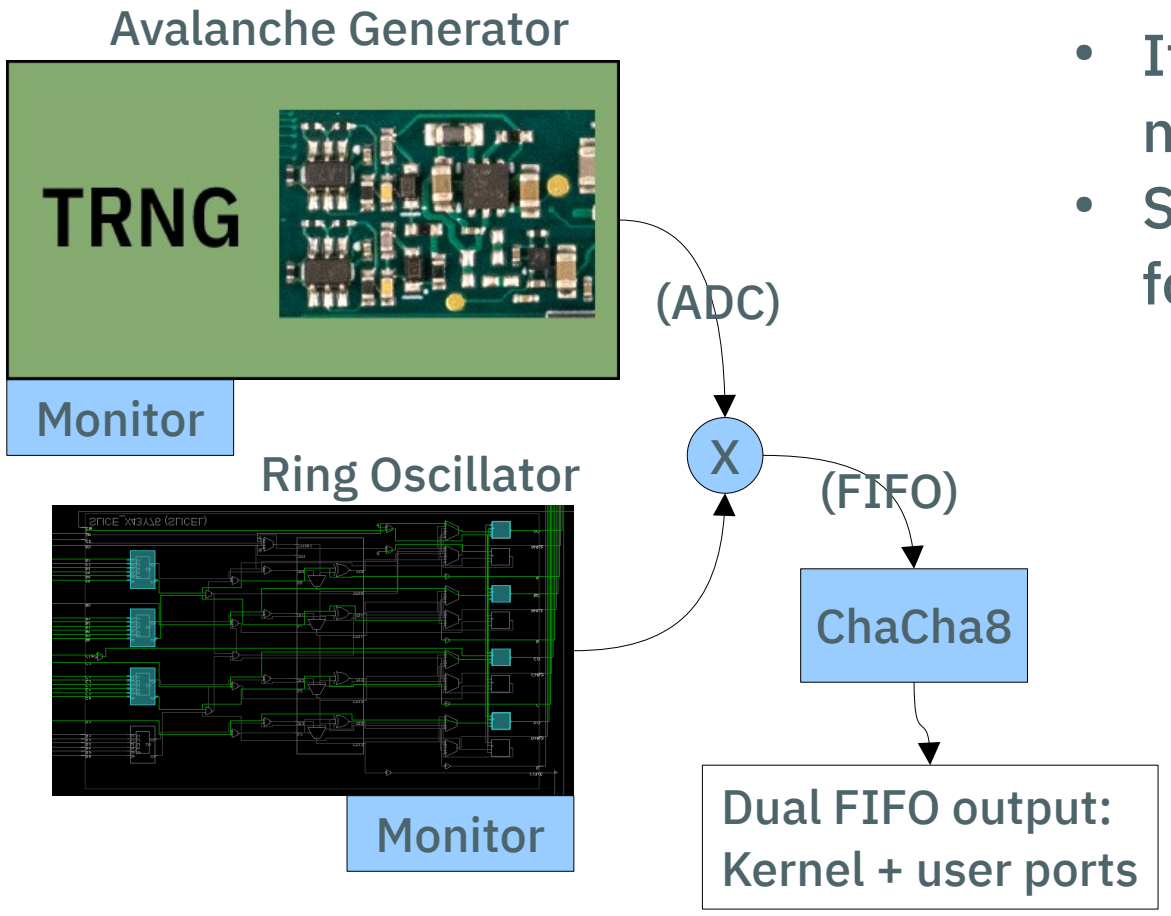
Rust: Pros/Cons for Bootloaders

- Pros:
  - Memory-safe language
  - Strongly typed
  - Good community support for cryptography (via cryptography.rs)

- Cons:
  - Larger binary size
    - Hardware crypto is a must to keep binary size down
    - 32kiB for:
      - HW init
      - Ed25519 drivers
      - Character graphics
      - Minimal key management
  - Steep learning curve
    - See https://www.bunniestudios.com/blog/?p=6375

Self-Provisioning

Step 1: A Good TRNG

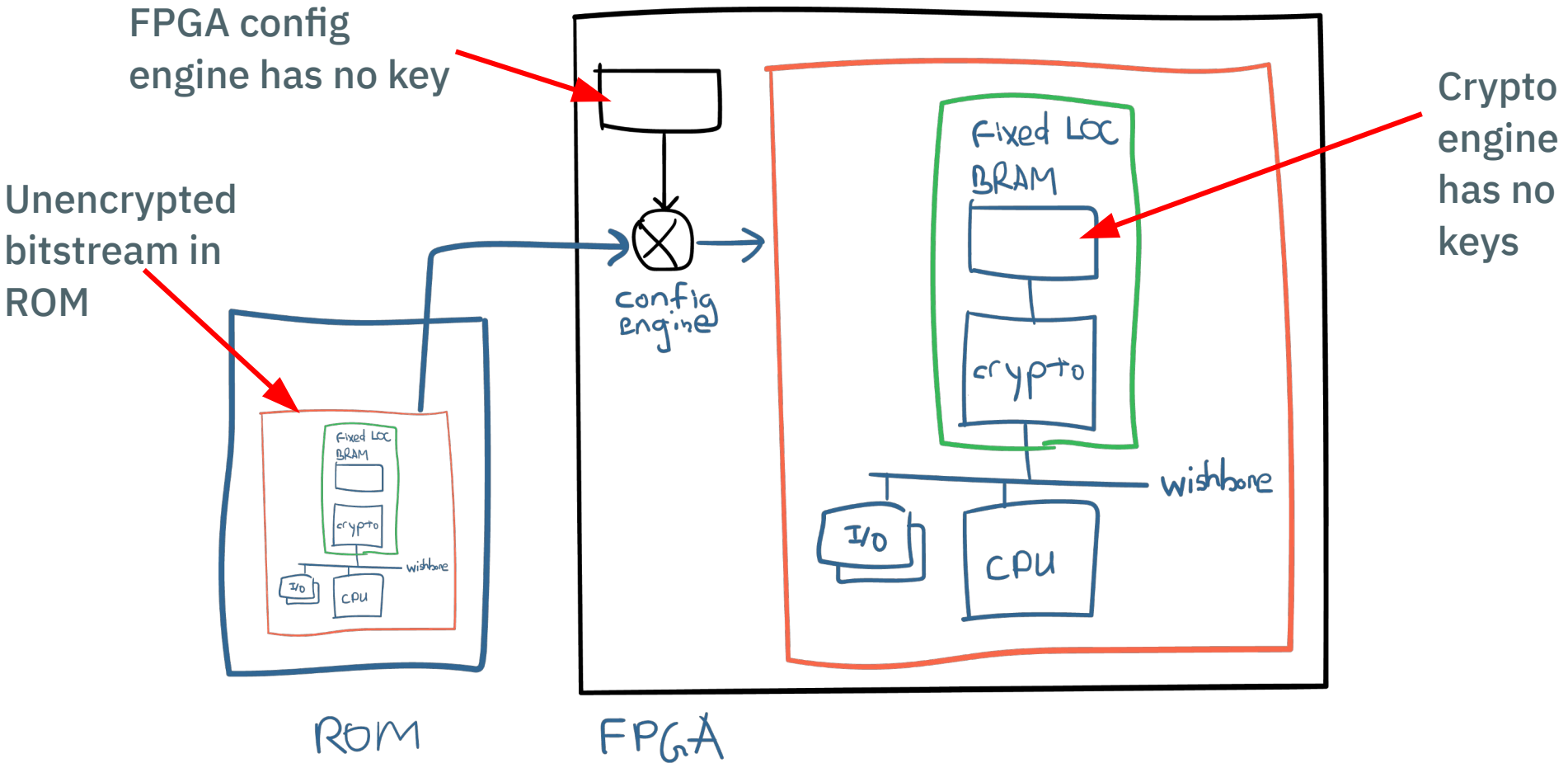


- If you don't get this right, nothing works.
- See https://betrusted.io/avalanche-noise for more

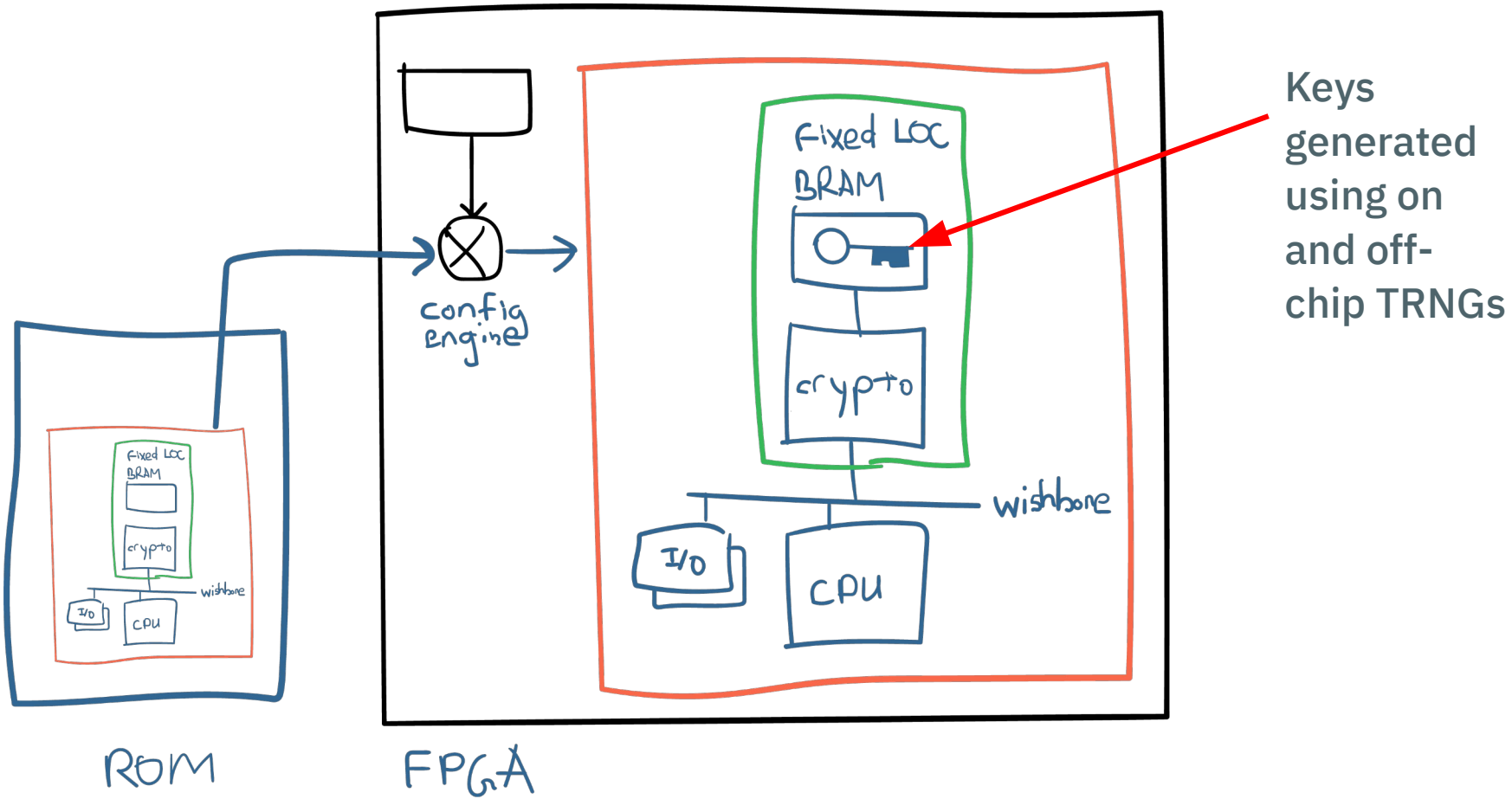
Step 2: Generate Your Keys

if you pick the right cipher, this is "easy"

Step 3: Save the Keys



Self-Provisioning and Sealing: Generate Keys

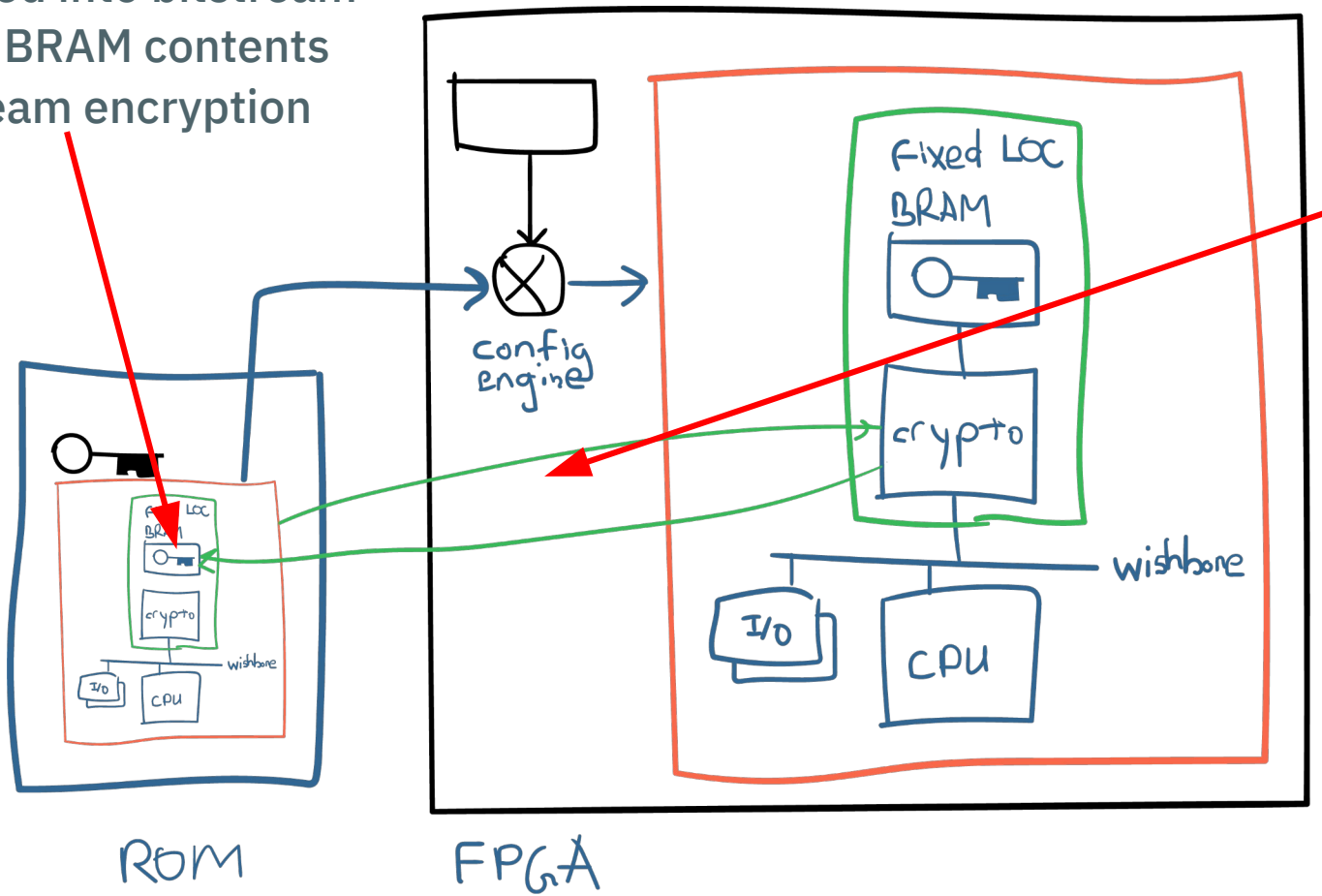


Keys generated using on and off-chip TRNGs

Self-Provisioning and Sealing: Encrypt Boot Image

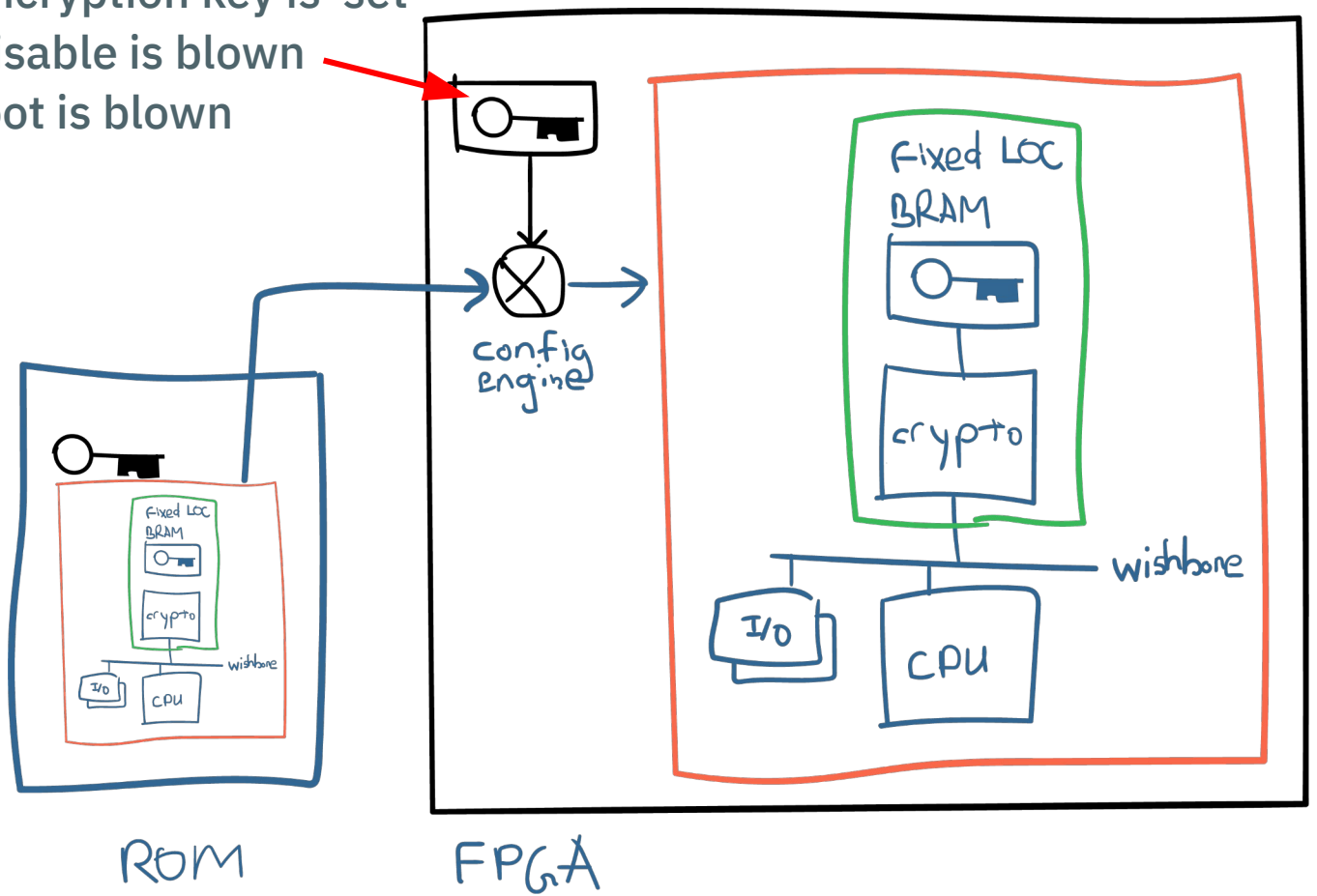
Keys are edited into bitstream by modifying BRAM contents during bitstream encryption

Crypto engine accesses ROM directly and encrypts bitstream



Self-Provisioning and Sealing: Seal FPGA

Bitstream encryption key is set
Readback disable is blown
AES-only boot is blown



HW info

Dev Chat

Q&A

@bunniestudios

Dev Chat:

https://matrix.to/#/#precursor.dev:matrix.org

https://precursor.dev for more device info

With thanks to:

nlnet FOUNDATION

NGI ZERO