ThoxOS

THOX.ai Research · Technical report v0.1.0 (preprint) · 2026-11-11

Preprint, not peer reviewed

ThoxOS™ RK: One Rust Core, Six Targets

A Measured, Provisioned Kernel Proof of Concept with a Reproducible Virtual Lab

A cross-repository review of THOX operating-system work, a small Rust kernel that boots the same portable core on five emulated machine classes and as a hosted process, a signed provisioning record bound to the image measurement, and a lab that a second team can rerun.

THOX.ai Research · Corresponding author: Tommy Xaypanya, CTO and Co-Founder (ORCID 0009-0000-4425-9565), THOX.ai LLC. Prepared with AI assistance; see Appendix C.

ABSTRACT

THOX ships local-first AI devices and has published an operating-system architecture, yet a review of 39 THOX repositories carried out for this report found no kernel that boots on a THOX device and no device provisioning format. The reviewed kernel of record, ThoxInux, is a fork of an independent Rust kernel that runs on one emulated x86-64 machine; the device images are specified as Linux; and a workspace named after the operating system does not compile as committed. This report contributes the missing bottom layer as a proof of concept, ThoxOS RK. One portable `no_std` Rust crate of 1,600 lines holds the whole boot policy: a SHA-256 measurement of the running image, an Ed25519-signed provisioning record of 168 bytes that binds a device identity and a deployment profile to that measurement, a lockdown state for every failure, a capability check on each service call, a kernel heap and a cooperative scheduler. Five bare-metal ports of 113 to 255 lines each (x86-64 by the PVH protocol, 64-bit Arm, RISC-V in supervisor and machine mode, and a Cortex-M microcontroller) and one hosted process run that same crate. In a lab of 46 boots under the QEMU emulator, 46 matched the expected verdict: every valid bundle provisioned, and all 36 boots with a missing, altered, wrongly signed or mismatched bundle, a patched image or an unkeyed image ended in lockdown. The measurement the guest computes equals the one the host tool predicts on every target, the images rebuild bit for bit on the lab host, and kernel text is 30.3 to 40.8 KiB. Nothing here ran on a THOX device or any physical board, there is no hardware root of trust, no user mode and no driver beyond a serial console; the report states these limits and lists the decisions that remain with the founders.

CONTENTS

Executive summary
1. Introduction
2. What exists today: the cross-repository review
3. Background and related work
4. System and threat model
5. Design
6. Implementation
7. Evaluation
8. Discussion and limitations
9. Conclusion
10. Roadmap and decisions for the founders
Appendix A. Reproducibility
Appendix B. Console and wire reference
Appendix C. Authorship and disclosure
References

Executive summary

ThoxOS™ is the name THOX gives to the operating system of its local-first AI products. This report does two things. It records what a parallel review of THOX's operating-system, kernel, firmware, provisioning and research repositories found on 2026-11-11. It then builds and measures the layer the review found missing: a kernel image that knows which image it is, checks that it has been provisioned for the device it runs on, and refuses to do more than talk on its console when that check fails.

46/46	36/36	6	30.3 to 40.8
lab boots matched their expected verdict	missing, altered or mismatched provisioning ended in lockdown	targets run one shared core of 1,600 lines	KiB of kernel text per bare-metal target
EMULATED, MEASURED	EMULATED, MEASURED	BUILT AND BOOTED	HOST, MEASURED

> WHAT THIS MEANS FOR THE PRODUCT
>
> The result is a working method, not a shipping operating system. Three things are ready to reuse today: the portable core crate, the 168-byte provisioning record with its fleet tool, and the lab. Three things are not decided and cannot be decided by engineering alone: which kernel ships on THOX devices, which processor each device carries, and who holds the fleet signing key. Section 10 lists them with options.

The evidence class of every result is stated where it appears. **Emulated** means a QEMU machine model executed by the TCG interpreter on the lab host (Intel(R) Xeon(R) Processor @ 2.10GHz, QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18)). **Host** means a native process on that machine. No result in this report is a measurement of a THOX device.

1 Introduction

A device that runs a user's models and holds a user's data makes a promise about what code it runs. Keeping that promise starts below the application: the first code that executes has to establish which image it is, whether that image was authorised for this unit, and what the unit is allowed to do. THOX's published architecture describes the layers above that point in detail [1]. It does not yet have the point itself.

The request behind this report was broad: review the latest operating-system work across THOX repositories, study three public bodies of Rust operating-system work (the *Writing an OS in Rust* series [2], Redox OS [3] and a community comparison of Rust operating systems [4]), and move toward a Rust firmware, kernel and operating system for four devices and for desktop, server, virtual-machine and container hosts. We narrowed it to a question that can be answered with evidence in one pass: *can one small, portable Rust core carry the same boot-time trust decision across every machine class THOX names, and can a second team reproduce the proof?*

This report makes five contributions.

1. **A cross-repository review** with provenance: 39 repositories read by six teams, with a seventh for external sources, test suites run where they could run, contradictions between repositories recorded side by side rather than merged (Section 2).
2. **A portable kernel core.** One `no_std` crate of 1,600 lines with no architecture-specific code, unit tested on the host, that every target links unchanged (Section 5).
3. **A provisioning record bound to the image.** A fixed 168-byte, Ed25519-signed record that names a device, a deployment profile and the SHA-256 measurement of exactly one image, with a fleet-side tool that predicts that measurement from the ELF file (Sections 5.3 and 5.4).
4. **Six targets.** Five bare-metal ports of 113 to 255 lines each and a hosted process, covering the PVH protocol used by lightweight virtual machine monitors, 64-bit Arm, two RISC-V privilege configurations and a Cortex-M microcontroller (Section 6).
5. **A reproducible lab.** One command builds every image, provisions it, boots 46 scenarios and writes hashed evidence; the report and its companion page are generated from that evidence (Section 7 and Appendix A).

We also report what the lab caught in our own work. The first x86-64 image verified its bundle signature and still locked down, because the processor had modified a byte inside the range the image measures (Section 6.3). The check did its job against its authors, which is the behaviour a boot measurement is for.

2 What exists today: the cross-repository review

Six teams read 39 repositories under a read-only rule, five of them in parallel and one in a later pass, ran the test suites that could run offline, and wrote findings with file and line provenance. A seventh read the external sources and verified their licences. An eighth consolidated the findings. The full findings, the consolidated review, a list of safe fixes and a tracker of work items ship with the source under `rk/docs/review/`. Table 1 gives the headline per area.

Area	What the name suggests	What the review found	Evidence class
`thoxos` workspace	A Rust kernel with drivers, boot stage and update	Linux userspace crates for one vendor development kit. No `no_std` code, no scheduler or memory manager. The workspace does not compile as committed: 5 of 10 crates fail, and 11 of 100 declared tests can run.	Built and tested by Team A
ThoxInux	The THOX kernel	A four-patch fork of an independent Rust kernel with a Linux system-call personality [5]. x86-64 only, exercised on one emulated machine with one virtual processor. No physical hardware run exists. Guard and unit suites pass; the kernel build and lifecycle suite were not rerun.	Read; suites run by Team B
Device images	ThoxOS on ThoxAir, ThoxMini, ThoxClip	A specification for Linux with A/B updates: class floors for memory and storage, no kernel tree, no board named, update configuration never rendered or executed.	Read by Team B

Area	What the name suggests	What the review found	Evidence class
ThoxKey	A device that runs firmware	Three different things share the name: a storage product with no processor, a single-board proof of concept running a vendor Linux kernel, and a microcontroller display variant held in a repository outside this review.	Read; suites run by Team C
Editions	ThoxOS for desktop, mobile, web	At least 18 distinct artefacts carry the name: Linux remasters on five bases, web and desktop applications, a theme, and a language model titled a kernel.	Read by Team D
Provisioning	A way to bind an image to a unit	No device provisioning format exists. No production signing key exists. The entitlement token uses a shared secret, which the published report already shows to be unfit for on-device verification [1].	Read by Teams B and C
Research catalog	One kernel story	Three: a Rust-first kernel fork in one report, a Linux fork line in a second, Arm system-on-chip images in a third.	Read by Team E

Table 1. Headline findings of the review. Each row is traced to file and line in the findings shipped with the source. Where repositories disagree, the review shows both statements and names who decides.

Two findings shaped the design. First, hardware is unresolved: for three devices the record holds category floors and candidate boards that disagree between documents, and for the fourth the shipping product has no processor to run firmware on. A kernel proof of concept therefore cannot honestly be *for* a THOX device today. It can be for the machine *classes* the devices will fall into, and it can treat a device as a provisioned identity rather than a board. Second, the missing piece is not a scheduler or a file system, of which good open designs exist (Section 3). It is the trust decision at boot, which nothing in the reviewed repositories implements below a running Linux system.

> POSITION RELATIVE TO THOXINUX
>
> ThoxOS RK does not replace ThoxInux, and this report does not recommend that it should. ThoxInux answers a different question: how much Linux userspace can run on a Rust kernel. RK answers how small a portable trusted base can be and still carry provisioning across machine classes. The core crate is written so that either kernel, a Linux userspace daemon, or a microcontroller firmware can link it. Which kernel ships is a founder decision (Section 10).

3 Background and related work

3.1 The three sources named in the brief

Writing an OS in Rust. Oppermann's series builds an x86-64 kernel from a freestanding binary: no standard library, a custom entry point, a panic handler, a bootloader crate that sets up long mode, and a text console [2]. Its code is offered under MIT or Apache-2.0 terms; its prose carries a non-commercial licence and is not reused here. RK follows the same freestanding-binary discipline and departs from the series in three ways: it boots without a bootloader crate, by implementing the entry protocols directly; it targets five architectures, not one; and it builds on the stable compiler with built-in targets. RK contains no code from the series.

Redox OS. Redox is a general-purpose microkernel operating system in Rust with user-space drivers, its own C library and file system, released under the MIT licence [3]. It is the most complete open Rust operating system we reviewed and the natural reference for a long-term general-purpose direction. Adopting it means adopting its design as a whole; RK takes no code from it.

The comparison list. The community comparison catalogues several dozen Rust operating systems by kernel type, architecture and licence [4]. We checked the licence of each candidate against its own repository rather than the table, and found the table wrong in two places; the verified list is in `rk/docs/STACK.md`. The practical outcome: permissively licensed components are available for every layer THOX needs, and the copyleft kernels on the list are avoidable.

3.2 Rust operating systems and what they isolate

Tock isolates untrusted applications on microcontrollers with a memory protection unit and uses Rust's type system to isolate kernel components at no run-time cost [6]; its authors later reported the path from that research system to deployment as a root of trust in shipped computers [7], and a tiered trust model for embedded systems built on it [8]. Hubris takes the opposite position on dynamism: a fixed set of tasks declared at build time, with memory protection between them and no run-time task creation [9]. Theseus structures a whole system as many small, safely reloadable components in one address space and one privilege level [10]. RedLeaf builds fault isolation between kernel subsystems from language safety alone [11]. seL4 remains the reference for a formally verified kernel with capability-based access control [12]. Each of these systems isolates more than RK does. RK isolates nothing at run time: it enforces a capability check at a call boundary inside one address space (Section 5.5). Its contribution is narrower and sits earlier in the boot.

3.3 Measured boot, device identity and capabilities

Two ideas in this report are long established and we claim neither: that each boot stage should check the next before running it, and that a system should record a measurement of what it loaded so that the measurement can be judged. The Device Identifier Composition Engine specification derives a device identity from a hardware secret and the measurement of the first mutable code [13]. RK implements the measurement and the binding in software and leaves the hardware anchor as stated future work: without an immutable first stage, an attacker who can replace the entire image can also replace the key it trusts. We state that boundary in the threat model rather than imply more. Signatures are Ed25519 [14] and the measurement is SHA-256 [15]. The capability vocabulary follows the usual formulation, in which holding a token is the authority to act, as in [12].

3.4 Boot protocols used

On x86-64 RK is entered through the PVH direct-boot interface, in which the loader reads a 32-bit entry address from an ELF note and starts the image in protected mode with paging off [16]. The QEMU microvm machine, a minimal machine type modelled on lightweight virtual machine monitors, accepts such an image directly [17]. PVH is a vendor-neutral direct-boot interface, which is why we chose it for the virtual-machine class. We ran RK only on QEMU. On RISC-V the supervisor-mode port runs under OpenSBI [18] and powers off through the system reset extension of the supervisor binary

interface [19], while the machine-mode port runs with nothing beneath it. On every emulated machine the lab places the provisioning record in guest memory or flash with the generic loader device [20].

System	Primary goal	Isolation at run time	Boot-time image binding	What RK shares	What RK lacks
Tock [6]	Untrusted apps on microcontrollers	MPU processes, type-safe capsules	Outside its kernel scope	Small footprint, `no_std`	Processes, MPU, drivers
Theseus [10]	Live evolution, state management	Language level, single address space	No	Single address space	Component reloading
RedLeaf [11]	Fault isolation by language	Language domains	No	Language safety as the base	Domains, recovery
Hubris [9]	Fixed-task embedded kernel	MPU tasks, static configuration	Outside its kernel scope	Static configuration, small size	Tasks, MPU, drivers
seL4 [12]	Verified microkernel	Capabilities, MMU, proofs	Platform specific	Capability vocabulary	Verification, everything else
Redox [3]	General-purpose OS	User mode, MMU, user drivers	No	Rust, permissive licence	An operating system
ThoxOS RK	Portable trusted base for provisioning	**None** (call-boundary policy)	**Yes**, signed record bound to a measurement		

Table 2. Positioning. RK is the smallest system in the table and the only one whose stated purpose is the boot-time binding. It is not a substitute for any of the others; it is a layer they do not provide.

4 System and threat model

System. A fleet operator holds a root signing key offline. Each unit runs one kernel image and has one provisioning slot, a small region of flash or memory at an address fixed per board. A unit is provisioned when the slot holds a record signed by the root key that names the measurement of the image actually running.

Adversary. The adversary can read and write the provisioning slot, can replace or patch the kernel image without the root key, can replay a record made for another image or another unit class, and can supply an image built without the fleet key. The adversary cannot forge an Ed25519 signature or find a SHA-256 collision. **Out of scope**, and stated as a limit rather than a defence: an adversary who replaces the whole image *including* the embedded root public key, physical attacks, fault injection, side channels, rollback to an older correctly signed image, and any attack after boot.

Requirements.

1. **R1 Portability.** The trust decision is one body of code with no architecture-specific branch.
2. **R2 Binding.** A record is accepted only if the root key signed it and its measurement equals that of the running image.

3. **R3 Fail closed.** Every other outcome, including an unreadable slot and a failed self-test, yields lockdown.
4. **R4 Least authority.** In lockdown the system holds the console capability only; when provisioned it holds the set its profile grants and no more.
5. **R5 Predictability.** The fleet tool computes from the image file the same measurement the image computes of itself.
6. **R6 Small footprint.** The image fits a microcontroller with 256 KiB of flash and 64 KiB of memory.
7. **R7 Reproducibility.** A second team can rebuild the images and rerun every claim with one command.
8. **R8 Minimal dependencies.** Third-party code is limited to what can be audited in an afternoon.

5 Design

5.1 Architecture

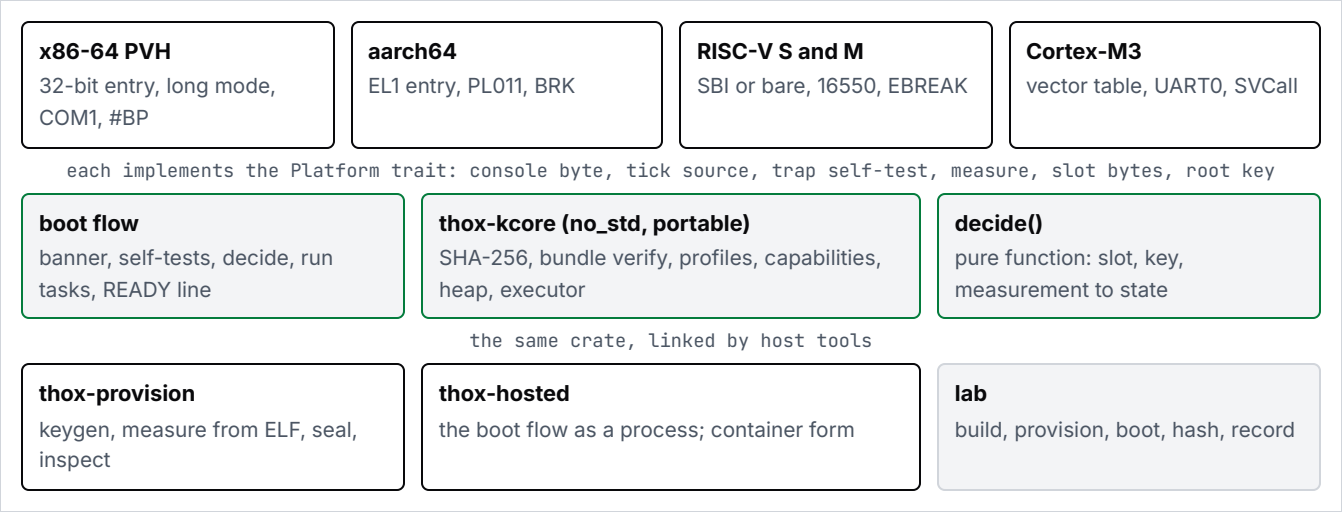


Figure 1. Structure. Green boxes are the portable core. Nothing above the trait boundary knows which machine it runs on; nothing below it makes a trust decision.

The split is strict. The core crate has no `cfg` on architecture and no inline assembly. A port supplies an entry stub that reaches Rust with a stack, a way to write one byte to a console, a trap vector with one handled trap, and a power-off call. Everything a reviewer needs to audit for the trust decision is in one crate that also compiles and tests on a workstation. Of the kernel's Rust source, 62.1 percent is this shared crate.

5.2 Boot flow

Algorithm 1. Boot decision (crate thox-kcore, module boot)

```
trap   := platform.trap_selftest()            # raise and handle one synchronous trap
m      := SHA-256(image[measure_start .. measure_end])
slot   := platform.bundle_bytes()             # 168 bytes at the board's slot address
state  := LOCKDOWN(bundle-absent)             if slot is erased or missing
          LOCKDOWN(bad-magic | bad-signature) if not Verify(root_pub, DOMAIN || slot[0..104],
slot[104..168])
          LOCKDOWN(bad-format | bad-profile | bad-serial)  on any authenticated field out of range
          LOCKDOWN(measurement-mismatch)      if slot.measurement ≠ m
          PROVISIONED(slot.profile)           otherwise
if trap failed: state := LOCKDOWN(trap-selftest-failed)
caps   := profile.default_caps()  if PROVISIONED  else  {CONSOLE}
run service tasks; every call c by task t is granted iff required(c) is in caps and in grant(t)
print one READY line with the state; power off with a status that mirrors it
```

No field of the record is interpreted before the signature verifies, apart from the two cheap checks that only choose which error is reported for an empty or foreign slot.

5.3 The provisioning record

Offset	Size	Field
0	8	magic `THOXPB01`
8	2	format version, currently 1
10	1	deployment profile
11	5	flags and reserved, must be zero
16	16	device serial, ASCII, zero padded
32	32	SHA-256 measurement of the kernel image
64	8	issue time, Unix seconds
72	32	device Ed25519 public key
104	64	Ed25519 signature by the fleet root key over a domain separator and bytes 0 to 103

Table 3. Wire format, little-endian, 168 bytes. Fixed size and fixed offsets: no parser, no length field to trust, and small enough for one flash page on a microcontroller.

Signing is deterministic, so sealing the same inputs twice gives the same bytes and a fleet database can store records by hash. The domain separator prevents a signature made for this purpose from being accepted as anything else signed by the same key. Eight profiles are defined: the four devices, desktop, server, cloud virtual machine and container. A profile is an identity and a default capability set. It deliberately does not encode a processor or a memory size.

5.4 Self-measurement and its prediction

The linker script of each port places two symbols around the code and read-only data. At boot the image hashes the bytes between them. The fleet tool reads the same two symbols from the ELF symbol table and rebuilds that address range from the loadable segments, so it predicts the value before any unit boots. Text and read-only data share one loadable segment on every port, which

makes padding between sections file-backed and removes any dependence on what a loader leaves in gaps. The embedded root public key lies inside the measured range: changing the trusted key changes the measurement.

5.5 Capabilities and lockdown

Six capabilities are defined: console, storage, network, inference, key use and update staging. A service call names the one it needs. A call is granted only when the capability is held both by the system and by the calling task. A container profile never holds key use or update staging; the ThoxKey profile holds no network capability in this model. Lockdown is the same mechanism with the system set reduced to the console, so there is one code path, not a special mode.

5.6 Security argument

Proposition 1 (binding). If the image reaches the provisioned state, then the slot holds a record signed by the holder of the root private key whose measurement field equals the SHA-256 of the measured range of the running image.

Argument. By Algorithm 1 the provisioned state is reached only on the last branch, which requires the signature to verify under the embedded key and the measurement field to equal the computed value. Under the unforgeability of Ed25519 the first holds only for a record the key holder signed. The embedded key is inside the measured range, so an image with a different key has a different measurement.

Proposition 2 (fail closed). Every execution of the boot decision that does not reach the provisioned state holds exactly the console capability.

Argument. The state type has two variants. The capability set is computed by one expression that returns the profile's set for the first variant and the console set for the second. There is no third path. The property is checked by unit tests for each lockdown reason and by every negative lab boot.

What the propositions do not say. They assume the measured range is what executes and that the code performing the check has not itself been replaced. Without an immutable first stage that assumption is not enforced by hardware. The propositions therefore bound accidental and partial tampering, replay across images and unkeyed images. They do not bound a full image replacement.

6 Implementation

RK is 2,576 non-blank lines of Rust for the kernel (1,600 portable core, 167 shared kernel glue, 809 across 5 ports), 110 lines of linker script, 739 lines for the fleet tool and 106 for the hosted process. It builds with the stable toolchain (rustc 1.97.0 (2d8144b78 2026-07-07)) against built-in targets. The only third-party crate is a dependency-free Ed25519 implementation under the MIT licence [21]; SHA-256, the heap and the executor are written in the core crate and tested against published vectors where they exist.

6.1 Ports

Target	Rust target	Entry state	Console	Self-test trap	Power-off	Slot	Port lines	Text bytes
x86_64 (PVH)	`x86_64-unknown-none`	PVH note, 32-bit protected mode; stub builds page tables and enters long mode	COM1	#BP via IDT	debug-exit port	64 MiB physical	255	32,943
aarch64	`aarch64-unknown-none-softfloat`	EL1, MMU off	PL011	BRK via VBAR_EL1	PSCI	128 MiB into memory	151	34,864
riscv64 (S-mode)	`riscv64imac-unknown-none-elf`	S-mode from OpenSBI	NS16550A	EBREAK via stvec	SBI system reset	128 MiB into memory	142	32,008
riscv32 (M-mode)	`riscv32imac-unknown-none-elf`	M-mode, first code to run	NS16550A	EBREAK via mtvec	test finisher	128 MiB into memory	148	41,800
Cortex-M3	`thumbv7m-none-eabi`	vector table at reset; copies data, clears bss	UART0	SVCall	semihosting exit	last flash page	113	31,010

Table 4. The five bare-metal ports. Machine names are QEMU models chosen as stand-ins for a machine class; none is a THOX board.

The hosted target runs the same boot flow as an ordinary process and measures its own executable file. It is the form a container takes, because a container shares its host's kernel and has no kernel of its own to boot. It also builds as a fully static binary.

6.2 Requirements to tests

Requirement	Where it is checked
R1 Portability	The core crate has no architecture conditional; all six targets link it and pass the same matrix (Figure 2).
R2 Binding	Unit tests: round trip, every single-bit flip of a sealed record rejected (1,344 flips), wrong key, zero key. Lab: valid, bit-flipped, wrong-key, other-image, patched-image scenarios.
R3 Fail closed	Unit tests for each lockdown reason including a failed trap self-test overriding a valid record. Lab: all negative scenarios.
R4 Least authority	Unit tests on capability intersection; lab console shows grants and denials per call in both states.
R5 Predictability	Lab compares the host-predicted measurement with the value each guest prints.
R6 Small footprint	The Cortex-M image links into 252 KiB of flash and runs in 64 KiB of memory (Figure 3).
R7 Reproducibility	Clean rebuild of each image in a second directory compared by hash; lab driven by one command.
R8 Minimal dependencies	One third-party crate; lock file committed.

Table 5. Requirement to test mapping. Host suite: 37 tests passed, 0 failed; formatting and lints pass on the host and on all five bare-metal targets (8 checks).

6.3 What the lab caught

Three defects in our own work surfaced only because the lab compares what the host predicts with what the guest reports.

1. **The processor wrote into the measured range.** On x86-64 the segment descriptor table sat in read-only data. Loading a segment register sets an accessed bit in its descriptor, so the image hashed bytes that differed from the file and locked itself down with a valid, correctly signed record. The fix sets the bit in the source. The general rule: nothing the hardware may write belongs inside a measured range.
2. **Padding that the file did not back.** On three ports the linker placed code and read-only data in separate segments with a gap between them. The gap is defined in memory only by what the loader leaves there. One loadable segment for both removes the dependence.
3. **Size on 32-bit targets.** With the default signature tables the microcontroller image overflowed its flash by 28 KiB. The size-optimised configuration is now used on every target so that all images run one verification path.

7 Evaluation

7.1 Setup

All results come from lab run `2026-11-11-run1` at source commit `35a21f1219` with a clean tree, on a virtual machine with 4 logical processors (Intel(R) Xeon(R) Processor @ 2.10GHz, Linux 6.18.44-fc-v378), QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18), rustc 1.97.0 (2d8144b78 2026-07-07). Guests run under the TCG interpreter with one virtual processor; hardware virtualisation was not available. The lab uses deterministic, publicly known keys so that a rerun reproduces the same image hashes. The evidence directory holds the environment, every command, every console capture and a hash of each file.

7.2 Does one core carry the decision on every target?

	valid bundle PROVISIONED	no bundle LOCKDOWN	bit-flipped bundle LOCKDOWN	wrong signing key LOCKDOWN	bundle for other image LOCKDOWN	image patched after sealing LOCKDOWN	image built without root key LOCKDOWN
x86_64 (PVH)	as expected	as expected	as expected	as expected	as expected	as expected	as expected
aarch64	as expected	as expected	as expected	as expected	as expected	as expected	as expected
riscv64 (S-mode)	as expected	as expected	as expected	as expected	as expected	as expected	as expected
riscv32 (M-mode)	as expected	as expected	as expected	as expected	as expected	as expected	as expected
Cortex-M3	as expected	as expected	as expected	as expected	as expected	as expected	as expected
hosted process	as expected	as expected	as expected	as expected	as expected	as expected	as expected

Each cell is one boot. "as expected" means the READY line, exit status and trap self-test all matched the expected verdict.

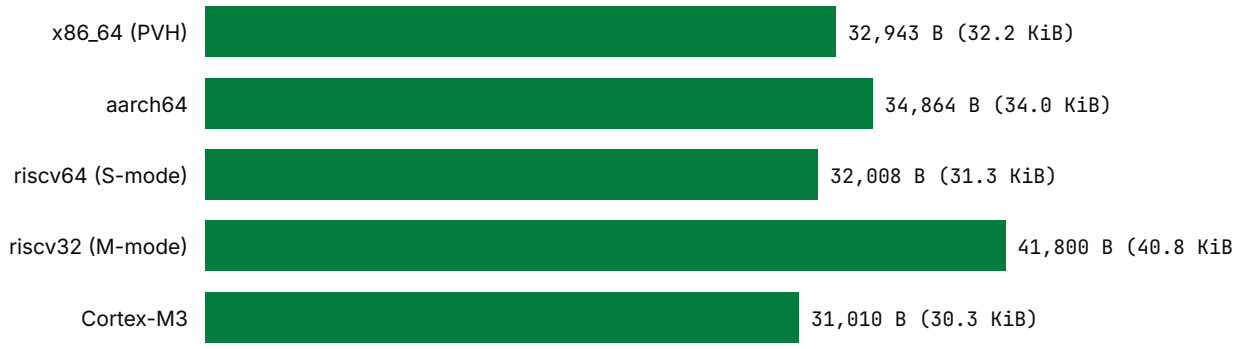
Figure 2. Verdict matrix, emulated. 7 scenarios on each of 6 targets, 42 boots; all matched. A further 4 boots provisioned the remaining profiles, for 46 in total.

All 46 boots matched their expected verdict. A boot counts only if three independent signals agree: the READY line names the expected state and reason, the process exit status matches where the platform carries one, and the trap self-test reports success. All 10 boots with a valid record reached the provisioned state, covering all eight profiles (cloudvm, container, desktop, server, thoxair, thoxclip, thoxkey, thoxmini). All 36 negative boots ended in lockdown with the specific reason expected: `bad-signature`, `bundle-absent`, `measurement-mismatch`.

7.3 Does the host predict what the guest measures?

Yes. On every target the measurement printed by the provisioned guest equals the value the fleet tool computed from the image file before boot. The patched-image scenario is the converse: one byte of read-only data changed after sealing, and every target reported a measurement mismatch. Clean rebuilds of all five kernel images in a second build directory were bit-identical to the first on the lab host. We did not test reproducibility across hosts or toolchain versions.

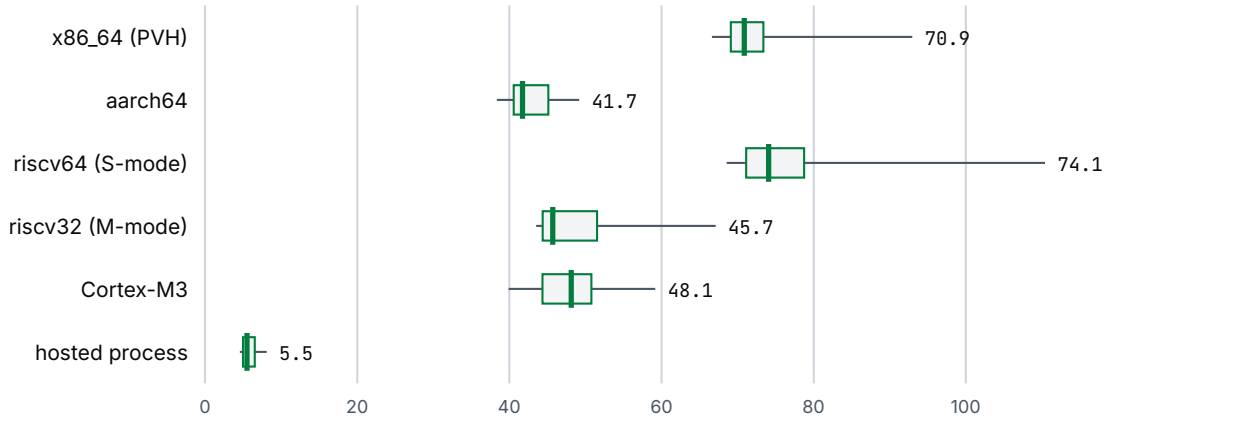
7.4 How large is it?



.text plus .rodata as reported by size(1); release profile, opt-level s, LTO.

Figure 3. Kernel text per target, host measured. All five images fall between 30.3 and 40.8 KiB. Most of each image is signature verification and formatting; the port itself is a few hundred bytes.

7.5 How long does a boot take under emulation?



Milliseconds, process start to exit, n = 30 per target. Box: quartiles. Line: min to max. Bar and label: median. Emulator wall-

Figure 4. Boot wall-clock, emulated, 30 runs per target after one discarded warm-up. Start-up of the emulator is included.

Target	Runs	Wall-clock median (quartiles), ms	Guest clock median (quartiles), ms
x86_64 (PVH)	30	70.9 (69.1 to 73.4)	n/a
aarch64	30	41.7 (40.6 to 45.1)	7.8 (7.6 to 8.7)
riscv64 (S-mode)	30	74.1 (71.1 to 78.8)	8.4 (7.9 to 9.4)
riscv32 (M-mode)	30	45.7 (44.4 to 51.5)	17.8 (17.1 to 20.2)
Cortex-M3	30	48.1 (44.4 to 50.8)	n/a
hosted process	30	5.5 (5.0 to 6.5)	3.2 (3.0 to 4.2)

Table 6. Timing. Wall-clock is process start to exit. Guest clock is the interval the guest's own timer counts across the boot flow, on targets where the proof of concept has a calibrated counter.

The longest median is 74 ms. These figures show that the check adds no visible delay to an emulated boot and that the lab is quick to rerun. They are **not** device performance: an interpreter on a shared virtual machine says nothing reliable about a microcontroller or an application processor, and the review and documentation teams were working on the same host during the run. We report no comparison between targets and draw no conclusion from the differences.

8 Discussion and limitations

What was not run.

- any physical THOX device or development board.
- KVM or other hardware virtualisation (TCG only).
- Firecracker and Cloud Hypervisor (PVH image built, booted on QEMU microvm only).
- UEFI or BIOS boot on desktop or server hardware.
- container image build (no container runtime on the lab host).
- Armv7-A and Xtensa targets (not built).

What is not built. There is no user mode and no memory protection, so the capability model is policy at a call boundary, not isolation. There are no interrupts, no preemption and no timer-driven scheduling. The only driver is a polled serial console: no storage, network, display, radio or accelerator. There is no file system, no update mechanism and no model runtime. The device public key in the record is carried and displayed but not yet used for attestation.

What the trust model does not cover. No hardware root of trust anchors the measurement; a full image replacement including the embedded key is out of scope. There is no rollback protection and no revocation: an old, correctly sealed image remains acceptable. The lab keys are public by design and must never seal a real unit.

Statistical scope. The verdict matrix is deterministic and each cell is one boot; it is a functional result, not a rate. Timing is 30 runs per target on one shared host. The review read 39 repositories once, on one day; repositories outside the session's access, including two that hold firmware-relevant code, were not seen.

Literature access. Publisher and preprint sites were not reachable from the lab network. References were therefore limited to those whose title, authors and venue could be read from a project or publisher repository, and several relevant works we intended to discuss, including the Linux-compatible framekernel line, studies of Rust in the Linux kernel, the lightweight virtual machine monitor literature and the classic secure-boot and capability papers, are omitted rather than cited from memory. No systematic search for the closest prior work was possible, so this report makes no novelty claim; a full literature pass is required before any external submission.

Why not start from an existing system? For a microcontroller product with application isolation, an existing embedded Rust operating system is the stronger base, and for Linux-class boards the practical production path remains Linux with Rust above it, at the cost of copyleft obligations for anything inside the kernel. We built a small kernel anyway because the question was whether one

core could carry the trust decision across classes, and the cheapest way to prove portability is to have nothing else in the image. The core crate is the reusable result; the kernel around it is scaffolding.

9 Conclusion

One portable Rust crate of 1,600 lines carried the same measured, signed provisioning decision onto five emulated machine classes and a hosted process, in 30.3 to 40.8 KiB of kernel text, and failed closed in every one of 36 tampering and misconfiguration boots. The review that preceded it found that THOX's operating-system record names more than it contains, and that the most useful next step is not another kernel but a decision about which one to finish. RK gives that decision a tested trust base it can keep whichever way it goes.

10 Roadmap and decisions for the founders

Decision	Options	What the evidence says
Which kernel ships on devices	Continue the ThoxInux fork; grow a small owned kernel; Linux with Rust userspace	ThoxInux is x86-64 only and its upstream moves faster than the fork tracks. Device images are specified as Linux on Arm. RK proves portability of the trust base, not of an operating system.
What each device is	Confirm a board per device; decide what ThoxKey is as a compute target	The record holds category floors and conflicting candidates. No port can be finished without this.
What ThoxOS names	One product name with editions; separate names per artefact	At least 18 artefacts carry the name today, including a language model titled a kernel.
Signing key custody	Offline key with a ceremony; hardware security module; per-line intermediate keys	No production signing key exists. Every provisioning design depends on this.

Table 7. Decisions that engineering cannot make. Each is tracked as an issue with its findings attached.

Engineering next, in order.

1. Run the x86-64 image under hardware virtualisation on lightweight monitors that accept PVH images, and add a UEFI entry for desktop and server hardware.
2. Add user mode with memory protection on one application-class and one microcontroller-class port, so the capability model becomes isolation.
3. Add interrupts and a timer, then preemption.
4. Bring up the first physical board once a device is confirmed; add the 32-bit Arm application-class port the proof-of-concept hardware needs.
5. Anchor the measurement in an immutable first stage, add rollback protection, and use the device key for attestation.
6. Tie A/B update to the record, reusing the slot state machine the review found in the existing update crate.

7. Link the core crate into the existing Linux provisioning daemon so shipping units and RK share one record format.

A Appendix A. Reproducibility

```
# from a checkout of the THOX source tree (private repository, access on request)
cd thoxos/rk
git checkout 35a21f1219f40fbe1c498e77ef3076e9ebfe0553
scripts/rk setup            # Rust targets and QEMU
scripts/rk test             # 37 host tests, formatting, lints on six targets
scripts/rk lab              # builds, provisions, boots; writes lab/evidence/<run-id>/
python3 paper/build.py      # regenerates this report and the lab page from that evidence
```

The evidence for this report is `rk/lab/evidence/2026-11-11-run1/`: `environment.json`, `builds.json`, `host_checks.json`, `scenarios.json`, `timing.json`, `summary.json`, one console capture per boot, every sealed record, the public keys, and `SHA256SUMS`. Every number in this report is read from those files by the build script; none is typed. The companion lab page embeds the same evidence, recomputes each capture's hash in the browser and reruns the record check.

B Appendix B. Console and wire reference

Console capture of one provisioned boot (aarch64, emulated), verbatim from the evidence:

```
THOX-RK 0.1.0 arch=aarch64 machine=qemu-virt
[trap] self-test ok
[heap] capacity=262144 used=416 peak=752
[measure] sha256=a8d9d592ae8e0b32aca924ef3bdaacfe3d09c363c51d50b30438089010505c7d bytes=34864
[root] pub=a0d9a0ed8de21eabe6ced8d6184721f267ce9ac02386039a03026f99a69fff1d
[bundle] profile=thoxmini serial=LAB-0001 issued=1794355200
device=7b9bbc2061742769b1f9143d76f7e352b460569b6536b61bce4e7e3f52ae6ed3
[cap] grant task=attest call=log
[cap] grant task=infer call=storage.read
[cap] grant task=telemetry call=log
[cap] grant task=attest call=key.sign
[cap] grant task=infer call=infer
[cap] DENY task=telemetry call=net.send
[cap] grant task=infer call=log
[cap] DENY task=telemetry call=update.stage
[sched] tasks=3 polls=11 rounds=4 starved=false
[time] ticks=475862 hz=62500000
THOX-RK READY state=PROVISIONED profile=thoxmini granted=6 denied=2
```

Lockdown reasons: `bundle-absent`, `bundle-truncated`, `bad-magic`, `bad-format`, `bad-profile`, `bad-serial`, `bad-signature`, `measurement-mismatch`, `trap-selftest-failed`. The signed message is the 18-byte domain separator `THOX-PROVISION-V1` with a trailing zero byte, followed by bytes 0 to 103 of the record.

C Appendix C. Authorship and disclosure

This report was prepared by THOX.ai Research with substantial AI assistance: the repository review, the source code, the lab, the analysis and the text were produced by AI systems working under the

direction of the corresponding author, and every result was generated by running the released code. No AI system is an author. This is a preprint: it has not been peer reviewed, has no DOI, and has not been submitted to any venue. External references were opened and checked on the date of this report; where a primary site could not be reached from the lab network the source repository was read instead, and the reference list gives the address that was opened. Licence statements are engineering research and should be confirmed by counsel before a product decision. The lab signing keys are public test keys.

References

[1] Craig Ross, Tommy Xaypanya. ThoxOS and ThoxInux: An Evidence-Bounded Architecture for a Local-First AI Operating System. THOX.ai Research technical report v1.0.1, Zenodo, 2026. https://doi.org/10.5281/zenodo.23969073

[2] Philipp Oppermann. A Minimal Rust Kernel. Writing an OS in Rust (blog series), post dated 2018-02-10, 2018. https://os.phil-opp.com/minimal-rust-kernel/

[3] Redox OS Developers. Redox (Build System for Redox OS). Software repository (GitHub mirror; README points to gitlab.redox-os.org and doc.redox-os.org/book), 2026. https://github.com/redox-os/redox

[4] flosse (GitHub account) and contributors. Rust OS comparison. GitHub repository, 2026. https://github.com/flosse/rust-os-comparison

[5] gmkurtzer (GitHub account). Ginux. GitHub repository, 2026. https://github.com/gmkurtzer/ginux

[6] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17), ACM, pp. 234-251, 2017. https://doi.org/10.1145/3132747.3132786

[7] Leon Schuermann, Brad Campbell, Branden Ghena, Philip Levis, Amit Levy, Pat Pannuto. Tock: From Research To Securing 10 Million Computers. Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25), pp. 36-49, 2025. https://doi.org/10.1145/3731569.3764828

[8] Hudson Ayers, Prabal Dutta, Philip Levis, Amit Levy, Pat Pannuto, Johnathan Van Why, Jean-Luc Watson. Tiered Trust for Useful Embedded Systems Security. Proceedings of the 15th European Workshop on Systems Security (EuroSec '22), ACM, pp. 15-21, 2022. https://doi.org/10.1145/3517208.3523752

[9] Oxide Computer Company (GitHub organisation oxidecomputer). Hubris. Software repository and 'Hubris Reference' documentation (doc/index.adoc, doc/intro.adoc), 2026. https://github.com/oxidecomputer/hubris

[10] Kevin Boos, Namitha Liyanage, Ramla Ijaz, Lin Zhong. Theseus: an Experiment in Operating System Structure and State Management. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), USENIX Association, pp. 1-19, 2020. https://www.usenix.org/conference/osdi20/presentation/boos

[11] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, Anton Burtsev. RedLeaf: Isolation and Communication in a Safe Operating System. USENIX Symposium on Operating Systems Design and Implementation (OSDI '20), 2020. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[12] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood. seL4: Formal verification of an OS kernel. ACM Symposium on Operating Systems Principles (SOSP), pp. 207-220, Big Sky, MT, USA, 2009. https://sel4.systems/Research/publications.html

[13] Trusted Computing Group. Hardware Requirements for a Device Identifier Composition Engine. TCG DICE Work Group specification, n.d.. https://trustedcomputinggroup.org/resource/hardware-requirements-for-a-device-identifier-composition-engine/

[14] S. Josefsson, I. Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, IRTF stream, 2017. https://www.rfc-editor.org/info/rfc8032

[15] Quynh H. Dang. Secure Hash Standard. NIST FIPS 180-4, National Institute of Standards and Technology, 2015. https://doi.org/10.6028/NIST.FIPS.180-4

[16] Xen Project. x86/HVM direct boot ABI. Xen documentation, docs/misc/pvh.pandoc, 2026. https://xenbits.xen.org/docs/unstable/misc/pvh.html

[17] The QEMU Project Developers. 'microvm' virtual platform (microvm). QEMU documentation, docs/system/i386/microvm.rst, 2026. https://www.qemu.org/docs/master/system/i386/microvm.html

[18] Western Digital Corporation or its affiliates, RISC-V International. RISC-V Open Source Supervisor Binary Interface (OpenSBI). Software repository, version 1.9 (include/sbi/sbi_version.h on master), 2026. https://github.com/riscv-software-src/opensbi

[19] RISC-V Platform Runtime Services Task Group. RISC-V Supervisor Binary Interface Specification. RISC-V Non-ISA Specification, Version 3.0 (Ratified); System Reset Extension (EID #0x53525354 "SRST"), 2026. https://github.com/riscv-non-isa/riscv-sbi-doc

[20] The QEMU Project Developers. Generic Loader. QEMU documentation, docs/system/generic-loader.rst, 2026. https://www.qemu.org/docs/master/system/generic-loader.html

[21] Frank Denis. A compact Ed25519 and X25519 implementation for Rust (ed25519-compact). Rust crate, version 2.6.1 (Cargo.toml on master), 2026. https://github.com/jedisct1/rust-ed25519-compact