

MITRE eCTF 2026 | Team NEU-1 / NeuSense

The NeuSense Hardware Security Module

Architecture and Implementation Guide

A bare-metal HSM on ARM Cortex-M0+

Kalyan Gopalam

Team Lead, Team NEU-1 / NeuSense

M.S. Cybersecurity, Northeastern University

gopalam.k@northeastern.edu | kalyangopalam.com

Version 1.0 | August 2026

Status of this document

I want to be straight with you about what this document is before you read a single technical page, because a specification that overstates its own authority is worse than no specification at all.

This is a **reconstruction and expansion** of the NeuSense HSM architecture, written after the fact, from the design decisions and the security model my team settled on during the competition. It is written to be complete enough that somebody who has never seen my repository can build the same device from scratch, in whatever language they like, on whatever microcontroller they like.

What that means in practice:

- **The architecture, the threat model, the key hierarchy, the permission model, the authentication design, and the storage design are mine.** These are the load-bearing parts of the document, and they are the parts I would defend in a design review.
- **The code in this document is reference code, not the competition code.** It is written to teach the architecture clearly and to be portable. My competition firmware is tuned for one board and one scoring rubric, and reading it cold would teach you less than reading this.
- **Any specific register name, memory address, peripheral offset, or silicon quirk should be checked against the current TI datasheet and technical reference manual before you rely on it.** I have written the hardware chapters in terms of *what the peripheral must do for you* rather than in terms of magic numbers, precisely so this document does not rot the next time TI revises a manual.
- **Appendix F is a list of things I still need to confirm against my own repository.** It is short, it is honest, and it is at the back where I can find it. The most important item on it is the exact depth of the key derivation hierarchy, which I have documented here as three tiers.

If you are reading this as a hiring manager or a reviewer: the value is in the reasoning, not in the line count. Every significant choice in here has a section explaining what else I could have done and why I did not do it. That is the part I care about.

If you are reading this to build something: start at Chapter 1, read Part I and Part II properly, then skip to whichever of Chapters 19 or 20 matches your language and use the rest as reference.

How to read this document

The document is built in ten parts, and the order is deliberate. It moves from *why the device exists* to *what it runs on* to *what it computes* to *what it remembers* to *who it lets in* to *how it talks* to *how it survives an attacker with an oscilloscope* to *how you actually write it* to *how you prove it works*. Nothing forward-references something you have not read yet, except where I say so explicitly.

Part	Title	What you get out of it
I	The Problem	The device's job, the attacker, and the numbered requirements everything else answers to
II	The Substrate	What a Cortex-M0+ can and cannot do, and what bare metal really costs
III	The Cryptographic Core	AES, HMAC, HKDF, the key hierarchy, and where randomness comes from
IV	State That Survives Power Loss	The secure flash filesystem and the anti-rollback counters
V	Deciding Who Gets In	PIN handling, rate limiting, and the fail-closed permission model
VI	Talking to the Host	The command protocol and the parser, which is where most firmware dies
VII	Surviving Physical Access	Side channels, glitching, and constant-time discipline
VIII	Building It	Reference implementations in C and Rust, plus a porting guide
IX	Proving It	Test strategy, test vectors, self-attack playbook, and release process
X	The Decision Register	Every alternative I rejected, collected in one table with reasons

The three kinds of boxes

I use exactly three callout styles, and they always mean the same thing.

The point

This is the sentence I would want you to remember if you forgot everything else on the page. If you are skimming, read only these.

Get this wrong and the device is broken

This marks a place where a reasonable-looking implementation is silently insecure. Every one of these is a mistake I either made, nearly made, or watched somebody else make.

What else I could have done

This is the alternative design, the reason it is legitimate, and the reason I did not pick it. There is at least one of these in every technical chapter, because a design document without rejected options is just a description.

Notation and conventions**Order**

Everything on the wire and everything in flash is big-endian, unless a cryptographic standard says otherwise. Length fields, counters, and identifiers are all big-endian. This is a deliberate choice and I explain it in Chapter 15.

Concatenation of byte strings. $A\|B$ is the bytes of A followed by the bytes of B .

Bitwise exclusive-or, applied bitwise over equal-length strings.

Hexadecimal literal. 0×00 is one byte of zero.

The extract-and-expand key derivation function of RFC 5869, instantiated with HMAC-SHA-256 throughout.

Case names

`ROOT_KEY`, `DEVICE_KEY` and similar are named cryptographic values defined in Chapter 8.

A numbered security requirement from Chapter 3. Every control in the document traces to one, and Appendix A is the full traceability matrix.

Pseudocode

Language-neutral pseudocode is used when the logic matters more than the syntax. It is always followed by real C or real Rust.

A note on the writing

I have tried to write this the way I would explain it to somebody sitting next to me, which means short sentences, ordinary words, and no reaching for a longer term when a shorter one is exact. Where a piece of jargon is genuinely load-bearing I define it the first time and then use it without apology, because that is how you actually learn a field.

Two things I have deliberately avoided.

The first is the passive voice used to hide who decided what. When I write “I chose,” I chose. When I write “the competition required,” somebody else decided and I worked inside it. The difference matters when you are reading a design document to judge the designer.

The second is confidence I have not earned. Where I am uncertain, the document says so in the sentence itself rather than in a footnote nobody reads. There are five or six of these. They are not weaknesses in the design. They are the parts where the honest answer is “this depends on your threat model” or “I have not measured this yet,” and pretending otherwise would be the actual weakness.

*Kalyan Gopalam
Boston, Massachusetts
August 2026*

Contents

I	The Problem	1
1	What This Device Is, and Why It Exists	2
1.1	The one-sentence version	2
1.2	Why you would want such a thing at all	2
1.3	Where this particular device sits	3
1.4	The functional shape of the device	3
1.5	A picture of the whole thing	4
1.6	What this device deliberately does not do	4
2	The Attacker	6
2.1	Why the threat model comes before anything technical	6
2.2	The capability ladder	6
2.3	What the attacker wants	7
2.4	Trust assumptions, stated explicitly	7
2.5	Attack surface inventory	8
3	Security Requirements	9
3.1	Why requirements have numbers	9
3.2	Confidentiality	9
3.3	Authentication and access control	9
3.4	Integrity	10
3.5	Behaviour under stress	10
3.6	Side channel	10
II	The Substrate	12
4	The Hardware You Actually Have	13
4.1	Reading a part with security eyes	13
4.2	The core: ARM Cortex-M0+	13
4.3	Memory: how much and of what kind	14
4.4	The AES accelerator	15
4.5	DMA, and why it earns its place	15
4.6	The gap list	16
5	What Bare Metal Actually Costs	18
5.1	The phrase, unpacked	18
5.2	Cost one: you write the boot sequence	18
5.3	Cost two: you own every byte of memory	19
5.4	Cost three: you have no stack protection	20
5.5	Cost four: you decide what happens when things go wrong	21
5.6	The main loop, in full	21

III	The Cryptographic Core	23
6	Choosing the Primitives	24
6.1	The shortest possible list	24
6.2	Why symmetric only	24
6.3	Why AES-256 and not AES-128	25
6.4	Why CBC, honestly	25
6.5	HMAC, and why not a bare hash	27
7	Driving the AES Engine	29
7.1	The contract, restated portably	29
7.2	The DMA pairing	29
7.3	Boot-time known-answer tests	30
8	The Key Hierarchy	32
8.1	The problem a hierarchy solves	32
8.2	HKDF in two paragraphs	32
8.3	The three tiers	33
8.4	What the hierarchy buys you	35
9	Where Randomness Comes From	36
9.1	The hardest problem in the document	36
9.2	What this part gives you, and what it does not	36
9.3	The rules I follow	38
9.4	Designing so that randomness matters less	38
IV	State That Survives Power Loss	39
10	The Secure Flash Filesystem	40
10.1	Why this is harder than it looks	40
10.2	The design: an append-only authenticated log	40
10.3	The record format	41
10.4	Reading: the boot-time scan	42
10.5	Writing: the append	43
10.6	Compaction: the one dangerous moment	44
11	Counters That Cannot Go Backwards	45
11.1	Why counters are the whole game	45
11.2	The counters this device keeps	45
11.3	The increment-before-check rule	45
11.4	Replay defence with the message counter	46
11.5	Revocation through the generation counter	47
V	Deciding Who Gets In	49
12	PIN Authentication and Rate Limiting	50
12.1	The unavoidable weakness of a PIN	50

12.2 Never store the PIN	50
12.3 The comparison	51
12.4 The backoff schedule	52
12.5 The complete authenticate path	53
13 The Fail-Closed Permission Model	55
13.1 What fail-closed means, precisely	55
13.2 The model: capabilities, not roles	55
13.3 Redundancy against fault injection	56
13.4 Control-flow integrity, cheaply	57
14 Session Lifecycle	58
14.1 Four states, and no others	58
14.2 The rules that hold in every state	58
VI Talking to the Host	60
15 The Command Protocol	61
15.1 Design goals, in priority order	61
15.2 Frame format	61
15.3 One response, always	62
15.4 Session binding and replay	63
16 The Parser Is the Most Dangerous Code You Will Write	64
16.1 Why this chapter exists	64
16.2 Total parsing	64
16.3 Fuzzing is not optional	66
16.4 The receive path	66
VII Surviving Physical Access	68
17 Side Channels and Fault Injection	69
17.1 The shift in mindset	69
17.2 The channels, and what leaks through each	69
17.3 Timing: the one you must fix	69
17.4 Power analysis, honestly assessed	70
17.5 Fault injection	71
18 Constant-Time Discipline	73
18.1 The rule, and why it is a rule rather than a technique	73
18.2 The toolkit	73
18.3 The compiler is not on your side	74
18.4 Secure zeroing, and why <code>memset</code> is not it	74
18.5 Verifying that it worked	75

VIII Building It	77
19 The C Implementation	78
19.1 Project layout	78
19.2 The single configuration header	78
19.3 Command dispatch	79
19.4 A complete handler	80
19.5 Measuring worst-case stack depth	82
20 The Rust Implementation	83
20.1 Why a second implementation	83
20.2 Zeroize on drop	83
20.3 Type-state for the session	84
20.4 Constant-time code in Rust	86
20.5 Panics, and <code>no_std</code>	86
21 Porting This Design	88
21.1 What is portable and what is not	88
21.2 Porting to a different microcontroller	88
21.3 Porting to another language	89
21.4 Porting the architecture, not the code	91
IX Proving It	93
22 Test Strategy	94
22.1 Five layers, each catching what the one below misses	94
22.2 Known-answer tests	94
22.3 Property-based tests	94
22.4 Hardware-in-the-loop	95
23 Attacking Your Own Device	97
23.1 Why you do this yourself first	97
23.2 The playbook	97
23.3 What to do when you find something	98
24 Build, Provisioning, and Release	99
24.1 Reproducible builds	99
24.2 Provisioning	100
24.3 Release checklist	101
24.4 What a real product would need that this does not have	101
X The Decision Register	103
25 Every Alternative, In One Place	104
25.1 Why this chapter exists separately	104
25.2 The register	104
25.3 The four decisions I would revisit first	107

A Requirement Traceability Matrix	108
B Test Vectors	109
B.1 Where to get them	109
B.2 The minimum set to embed on-target	109
B.3 Vectors you must generate yourself	110
C Memory Map and Constants	111
C.1 Flash layout	111
C.2 SRAM layout	111
C.3 Linker script fragment	111
D Glossary	113
E Review Checklists	115
E.1 Reviewing a change to the parser	115
E.2 Reviewing a change that touches key material	115
E.3 Reviewing a change that touches persistent state	115
E.4 Reviewing a change that adds a command	116
F Open Items	117
F.1 Things I need to confirm against my own repository	117
F.2 Things I would add with more time	117

PART I

The Problem

What This Device Is, and Why It Exists

1.1 The one-sentence version

The NeuSense HSM is a small, sealed computer whose entire job is to hold secrets that the machine it plugs into is not allowed to see, and to perform operations with those secrets only for people who can prove they are permitted.

That sentence contains the whole design. Everything in the remaining seventy-odd pages is an elaboration of one of its four clauses:

- **small, sealed computer** gives us Part II, the hardware substrate and what bare metal costs us.
- **hold secrets** gives us Part III and Part IV, the cryptographic core and the storage that survives power loss.
- **the machine it plugs into is not allowed to see** gives us Part VI and Part VII, the protocol boundary and the physical hardening.
- **only for people who can prove they are permitted** gives us Part V, authentication and the permission model.

The point

An HSM is not a computer that does cryptography. Plenty of computers do cryptography. An HSM is a computer built so that the secrets inside it cannot leave, even when the person holding the device is the attacker.

1.2 Why you would want such a thing at all

Suppose you write an application that needs to sign messages. You generate a signing key, you store it in a file, you set the file permissions carefully, and you feel reasonably good about it.

Now list the things that can read that file. The application, obviously. Also the operating system, any process running as root, anybody with physical access to the disk, the backup system, the crash dump handler if the key ever sits in memory during a fault, the swap file, and any other process that can trick the kernel into reading it. That is a long list, and the length of the list is the problem. Every item on it is a thing you must now defend.

The HSM approach is to shorten the list to one item: the HSM itself. The key is generated inside the device, never leaves the device in usable form, and the only thing the outside world can do is ask the device to perform an operation and receive the result. The outside world does not get the key. It gets the *effect* of the key.

That distinction is the whole value proposition, and it is worth stating carefully, because it is easy to nod along without absorbing it:

The point

The security boundary is not the file, and it is not the process. It is the physical edge of the chip. Anything inside is trusted. Anything outside, including the host computer that supplies power to the device, is treated as hostile.

1.3 Where this particular device sits

The eCTF gives every team a common embedded platform and a common functional specification, and then invites every other team to break what you built. That structure shapes the design in three ways that a normal product would not be shaped.

1.3.1 You do not choose your hardware

The board is given. In our case that is a Texas Instruments MSPM0L2228, which puts an ARM Cortex-M0+ core, a modest amount of SRAM, an on-chip flash array, an AES hardware accelerator, and a DMA controller in front of you, and nothing else. You do not get a secure element. You do not get a memory protection unit worth the name. You do not get a hardware true random number generator you can lean on without thinking. Every gap between what you have and what you want has to be closed in firmware, and Chapter 4 is a full accounting of those gaps.

1.3.2 Your attacker is competent, motivated, and has your board

This is the part that separates the exercise from a normal school project. The attacker is another team of security students who have the same datasheet you do, the same board you have, an oscilloscope, a logic analyser, and several weeks. They are not guessing. They are reading your protocol, timing your comparisons, and glitching your power rail while your PIN check runs.

1.3.3 Functionality is scored, but a single break is worse than a hundred missing features

A device that does everything the specification asks and leaks one key is a failed device. A device that is slightly slow but never leaks anything is a good device. This asymmetry justifies design choices that would look paranoid in a commercial product with a deadline, and I have flagged them in the text where they appear.

What else I could have done

The tempting alternative, and the one several teams take, is to build for the functional tests first and harden afterwards. It is faster and it scores early points. I did not do it, because hardening is not a coat of paint you apply at the end. Constant-time comparison, fail-closed defaults, and the ordering of authentication before parsing are all structural. Retrofitting them means rewriting the same code twice, and the second rewrite happens under deadline pressure, which is exactly when people leave a `memcmp` in place.

1.4 The functional shape of the device

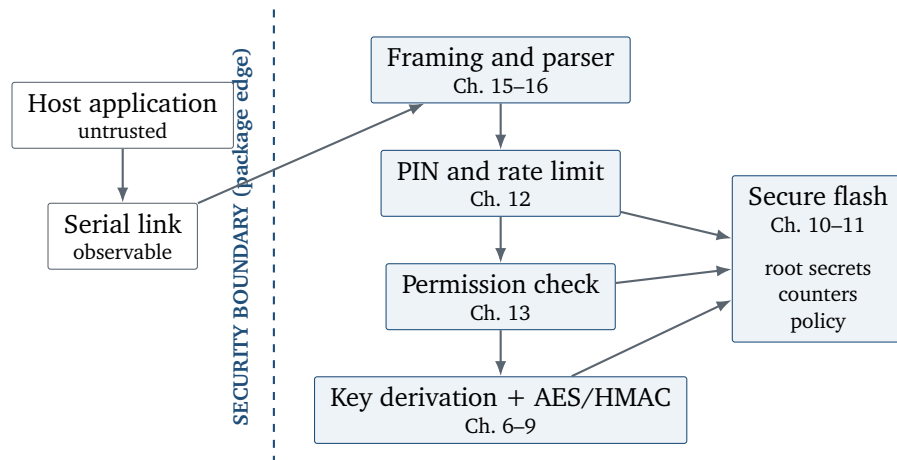
Stripped of competition-specific detail, the device does five things. Everything the firmware contains exists to support one of these.

1. **It is provisioned once, at manufacture.** A trusted build process writes a device identity and a small number of root secrets into flash. This happens exactly once, in a controlled environment, and after it the provisioning path is closed forever.
2. **It authenticates a user.** A human presents a PIN. The device decides whether that PIN is correct, and it does so in a way that survives an attacker who can measure how long the decision took and who can try again as many times as physics allows.
3. **It derives working keys.** From the root secrets, plus context describing what is being asked, it computes a key that is specific to that context and useless anywhere else.

4. **It performs an authorised operation.** Encrypt, decrypt, or authenticate a block of data with the derived key, and return only the result.
5. **It remembers what happened.** Failed attempt counters, revocation state, and version counters persist across power loss, because an attacker who can reset the device to a fresh state has beaten every counter you rely on.

1.5 A picture of the whole thing

Before any detail, here is the shape of the system. Read it left to right: the untrusted world, the boundary, and the trusted interior.



Notice the order of the interior boxes. Parsing happens first because you cannot avoid it, and that is exactly why Chapter 16 treats the parser as the most dangerous code in the firmware. Authentication happens before permission checking, and permission checking happens before any key is derived. A request that fails at any stage never reaches the next one, and no key material is ever computed for a request that will be refused. That ordering is not an optimisation. It is the design.

1.6 What this device deliberately does not do

A specification is defined as much by its exclusions as its inclusions, and vague scope is how firmware projects rot. NeuSense does not:

- **Do public key cryptography.** No RSA, no ECDSA, no key agreement. Everything is symmetric. Chapter 6 explains why that is a defensible choice here and a bad choice elsewhere.
- **Support multiple concurrent sessions.** One session at a time, strictly. Concurrency in security state machines is where subtle bugs live, and there was no requirement that justified the risk.
- **Support firmware update in the field.** The update path is the single most attractive target on any embedded device, and a device that cannot be updated cannot be maliciously updated. Chapter 24 covers what a real product would need to add here and why it is genuinely hard.
- **Attempt tamper detection or tamper response.** No mesh, no light sensor, no case switch. The package is what it is. Chapter 17 is honest about what that means for an attacker with the right equipment.

- **Provide an audit log readable by the host.** Counters, yes. A log, no. A log the host can read is a log the host can use for reconnaissance.

The point

Every one of those exclusions removed code. Removed code cannot be attacked. On a device this small, the strongest security control available to you is usually the feature you decided not to build.

The Attacker

2.1 Why the threat model comes before anything technical

If you cannot say who you are defending against, you cannot tell whether a control is worth its cost, and you will spend your effort in the wrong places. I have seen embedded projects add elaborate obfuscation to their protocol while leaving a timing leak in the PIN comparison. That is not carelessness. It is what happens when nobody wrote down who the attacker is.

So: here is the attacker, with capabilities enumerated, in increasing order of how much they cost the attacker to obtain.

2.2 The capability ladder

Level	Attacker	What they can do
A-0	Curious user	Send arbitrary well-formed commands through the normal interface. Read every response. Try PINs.
A-1	Protocol attacker	Send arbitrary <i>malformed</i> bytes. Truncate messages, send oversized lengths, replay old messages, reorder, interleave, disconnect mid-transaction.
A-2	Passive physical	Everything above, plus an oscilloscope on the power rail and a logic analyser on every exposed pin. Measure the exact duration of any operation. Measure power consumption during it.
A-3	Active physical	Everything above, plus the ability to cut power at a chosen instant, to inject a voltage or clock glitch at a chosen instant, and to reset the device at will.
A-4	Invasive	Decapsulate the package, probe internal buses, read flash directly with a microscope or a focused ion beam.

2.2.1 Where I drew the line

NeuSense defends fully against A-0 through A-3. It does not defend against A-4.

That is a real limit and I want to state it plainly rather than bury it. If somebody decapsulates the chip and reads the flash array directly with the right equipment, they get the root secrets, and there is nothing in this design that stops them. The platform provides no tamper mesh, no active shield, no zeroisation-on-intrusion. Adding meaningful A-4 defence requires silicon features this part does not have.

Get this wrong and the device is broken

Do not write a threat model that claims defences you do not have. It is tempting, especially in a competition writeup, to describe A-4 resistance in aspirational terms. A reviewer who knows the part will catch it immediately, and the credibility you lose costs you more than the defence you claimed would have gained. State the limit; explain what would close it;

move on.

What *does* reduce the damage from an A-4 break is the key hierarchy in Chapter 8. Extracting the root secret from one device compromises that device, and devices derived from the same root, but the per-context derivation means the attacker still has to do work per operation rather than getting a single master key that unlocks the entire fleet's traffic. That is damage limitation, not prevention, and I will keep calling it that.

2.3 What the attacker wants

Capabilities are only half the model. The other half is goals, because a control that stops an attacker from doing something they did not want to do is wasted effort.

Goal	Name	What success looks like for them
G-1	Key extraction	Recover any root or derived key in usable form, by any means.
G-2	Authentication bypass	Perform a privileged operation without knowing the PIN.
G-3	Permission escalation	Authenticate legitimately at one privilege level, then perform an operation reserved for a higher one.
G-4	Counter rollback	Undo the persistent record of failed attempts or revocations, so that a lockout can be reset or a revoked capability restored.
G-5	Replay	Cause the device to accept a message it has already processed, or a message intended for a different device.
G-6	Denial of service	Permanently brick a legitimate device.

G-4 deserves a comment because it is the goal that new embedded designers most often forget. Every rate limit in the world is useless if the attacker can power-cycle the device and find the failure counter back at zero. This is why Chapter 11 exists and why the counter is written *before* the check rather than after it. I will come back to that ordering several times, because it is counterintuitive and it is the single most commonly botched detail in PIN-protected embedded devices.

G-6 is the one where I accept partial loss deliberately. If an attacker with physical access wants to destroy a device, they can destroy it with a hammer, and no firmware control competes with a hammer. What I do defend is a *remote* or *protocol-level* permanent brick, because that would let an attacker with only A-1 capability do lasting damage.

2.4 Trust assumptions, stated explicitly

These are the things I am assuming are true. If any one of them is false, parts of this design fail, and I would rather you know which parts.

1. **The provisioning environment is trusted.** The build machine that generates root secrets and writes them into the device is not compromised. If it is, every device it ever provisioned is compromised, and no on-device control helps.
2. **The cryptographic primitives are sound.** AES-256 and SHA-256 are not broken in a way that matters at this scale. If they are, the world has bigger problems than this document.

3. **The hardware AES engine computes AES correctly.** I verify this at boot with a known-answer test (Chapter 7), but I cannot verify it against a malicious accelerator, only a faulty one.
4. **Flash retains data and the erase operation actually erases.** On a device with wear levelling in a hidden translation layer this assumption would be false and dangerous. On raw on-chip flash it holds.
5. **The attacker cannot execute code inside the device.** This is the assumption the entire parser chapter is devoted to protecting. If a buffer overflow gives the attacker code execution, every other control in this document is decoration.

The point

Assumption five is the load-bearing one. Cryptography protects you against an attacker outside the boundary. Nothing in cryptography protects you against an attacker running code *inside* the boundary. That is why memory safety gets its own chapter and why Chapter 20 exists at all.

2.5 Attack surface inventory

Finally, a complete list of every path by which an attacker's influence reaches the firmware. If a path is not on this list, either it does not exist or I have missed it, and in security work the second option is always live, so this list gets reviewed whenever the code changes.

Surface	Reachable by	Primary defence
Serial byte stream	A-0, A-1	Length-bounded framing, Ch. 15
Command parser	A-0, A-1	Total parsing, no pointer arithmetic, Ch. 16
PIN entry path	A-0, A-2, A-3	Constant-time compare, persistent counter, Ch. 12
Permission decision	A-0, A-3	Deny by default, redundant check, Ch. 13
Flash contents	A-3, A-4	Authenticated records, anti-rollback, Ch. 10–11
Timing of any operation	A-2	Constant-time discipline, Ch. 18
Power trace during AES	A-2	Hardware engine, masking discussion, Ch. 17
Reset and brown-out	A-3	Write-before-check ordering, Ch. 11
Debug port	A-3	Fused off in production, Ch. 24

Security Requirements

3.1 Why requirements have numbers

An unnumbered requirement cannot be traced, and an untraceable requirement cannot be tested. Everything in this chapter is written so that a person holding the device can decide, in finite time, whether the requirement holds. If you cannot imagine the test, the requirement is badly written and I have rewritten it.

Each requirement below has an identifier, a statement, a rationale, and the chapter where the control that satisfies it lives. Appendix A collects all of this into a single matrix that maps requirement to control to test.

3.2 Confidentiality

R-01 No key material leaves the device.

No root key, intermediate key, or derived key is ever transmitted on the serial interface, in any encoding, under any command, in any error path.

Rationale: this is the definition of an HSM. *Control:* Chapters 8 and 16. *Test:* exhaustive command fuzzing with a memory scanner watching for any known key byte pattern in output.

R-02 No key material is written to flash in plaintext except the provisioned root.

Derived keys live only in SRAM, and only for the duration of the operation that needs them.

Rationale: flash survives power loss and is readable by A-4. SRAM is not. *Control:* Chapter 8. *Test:* flash image inspection after a full operational cycle.

R-03 Key material is erased from SRAM as soon as the operation completes.

Erase uses a write that the compiler is not permitted to remove.

Rationale: a later fault, crash dump, or glitch that dumps memory should find nothing. *Control:* Chapter 18. *Test:* RAM inspection under debugger after a completed operation, in a debug build.

3.3 Authentication and access control

R-04 No privileged operation executes without a successful authentication in the current session.

Power-on state is unauthenticated. Reset returns to unauthenticated.

Rationale: G-2. *Control:* Chapters 12 and 14. *Test:* attempt every privileged command from a cold boot.

R-05 The permission decision denies by default.

The result of a permission check is deny unless an explicit, positively matched rule grants the operation. There is no code path where an unrecognised operation, an unrecognised role, or an uninitialised policy record produces a grant.

Rationale: G-3, and the general principle that unknown states must be safe states. *Control:* Chapter 13. *Test:* fault injection into the policy record; every corruption must produce deny.

R-06 Failed authentication attempts are counted in non-volatile storage, and the count is incremented before the comparison is performed.

Rationale: G-4. An attacker who cuts power the instant the comparison fails must not escape the increment. *Control:* Chapter 11. *Test:* power-cut sweep across the full timing window of the

authenticate command.

R-07 After a defined number of consecutive failures, the device enters a delay state, and the delay grows.

Rationale: makes brute force uneconomic without permanently bricking a device whose owner simply forgot the PIN. *Control:* Chapter 12. *Test:* measured wall-clock delay across a failure sequence.

3.4 Integrity

R-08 Every persistent record is authenticated, and the authentication covers the record's address.

A record lifted from one slot and written into another must fail verification.

Rationale: prevents the record-swap attack described in Chapter 10. *Control:* Chapter 10. *Test:* slot-swap harness.

R-09 Persistent counters are monotonic and cannot be decreased through any interface.

Rationale: G-4. *Control:* Chapter 11. *Test:* attempted rollback via every command, plus flash-level rewrite followed by verification failure.

R-10 Every command is authenticated before it is acted upon, and the authentication binds the command to this device and this session.

Rationale: G-5. *Control:* Chapters 14 and 15. *Test:* cross-device replay and same-device replay harness.

3.5 Behaviour under stress

R-11 Any error, at any stage, produces the same externally observable response.

One error code, one response length, one timing profile. The device does not tell the attacker *why* it refused.

Rationale: error differentiation is an oracle, and oracles are how protocols fall. *Control:* Chapter 16. *Test:* differential timing and response comparison across every distinct failure cause.

R-12 No input of any length or content causes a memory-safety fault.

Rationale: assumption five of the threat model. *Control:* Chapters 16, 19, and 20. *Test:* coverage-guided fuzzing of the parser on host, plus on-target sanity runs.

R-13 An interrupted write leaves persistent state either fully old or fully new, never partial.

Rationale: A-3 can cut power at any instant, and a torn record is an undefined state. *Control:* Chapter 10. *Test:* automated power-cut sweep across the write window.

R-14 The device cannot be permanently disabled by any sequence of protocol messages.

Rationale: G-6, limited to remote attackers. *Control:* Chapter 12's backoff design. *Test:* long-running adversarial command sequence.

3.6 Side channel

R-15 No comparison involving a secret branches on secret data or terminates early.

Rationale: A-2 measures timing to single-cycle resolution on a chip this simple. *Control:* Chapter 18. *Test:* statistical timing analysis over many trials with varying inputs.

R-16 No memory access address depends on secret data.

Rationale: closes cache and bus-timing leaks. On a cacheless M0+ the risk is lower than on a large core, but the discipline costs nothing and travels with the code when it is ported. *Control:*

Chapter 18.

The point

Sixteen requirements. Every chapter that follows exists to satisfy at least one of them, and every one of them is testable in a way I could demonstrate to somebody in a lab in under an hour. That is the bar a requirement has to clear to be worth writing down.

PART II

The Substrate

The Hardware You Actually Have

4.1 Reading a part with security eyes

When most people open a microcontroller datasheet they look for peripherals and clock speed. When you are building an HSM you are looking for something else: you are looking for the list of things the silicon will do for you that you would otherwise have to do badly in software, and the list of things the silicon will not do for you at all.

This chapter is that second list, written honestly. It is the most important chapter in Part II because every gap in it becomes a firmware obligation later, and firmware obligations are where bugs live.

4.2 The core: ARM Cortex-M0+

The Cortex-M0+ is the smallest useful ARM core. It executes the Thumb instruction set with a handful of Thumb-2 additions, it has a two-stage pipeline, and it is designed for low power and small die area rather than performance. For our purposes the important properties are these.

4.2.1 What helps us

There is no cache.

This is the single most useful security property of the part. Cache-timing attacks, which are the bane of AES implementations on application processors, simply do not apply. A table lookup takes the same time regardless of which entry you touch. That does not make me careless about **R-16**, but it removes an entire attack class for free.

There is no branch predictor and no speculative execution.

Spectre-class attacks have no foothold. Instruction timing is close to deterministic and can be reasoned about from the instruction timing table in the technical reference manual.

Two stack pointers exist.

Main Stack Pointer and Process Stack Pointer. A larger design would use these to separate a privileged kernel from unprivileged application code. I discuss below why I did not, and what you would gain if you did.

The instruction set is tiny.

Fewer instructions means fewer surprising behaviours, and it means that when you drop to assembly for a constant-time routine, you can actually reason about every cycle.

4.2.2 What hurts us

Single-cycle multiply is optional and division is absent.

There is no hardware divide instruction. Any division compiles into a library call whose execution time depends on the operand values. If a secret ever reaches a division, you have a timing leak that no amount of careful C will fix. Chapter 18 covers this. The practical rule is short: **secrets never participate in division or modulo.**

The Memory Protection Unit is optional, and where present it is minimal.

Even in the best case you get a small number of coarse regions. This is not enough to build a real privilege separation around, and I say more about that below.

No hardware stack limit register.

The Cortex-M33 and above have `MSPLIM`. The M0+ does not. A runaway stack silently walks into your globals and corrupts them, and the first symptom is a security control returning the wrong answer. Chapter 5 covers the guard-band technique that partially compensates.

Interrupt latency is short but not constant.

Late-arriving interrupts and tail-chaining make exact cycle counts hard to guarantee in the presence of interrupts. My answer is to disable interrupts across security-critical sections, which is only viable because the device does one thing at a time.

Get this wrong and the device is broken

The absence of a divide instruction is the trap that catches people. You write a perfectly reasonable line like `index = secret_value % table_size;` and you have just leaked information through the runtime of the software division routine. The leak is real and measurable on this class of part. Search your codebase for `/` and `%` applied to anything secret-derived, and replace them with masking against a power-of-two, or restructure so the operation is unnecessary.

4.3 Memory: how much and of what kind

The exact figures vary by device variant, so I will speak in the terms that matter architecturally rather than quoting numbers that may not match the part in your hand.

4.3.1 SRAM

You have tens of kilobytes. That sounds like plenty until you remember that everything is static: no heap, fixed buffers sized for the worst case, and every one of those buffers costs you permanently. My budget discipline is in Chapter 5.

The security-relevant property of SRAM is that it loses its contents when power is removed, but *not instantly and not completely*. Data remanence in SRAM is real. At room temperature the decay is fast enough that it is not a practical attack path for our threat model, but if you are designing for A-4 and a well-funded attacker who can chill the die, you should know the phenomenon exists and that the mitigation is to overwrite secrets rather than trusting power-down to clear them. Which is exactly what **R-03** demands anyway.

4.3.2 Flash

On-chip flash, directly memory-mapped and executed from in place. Three properties dominate every design decision in Part IV:

1. **Writing turns ones into zeroes. Erasing turns zeroes back into ones, and only a whole sector at a time.** You cannot rewrite a byte in place. You can only clear more bits, or erase the sector containing it.
2. **Erase is slow and erase is where power loss hurts.** A sector erase takes milliseconds. If power is cut in the middle, the sector's contents are undefined.
3. **Endurance is finite.** Tens of thousands of erase cycles per sector, and a failed attempt counter that erases on every write will exhaust that faster than you expect.

Those three facts, taken together, are the reason Chapter 10's storage design looks the way it does. A naive design that reads a counter, increments it, and writes it back is wrong on all three counts: it cannot write in place, it is not power-safe, and it burns the sector.

4.4 The AES accelerator

The part gives you a hardware AES engine, referred to as AESADV in the TI documentation. What matters is not its brand name but its contract, so let me state the contract in terms you can port:

- You load a key into a key register set.
- You select a mode, ECB or CBC, and a direction, encrypt or decrypt.
- You feed it 16-byte blocks and it produces 16-byte blocks, several times faster than a software implementation on this core would manage.
- It maintains chaining state internally for CBC mode, so consecutive blocks need no software involvement beyond feeding them.

4.4.1 Why using it is a security decision, not a performance one

The speed is welcome, but that is not why I use it. I use it because a software AES on a small core is a power-analysis target. Software AES executes a long sequence of table lookups and XORs whose power consumption correlates with the key bytes, and extracting a key from a software AES trace on a microcontroller is a well-established exercise that undergraduates complete in an afternoon with the right toolkit.

A hardware engine is not immune, but it is a substantially harder target. It computes rounds in dedicated logic, in parallel, in fewer cycles, with less of the key-dependent structure exposed in the power envelope. It is a real improvement and it costs me nothing but a peripheral driver.

The point

Use the hardware crypto engine even if you do not need the speed. The reason is side-channel resistance, not throughput. This flips the usual embedded reasoning, where accelerators are a performance optimisation you add if profiling demands it.

4.4.2 What the engine does not give you

It does not give you SHA-256, so HMAC is entirely software. It does not give you a MAC mode, so authentication is entirely software. And it does not give you a random number generator. That last one is Chapter 9 and it is the hardest problem in the whole design.

What else I could have done

I could have implemented AES in software with a bitsliced, constant-time construction, which is genuinely resistant to both timing and, with masking, to first-order power analysis. The reasons I did not: bitsliced AES on an M0+ costs a lot of code space and a lot of cycles, masking costs more of both, and correctness is hard to get right and harder to verify. Given that a hardware engine was sitting there unused, spending my scarce verification effort on a hand-rolled cipher would have been poor allocation. If I had *no* hardware engine, bitslicing is exactly what I would do, and I would allocate a week to testing it against the full NIST vector set.

4.5 DMA, and why it earns its place

The Direct Memory Access controller moves bytes between memory and peripherals without the CPU touching them. Pairing it with the AES engine means a block of data goes from a source

buffer, through the engine, into a destination buffer, with the core doing nothing but waiting.

Three benefits, in order of how much I care:

1. **The core never loads plaintext or ciphertext into registers.** Fewer places for data to sit, and one fewer power-analysis surface, because the register file and the data bus around the core are not toggling with your data.
2. **Timing becomes independent of data content,** because the transfer is a fixed number of bus cycles regardless of the bytes moving.
3. **It is faster,** which matters least but is not nothing.

Get this wrong and the device is broken

DMA is a bus master. It does not respect your language's memory model, and on a part with no MPU it does not respect anything else either. A DMA descriptor with a wrong length or a wrong destination address will happily scribble over your key material, your stack, or your policy record, and it will do so silently. Every descriptor must be constructed from constants and validated lengths, never from anything an attacker influenced, and the destination must be a statically allocated buffer whose size the compiler knows. I treat DMA descriptor construction with the same suspicion I treat the parser.

4.6 The gap list

Here is the honest accounting promised at the start of the chapter. On the left, what a purpose-built secure element would give you. On the right, what we do instead.

Ideal feature	Present here?	What we do instead
Hardware TRNG	Not dependable	Entropy pool from multiple weak sources, hashed; Ch. 9
Secure key storage	No	Root secrets in flash, accepted A-4 exposure; Ch. 8
Monotonic counter cell	No	Software counter in append-only flash log; Ch. 11
Tamper mesh	No	Not defended; documented limit
Memory protection unit	Minimal at best	Static allocation, guard bands, no heap; Ch. 5
Hardware stack limit	No	Guard band with known pattern, checked at boot and per command; Ch. 5
Fault detection logic	No	Redundant checks and control-flow counters; Ch. 17
Secure boot ROM	No	Not defended; production debug-lock only; Ch. 24
AES engine	Yes	Used for all bulk symmetric work; Ch. 7
SHA engine	No	Software SHA-256, constant time by construction; Ch. 6
DMA	Yes	Paired with AES; Ch. 7

Nine gaps, two gifts. That ratio is normal for a general-purpose microcontroller, and it is exactly why “we used an MCU as an HSM” is a sentence that should always be followed by a threat model.

What Bare Metal Actually Costs

5.1 The phrase, unpacked

“Bare metal” means there is no operating system. There is no scheduler, no memory manager, no driver framework, no libc initialisation you did not write, and no safety net of any kind. When the chip comes out of reset it starts executing your code, and everything that happens afterwards is your responsibility.

People say this like it is a burden. For a security device it is mostly a gift, and I want to explain why before I explain the costs.

The point

An operating system is a large, complex, attacker-reachable body of code that you did not write and cannot fully audit. Removing it removes that. On a device whose entire purpose is to be small and trustworthy, the absence of an OS is a security feature, not a hardship.

The costs are real though, and there are four of them.

5.2 Cost one: you write the boot sequence

When the core comes out of reset it does two things. It loads the initial stack pointer from the first word of the vector table, and it loads the reset handler address from the second word, and it jumps there. Everything after that is yours.

The reset handler must, in order:

1. Set up the clock tree, because the part boots on a slow internal oscillator and every timing assumption you make later depends on the final clock configuration.
2. Copy initialised data from flash to SRAM, because `.data` lives in flash at rest and must be in RAM to be writable.
3. Zero the `.bss` section, because C guarantees uninitialised statics are zero and nobody does that for you.
4. Install the guard bands described later in this chapter.
5. Run the boot-time self tests from Chapter 7.
6. Call `main`.

```

1  extern uint32_t _sidata, _sdata, _edata, _sbss, _ebss;
2  extern uint32_t _stack_guard, _estack_guard;
3
4  void Reset_Handler(void)
5  {
6      /* 1. Copy .data from its load address in flash to its run address in RAM.
7       * Until this loop finishes, any initialised global holds garbage. */
8      uint32_t *src = &_sidata;
9      for (uint32_t *dst = &_sdata; dst < &_edata; ) {
10         *dst++ = *src++;
11     }
12
13     /* 2. Zero .bss. The C standard promises this; the hardware does not. */
14     for (uint32_t *dst = &_sbss; dst < &_ebss; ) {
15         *dst++ = 0u;

```

```

16     }
17
18     /* 3. Paint the stack guard band with a known pattern so that overflow
19      * is detectable later. See section on guard bands. */
20     for (uint32_t *g = &_sstack_guard; g < &_estack_guard; ) {
21         *g++ = 0xDEADBEEFu;
22     }
23
24     /* 4. Clock configuration. Everything timing-related depends on this,
25      * including the flash wait states, so it comes before any flash write. */
26     clock_init();
27
28     /* 5. Known-answer tests on the crypto engine. If these fail we never
29      * reach main, because a broken AES engine that silently returns
30      * plaintext is the worst failure mode available to us. */
31     if (!selftest_all()) {
32         fatal_halt(); /* does not return */
33     }
34
35     main();
36
37     /* main must never return. If it does, that is a bug, and the safe
38      * response is to stop rather than to fall through into whatever
39      * bytes follow in flash. */
40     fatal_halt();
41 }

```

Listing 5.1: Reset handler skeleton. Order matters: nothing that touches a global may run before .data and .bss are ready.

Get this wrong and the device is broken

Notice that `selftest_all` runs before `main` and that its failure path halts. The tempting alternative is to run self tests inside `main` and set a flag. Do not. A flag can be glitched. A halt that never reaches application code cannot be. The general principle: **if a check must pass for the device to be safe, arrange the control flow so that failing the check makes the unsafe code unreachable, rather than making it conditional.**

5.3 Cost two: you own every byte of memory

There is no allocator. There will never be an allocator. This is a rule I enforce absolutely, and here is the reasoning in full because people push back on it.

Dynamic allocation on an embedded security device gives you three problems and no benefits you cannot get another way:

- **Failure at an inconvenient time.** `malloc` can return null, which means every call site needs an error path, which means more code, which means more paths an attacker can steer you down.
- **Fragmentation makes behaviour input-dependent.** The same sequence of operations succeeds or fails depending on history. That is a denial-of-service vector handed to the attacker for free.
- **Freed memory holds secrets.** A key in a freed block is a key sitting in RAM until something happens to overwrite it. Now every free of every buffer that might have touched a secret needs a scrub, and you will miss one.

The alternative is a static memory map computed at build time. Every buffer is a file-scope array sized for the worst case the protocol allows. The linker tells you the total, and if it does

not fit, you find out at build time rather than in the field.

```

1  /* Every one of these is sized from a protocol maximum defined in one
2  * header. If the protocol grows, the build breaks loudly here rather
3  * than overflowing quietly at runtime. */
4
5  static uint8_t g_rx_frame[HSM_MAX_FRAME_LEN];
6  static uint8_t g_tx_frame[HSM_MAX_FRAME_LEN];
7  static uint8_t g_scratch[HSM_MAX_PAYLOAD_LEN];
8
9  /* Key material gets its own section so the linker map makes it obvious
10 * where secrets live, and so we can scrub the whole region in one call. */
11 static uint8_t g_session_key[32] __attribute__((section(".secrets")));
12 static uint8_t g_derived_key[32] __attribute__((section(".secrets")));
13
14 _Static_assert(HSM_MAX_PAYLOAD_LEN <= HSM_MAX_FRAME_LEN,
15               "payload cannot exceed frame");

```

Listing 5.2: Static buffer discipline. Sizes come from protocol constants so that a protocol change forces a rebuild, not a runtime surprise.

The `.secrets` section is a small trick with a large payoff. Putting all key material in one contiguous linker section means the scrub routine is a single loop over a known range, the linker map shows you at a glance exactly how many bytes of secret you are carrying, and a reviewer can check that no new secret was declared outside the section by reading one map file.

5.4 Cost three: you have no stack protection

Without an `MSPLIM` register, stack overflow is silent. The stack grows down, your globals sit below it, and the first sign of trouble is a variable changing value for no reason. On a security device that variable might be the one holding “authenticated: yes.”

The compensation is a guard band: a region of known bytes placed between the bottom of the stack and the top of your data, painted at boot, and checked at points where you can still do something about it.

```

1  #define GUARD_PATTERN 0xDEADBEEFu
2
3  /* Check called at the top of the main command loop, before any parsing.
4  * If the guard is damaged we cannot trust any global, including the
5  * authentication flag, so the only safe action is to reset. */
6  static bool stack_guard_intact(void)
7  {
8      for (const uint32_t *g = &_ssstack_guard; g < &_estack_guard; g++) {
9          if (*g != GUARD_PATTERN) {
10             return false;
11          }
12      }
13      return true;
14  }
15
16  void command_loop(void)
17  {
18      for (;;) {
19          if (!stack_guard_intact()) {
20              /* Do not try to recover, log, or report. State is unknown.
21              * Reset is the only response we can reason about. */
22              system_reset();
23          }
24          handle_one_command();
25      }
26  }

```

Listing 5.3: Stack guard band. Cheap, imperfect, and far better than nothing.

Be clear-eyed about what this buys. It detects overflow *after* it happened, at the next check. It does not prevent the corruption. It is a smoke alarm, not a fire door. The actual defence against stack overflow is bounded recursion (we have none), bounded local buffers (we have those), and a measured worst-case stack depth, which Chapter 19 shows how to compute.

What else I could have done

On a part with a usable MPU, you place an unmapped region below the stack and get a hardware fault the instant the stack crosses it. That is a fire door: it stops the corruption rather than reporting it afterwards. If your part has an MPU with even four regions, spend one on this. I did not have that luxury and I want to be clear that the guard band is the weaker option, chosen because it was the available option.

5.5 Cost four: you decide what happens when things go wrong

There is no kernel to kill your process. A hard fault, by default, lands in a handler that loops forever, and a device that hangs silently is indistinguishable from a device that was attacked successfully.

My policy, applied everywhere:

Condition	Response	Why
Hard fault	Scrub secrets, reset	Unknown state; anything else risks acting on corruption
Guard band damaged	Reset immediately	Globals untrustworthy, including auth state
Self test failure	Halt permanently	A broken crypto engine must not be used, ever
Parse error	Generic error, stay up	Attacker-triggerable; must not become a DoS
Auth failure	Generic error, count it	Expected in normal use
Flash write failure	Retry once, then halt	Silent storage failure is worse than a dead device

The point

The distinction that matters is between conditions an attacker can trigger at will and conditions that indicate the device is genuinely broken. Attacker-triggerable conditions must never halt the device, or you have handed them [R-14](#). Genuine brokenness must always halt it, because a security device operating in an unknown state is worse than no device.

5.6 The main loop, in full

Everything above assembles into a main loop that is deliberately boring. Boring is the goal. A security state machine you can hold in your head is a security state machine you can reason about.

```
1 | int main(void)
```

```

2  {
3      storage_init(); /* mount the flash log, replay to current state */
4      session_reset(); /* unauthenticated, no keys, clean slate */
5      link_init(); /* bring up the serial interface last, because
6                   * after this point the attacker can talk to us */
7
8      for (;;) {
9          if (!stack_guard_intact()) {
10             system_reset();
11         }
12
13         frame_t f;
14         if (!link_receive_frame(&f)) { /* bounded, timeout-limited */
15             respond_generic_error();
16             continue;
17         }
18
19         command_t cmd;
20         if (!parse_command(&f, &cmd)) { /* total; never faults */
21             respond_generic_error();
22             continue;
23         }
24
25         if (!session_check_replay(&cmd)) { /* R-10 */
26             respond_generic_error();
27             continue;
28         }
29
30         if (!permission_allows(&g_session, cmd.opcode)) { /* R-05 */
31             respond_generic_error();
32             continue;
33         }
34
35         execute_command(&cmd); /* the only place keys live */
36         scrub_secrets(); /* R-03 */
37     }
38 }

```

Listing 5.4: The entire top level of the firmware. If this grows past one screen, something has gone wrong architecturally.

Read the order of the guards once more, because it is the same order as the diagram in Chapter 1 and it is not accidental. Receive, parse, replay-check, permission-check, and only then execute. Each stage assumes the previous one succeeded and validates nothing the previous stage already validated, which means each stage is small enough to review completely. And every failure path funnels into the same generic error, satisfying **R-11**.

PART III

The Cryptographic Core

Choosing the Primitives

6.1 The shortest possible list

The device uses three cryptographic primitives and nothing else:

Primitive	Instantiation	Job
Block cipher	AES-256 in CBC mode	Confidentiality of bulk data
Hash	SHA-256	Building block for everything below
MAC	HMAC-SHA-256	Integrity and authentication of everything
KDF	HKDF over HMAC-SHA-256	Turning one secret into many

Four rows, but really two primitives, because HMAC is built from SHA-256 and HKDF is built from HMAC. The entire cryptographic surface of this device is one block cipher and one hash function.

The point

A small primitive set is a security property in itself. Every additional algorithm is more code, more test vectors, more chances for a subtle implementation error, and more surface for an attacker to study. If you can do the job with two primitives, do the job with two primitives.

6.2 Why symmetric only

There is no public key cryptography anywhere in this design. No RSA, no elliptic curves, no key agreement, no signatures. That is a large omission and it deserves a real justification rather than a shrug.

6.2.1 The case for leaving it out

The trust topology does not need it.

Public key cryptography solves the problem of establishing trust between parties who have never met. Our device and its provisioning authority *have* met, at manufacture, in a controlled environment. A shared secret was installed then. Every subsequent trust relationship can be derived from that shared secret. When you already have a shared secret, asymmetric cryptography is solving a problem you do not have.

Bignum arithmetic on a Cortex-M0+ is expensive and dangerous.

An elliptic curve scalar multiplication on this core takes a meaningful fraction of a second in a naive implementation and requires careful, constant-time, well-tested code to be safe. Every one of those adjectives is a place to fail. The multiply-heavy inner loops are also a much richer power-analysis target than the hardware AES engine.

Code space is finite.

A constant-time X25519 or P-256 implementation is measured in kilobytes of code plus stack. On a part where I am counting bytes, that displaces things I need more.

6.2.2 The case against, stated fairly

Symmetric-only has a genuine cost and I will not pretend otherwise. It means **compromise of the provisioning authority's root secret compromises every device it ever provisioned**, retroactively and completely. With public key cryptography you can arrange things so that a compromise of the signing authority allows the attacker to authorise *new* things but does not decrypt *past* traffic. That property, forward secrecy against authority compromise, is real and valuable and I do not have it here.

What else I could have done

The design I would reach for if the constraints were looser: an Ed25519 verification key burned into the device at manufacture, used to verify signed capability grants, combined with X25519 ephemeral key agreement for session keys. That gives forward secrecy, allows the authority to be offline, and lets you rotate authority keys without touching devices. It costs perhaps six to eight kilobytes of carefully written constant-time code and a great deal of test effort. On a bigger part, in a product with a real update path, that is the right answer. Here, given the threat model in Chapter 2 and the fact that a compromised provisioning environment already loses everything under assumption one, the extra complexity buys less than it costs.

6.3 Why AES-256 and not AES-128

AES-128 is not broken and will not be broken. The honest reason for 256 is not that 128 is insufficient against a classical attacker, because it is not.

The reasons are:

1. **Grover's algorithm halves the effective security level of a symmetric key against a quantum adversary.** AES-256 lands at 128 bits of post-quantum margin; AES-128 lands at 64, which is uncomfortably close to reachable. This is a long-horizon concern and I am aware it borders on the theoretical, but the cost of buying the margin is small.
2. **Cost on a hardware engine is four extra rounds.** That is fourteen rounds instead of ten, a 40 percent increase in a computation that is already fast and not on any critical path. Nothing in the device's behaviour changes measurably.
3. **Key material is not scarce.** The key hierarchy produces 256-bit outputs from HKDF-SHA-256 naturally. Truncating them to 128 bits would be extra code doing extra work to end up with less security.

6.4 Why CBC, honestly

This is the choice in the document I am least comfortable defending, so I will defend it carefully and then say where it is weak.

CBC mode encrypts each block after XORing it with the previous ciphertext block, starting from an initialisation vector. It provides confidentiality and nothing else.

6.4.1 The reasons I used it

The hardware engine implements CBC natively, maintaining the chaining state internally. That means the core never touches intermediate values, which is exactly the property I argued for in Chapter 4. An engine that only does ECB would force me to do the chaining XOR in software, which puts plaintext and ciphertext through the CPU registers, which is precisely the exposure

the hardware engine was supposed to remove.

6.4.2 The reason it is not enough on its own

Get this wrong and the device is broken

CBC provides no integrity whatsoever. An attacker who can modify ciphertext can make predictable, controlled changes to the decrypted plaintext. Specifically, flipping a bit in ciphertext block n flips the corresponding bit in plaintext block $n + 1$, at the cost of scrambling block n entirely. If you decrypt CBC ciphertext and act on the result without verifying a MAC first, your device is an oracle and you will be taken apart by it. CBC without a MAC is not a mode. It is a mistake with a name.

So every use of CBC in this design is wrapped in HMAC, in encrypt-then-MAC order, which I explain in the next section because the order is not arbitrary.

6.4.3 Encrypt-then-MAC, and why the order is not negotiable

There are three ways to combine encryption and a MAC. Only one of them is generically safe.

Construction	What you send	Verdict
Encrypt-and-MAC	$E(P) \parallel \text{MAC}(P)$	Broken. The MAC of plaintext can leak plaintext equality.
MAC-then-encrypt	$E(P \parallel \text{MAC}(P))$	Fragile. You must decrypt before you can verify, which means attacker-chosen ciphertext enters your unpadding logic. This is the shape of the Lucky 13 and padding-oracle families.
Encrypt-then-MAC	$E(P) \parallel \text{MAC}(E(P))$	Correct. Verify first; if the MAC fails, the ciphertext never reaches the cipher.

Encrypt-then-MAC is the only one where a forged message is rejected *before* any attacker-controlled data touches your decryption or padding code. That is the entire point. The MAC becomes a gate, and everything downstream of the gate only ever sees data that a party holding the key produced.

The point

Verify the MAC before you decrypt. Always. Not as an optimisation, not as a preference, but because it is the only ordering in which your padding removal code is unreachable by an attacker.

```

1  /* Returns true and fills out_plain on success.
2  * Returns false, and writes nothing to out_plain, on any failure.
3  *
4  * Note the single exit discipline: every failure path produces an
5  * identical return value and identical timing to within measurement
6  * noise, because we always run the full MAC computation before deciding.
7  */
8  bool secure_open(const uint8_t enc_key[32],
9                  const uint8_t mac_key[32],
10                  const uint8_t *iv, /* 16 bytes */
11                  const uint8_t *ct, size_t ct_len,
```

```

12     const uint8_t *tag, /* 32 bytes */
13     uint8_t *out_plain, size_t *out_len)
14 {
15     if (ct_len == 0u || (ct_len % 16u) != 0u || ct_len > HSM_MAX_PAYLOAD_LEN) {
16         return false; /* structural, not secret */
17     }
18
19     /* 1. Recompute the tag over IV || ciphertext. The IV is included
20      * because an unauthenticated IV lets an attacker control the
21      * first plaintext block byte for byte. */
22     uint8_t computed[32];
23     hmac_sha256_ctx ctx;
24     hmac_sha256_init(&ctx, mac_key, 32u);
25     hmac_sha256_update(&ctx, iv, 16u);
26     hmac_sha256_update(&ctx, ct, ct_len);
27     hmac_sha256_final(&ctx, computed);
28
29     /* 2. Constant-time comparison. See Chapter 18 for why memcmp is
30      * unacceptable here. */
31     bool tag_ok = ct_eq(computed, tag, 32u);
32     secure_zero(computed, sizeof computed);
33
34     if (!tag_ok) {
35         return false; /* ciphertext never reaches the cipher */
36     }
37
38     /* 3. Only now do we decrypt. Everything below this line is operating
39      * on data that someone holding mac_key produced. */
40     aes256_cbc_decrypt(enc_key, iv, ct, ct_len, out_plain);
41
42     /* 4. Padding removal, also constant time, also only reachable
43      * after authentication. */
44     return pad_strip_ct(out_plain, ct_len, out_len);
45 }

```

Listing 6.1: The only decryption entry point in the firmware. There is deliberately no way to decrypt without passing through the MAC gate.

What else I could have done

AES-GCM or AES-CCM would give me authenticated encryption in one primitive, with one key instead of two, and with a well-analysed security proof. If the hardware engine had supported GCM I would have used it without hesitation. Two things pushed me away: the engine did not offer it, so I would be implementing GHASH in software on a core with no carryless multiply, which is both slow and fiddly to make constant-time; and GCM fails catastrophically on nonce reuse in a way CBC-with-HMAC does not, which on a device with a weak entropy source (Chapter 9) is a genuine hazard. **If you are porting this design to a part with a GCM-capable engine and a real TRNG, use GCM.** It is the better construction and my reasons for avoiding it are specific to this hardware.

6.5 HMAC, and why not a bare hash

A common beginner error is to authenticate with $\text{SHA-256}(K \parallel M)$. On SHA-256, which is a Merkle–Damgård construction, this is vulnerable to length extension: an attacker who knows M and the digest, without knowing K , can compute a valid digest for $M \parallel \text{padding} \parallel M'$ for a suffix of their choosing.

HMAC exists precisely to fix this. Its structure is:

$$\text{HMAC}(K, M) = H((K \oplus \text{opad}) \parallel H((K \oplus \text{ipad}) \parallel M))$$

The outer hash breaks length extension because the attacker cannot continue a computation whose intermediate state they do not have.

```

1  #define SHA256_BLOCK 64u
2  #define SHA256_DIGEST 32u
3
4  void hmac_sha256_init(hmac_sha256_ctx *c, const uint8_t *key, size_t key_len)
5  {
6      uint8_t k0[SHA256_BLOCK];
7      secure_zero(k0, sizeof k0);
8
9      if (key_len > SHA256_BLOCK) {
10         /* Keys longer than the block are hashed down first. */
11         sha256(key, key_len, k0);
12     } else {
13         memcpy(k0, key, key_len); /* remainder stays zero-padded */
14     }
15
16     uint8_t pad[SHA256_BLOCK];
17     for (size_t i = 0u; i < SHA256_BLOCK; i++) {
18         pad[i] = k0[i] ^ 0x36u; /* ipad */
19     }
20     sha256_init(&c->inner);
21     sha256_update(&c->inner, pad, SHA256_BLOCK);
22
23     for (size_t i = 0u; i < SHA256_BLOCK; i++) {
24         pad[i] = k0[i] ^ 0x5Cu; /* opad */
25     }
26     sha256_init(&c->outer);
27     sha256_update(&c->outer, pad, SHA256_BLOCK);
28
29     /* The key is gone from our stack before we return. R-03. */
30     secure_zero(k0, sizeof k0);
31     secure_zero(pad, sizeof pad);
32 }
33
34 void hmac_sha256_update(hmac_sha256_ctx *c, const uint8_t *m, size_t n)
35 {
36     sha256_update(&c->inner, m, n);
37 }
38
39 void hmac_sha256_final(hmac_sha256_ctx *c, uint8_t out[SHA256_DIGEST])
40 {
41     uint8_t ih[SHA256_DIGEST];
42     sha256_final(&c->inner, ih);
43     sha256_update(&c->outer, ih, SHA256_DIGEST);
44     sha256_final(&c->outer, out);
45     secure_zero(ih, sizeof ih);
46     secure_zero(c, sizeof *c);
47 }

```

Listing 6.2: HMAC-SHA-256. Short, and worth reading line by line once in your life.

Two details in that listing are security-relevant rather than cosmetic. The zero-padding of a short key must be genuine zeroing, not whatever was on the stack, or two different keys can produce the same internal state. And the context itself is scrubbed in `final`, because the context holds key-derived state and callers forget to clean up.

Driving the AES Engine

7.1 The contract, restated portably

I described the engine’s behaviour in Chapter 4. Here I want to write the driver in a way that survives being ported, which means expressing it as a small interface with a hardware implementation behind it, rather than sprinkling register writes through the cryptographic code.

```

1  typedef enum { AES_ENCRYPT, AES_DECRYPT } aes_dir_t;
2
3  /* Load a 256-bit key into the engine. The engine holds it until the
4   * next load or until aes_clear(). */
5  void aes256_load_key(const uint8_t key[32]);
6
7  /* Set the CBC chaining value. Must be called before each independent
8   * message; forgetting it chains one message into the next, which is a
9   * correctness bug that looks like a working device in testing. */
10 void aes256_set_iv(const uint8_t iv[16]);
11
12 /* Process len bytes, which must be a multiple of 16, through the engine
13  * in CBC mode. Uses DMA; blocks until complete. */
14 void aes256_cbc(aes_dir_t dir, const uint8_t *in, uint8_t *out, size_t len);
15
16 /* Wipe the engine’s key registers. Called on every path out of any
17  * operation that used a key. */
18 void aes256_clear(void);

```

Listing 7.1: The AES interface. Four functions. Everything above this line is portable; everything below it is silicon-specific.

Get this wrong and the device is broken

`aes256_clear` is not optional housekeeping. The engine’s key registers are memory-mapped, and on a part with no memory protection, any code that can read a peripheral address can read the key you left sitting in them. Leaving a key in the engine after an operation is exactly equivalent to leaving it in a global variable named `g_the_key`. Call `aes256_clear` on every exit path, including error paths, including the paths you added last week and forgot about. In the Rust implementation in Chapter 20 this is enforced by the type system rather than by discipline, and that is a large part of why that chapter exists.

7.2 The DMA pairing

Wiring DMA to the AES engine means configuring two channels: one that feeds the engine’s input register from your source buffer, and one that drains the engine’s output register into your destination buffer.

The sequence, in terms of what must be true at each step rather than which bit goes where:

1. Both channels are disabled and any pending flags are cleared. A stale flag from a previous transfer will cause the new transfer to appear complete instantly, and your “ciphertext” will be uninitialised buffer contents. This is a bug that passes a smoke test and fails silently.
2. The output channel is configured *first*, then the input channel. If you enable input first, the engine may produce a result before anything is waiting to collect it.

3. Both channels use fixed peripheral addresses and incrementing memory addresses, with a transfer width matching the engine’s register width.
4. The transfer count is derived from a validated length. It is never derived from a length field that arrived over the wire without bounds checking, and Chapter 16 is about why that sentence needs saying.
5. The core waits on the output channel’s completion, not the input channel’s. Input completing means all data was handed to the engine, not that the engine finished with it.

```

1 void aes256_cbc(aes_dir_t dir, const uint8_t *in, uint8_t *out, size_t len)
2 {
3     /* These are load-bearing. A DMA descriptor built from a bad length
4      * is an arbitrary memory write, and no later check can undo it. */
5     HSM_ASSERT(len != 0u);
6     HSM_ASSERT((len % 16u) == 0u);
7     HSM_ASSERT(len <= HSM_MAX_PAYLOAD_LEN);
8     HSM_ASSERT(buffer_is_in_static_ram(out, len));
9
10    aes_set_direction(dir);
11
12    dma_disable(DMA_CH_AES_OUT);
13    dma_disable(DMA_CH_AES_IN);
14    dma_clear_flags(DMA_CH_AES_OUT);
15    dma_clear_flags(DMA_CH_AES_IN);
16
17    /* Output channel first: somewhere to put results before results exist. */
18    dma_config(DMA_CH_AES_OUT,
19              /* src */ AES_DATA_OUT_ADDR, DMA_ADDR_FIXED,
20              /* dst */ (uintptr_t)out, DMA_ADDR_INCREMENT,
21              /* size */ len, DMA_WIDTH_WORD);
22
23    dma_config(DMA_CH_AES_IN,
24              /* src */ (uintptr_t)in, DMA_ADDR_INCREMENT,
25              /* dst */ AES_DATA_IN_ADDR, DMA_ADDR_FIXED,
26              /* size */ len, DMA_WIDTH_WORD);
27
28    dma_enable(DMA_CH_AES_OUT);
29    dma_enable(DMA_CH_AES_IN);
30
31    /* Wait on OUT. IN completing only means the engine was fed. */
32    dma_wait_complete(DMA_CH_AES_OUT);
33
34    dma_disable(DMA_CH_AES_IN);
35    dma_disable(DMA_CH_AES_OUT);
36 }

```

Listing 7.2: DMA-driven CBC. The assertions are compiled in for release builds, not just debug, because they are protecting a bus master.

7.3 Boot-time known-answer tests

Chapter 5 put `selftest_all` before `main` and said a broken engine is the worst failure mode available. Here is why: if the engine silently returns its input unchanged, every “encrypted” output is plaintext, and nothing else in the device notices. No MAC fails. No parse fails. The device works perfectly and broadcasts secrets.

The defence is a known-answer test with a NIST vector, run at every boot, whose failure halts the device.

```

1 static const uint8_t KAT_KEY[32] = {
2     0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
3     0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4

```

```

4  };
5  static const uint8_t KAT_IV[16] = {
6      0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07, 0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f
7  };
8  static const uint8_t KAT_PT[16] = {
9      0x6b,0xc1,0xbe,0xe2,0x2e,0x40,0x9f,0x96, 0xe9,0x3d,0x7e,0x11,0x73,0x93,0x17,0x2a
10 };
11 static const uint8_t KAT_CT[16] = {
12     0xf5,0x8c,0x4c,0x04,0xd6,0xe5,0xf1,0xba, 0x77,0x9e,0xab,0xfb,0x5f,0x7b,0xfb,0xd6
13 };
14
15 bool selftest_aes(void)
16 {
17     uint8_t buf[16];
18
19     aes256_load_key(KAT_KEY);
20     aes256_set_iv(KAT_IV);
21     aes256_cbc(AES_ENCRYPT, KAT_PT, buf, 16u);
22     if (!ct_eq(buf, KAT_CT, 16u)) { aes256_clear(); return false; }
23
24     aes256_set_iv(KAT_IV);
25     aes256_cbc(AES_DECRYPT, KAT_CT, buf, 16u);
26     if (!ct_eq(buf, KAT_PT, 16u)) { aes256_clear(); return false; }
27
28     /* An engine that passes both directions is doing AES. An engine that
29      * passes only encrypt might be echoing. Test both. */
30     aes256_clear();
31     secure_zero(buf, sizeof buf);
32     return true;
33 }

```

Listing 7.3: Known-answer test. The vector is from NIST SP 800-38A; use the published one, never one you generated with the same code you are testing.

Run the equivalent test for SHA-256 and HMAC. Three known-answer tests, a few hundred bytes of vectors, and a class of catastrophic silent failure is closed.

The Key Hierarchy

8.1 The problem a hierarchy solves

You have one secret installed at manufacture. You need many keys: one for encrypting data, a different one for authenticating it, different ones per user, different ones per operation, different ones per device.

The wrong answer is to use the one secret for everything. Key reuse across purposes is a well-worn source of breaks, because a construction that is safe when a key is used one way often stops being safe when the same key is also used another way. The classic shape of the failure is that an attacker gets to request operation A with a chosen input, and uses the result to forge operation B.

The right answer is a key derivation function: a one-way transformation that turns one secret plus some context into a new secret, such that knowing the output tells you nothing about the input and nothing about outputs derived with different context.

The point

One key, one purpose. If two things in your system need a key, they get two different derived keys, even if it feels wasteful. Derivation is cheap. Cross-purpose key reuse is how systems break.

8.2 HKDF in two paragraphs

HKDF, from RFC 5869, has two steps and the separation between them is the whole idea.

Extract takes input keying material that may be non-uniform, meaning it has entropy but is not evenly spread, and concentrates it into a fixed-length pseudorandom key:

$$\text{PRK} = \text{HMAC}(\text{salt}, \text{IKM})$$

Note that the salt is used as the HMAC key and the input material as the message, which reads backwards until you realise the salt is the thing playing the role of a randomiser.

Expand takes that pseudorandom key plus an application-specific info string and produces as many output bytes as you asked for:

$$T_i = \text{HMAC}(\text{PRK}, T_{i-1} \parallel \text{info} \parallel i), \quad T_0 = \varepsilon$$

The `info` parameter is the one that matters for us. It is the context. Two derivations with the same PRK and different `info` produce unrelated keys, and that is the property the entire hierarchy rests on.

```

1 void hkdf_extract(const uint8_t *salt, size_t salt_len,
2                  const uint8_t *ikm, size_t ikm_len,
3                  uint8_t prk[32])
4 {
5     static const uint8_t zero_salt[32] = {0};
6     if (salt == NULL || salt_len == 0u) {
7         salt = zero_salt; salt_len = 32u; /* RFC 5869 default */
8     }
9     hmac_sha256_ctx c;
10    hmac_sha256_init(&c, salt, salt_len);

```

```

11     hmac_sha256_update(&c, ikm, ikm_len);
12     hmac_sha256_final(&c, prk);
13 }
14
15 bool hkdf_expand(const uint8_t prk[32],
16                 const uint8_t *info, size_t info_len,
17                 uint8_t *okm, size_t okm_len)
18 {
19     if (okm_len > 255u * 32u) { return false; } /* RFC limit */
20
21     uint8_t t[32];
22     size_t t_len = 0u; /* T_0 is the empty string */
23     size_t done = 0u;
24     uint8_t counter = 1u;
25
26     while (done < okm_len) {
27         hmac_sha256_ctx c;
28         hmac_sha256_init(&c, prk, 32u);
29         hmac_sha256_update(&c, t, t_len); /* previous block */
30         hmac_sha256_update(&c, info, info_len); /* the context */
31         hmac_sha256_update(&c, &counter, 1u); /* the block index */
32         hmac_sha256_final(&c, t);
33         t_len = 32u;
34
35         size_t take = (okm_len - done < 32u) ? (okm_len - done) : 32u;
36         memcpy(okm + done, t, take);
37         done += take;
38         counter++;
39     }
40
41     secure_zero(t, sizeof t);
42     return true;
43 }

```

Listing 8.1: HKDF. Fifty lines, no allocation, no surprises.

8.3 The three tiers

Confirm this against your own build

I have documented the hierarchy here as three tiers, which is what my resume, my portfolio, and every piece of outreach I have sent says. There is an open question in Appendix F about whether the shipped firmware uses a deeper chain. Before I present this in a technical interview, I need to read the derivation code in my repository and settle it, because being wrong about the shape of my own key hierarchy in front of somebody who does this professionally is not a mistake I want to make. The three-tier structure below is architecturally correct and is what I would build again. The open item is purely whether the implementation added intermediate steps.

8.3.1 Tier 0: the root

ROOT_KEY is 32 bytes of randomness generated by the provisioning authority on a trusted machine, using that machine's cryptographic RNG. It is written into device flash exactly once and read by nothing except the derivation function.

Properties:

- It never appears on the wire. Not encrypted, not wrapped, not in any command.
- It is never used directly as an AES key or an HMAC key. Its only use is as input keying material to HKDF-Extract.

- It is per-device, or per-batch if the deployment model requires. Per-device is strictly better and costs only provisioning complexity.

8.3.2 Tier 1: the device key

$$\text{PRK}_{\text{dev}} = \text{HKDF-Extract}(\text{salt} = \text{DEVICE_ID}, \text{IKM} = \text{ROOT_KEY})$$

Binding the device identifier in as salt at this level means that even if two devices somehow shared a root, their tier-1 keys differ. It costs nothing and closes a whole class of provisioning mistakes.

PRK_{dev} lives in SRAM only, computed at boot, scrubbed at reset. It is never used as an operational key either. It is the input to tier 2.

8.3.3 Tier 2: the operational keys

This is where usable keys appear. Every operational key is:

$$K_{\text{purpose}} = \text{HKDF-Expand}(\text{PRK}_{\text{dev}}, \text{info} = \text{LABEL} \parallel \text{CONTEXT}, 32)$$

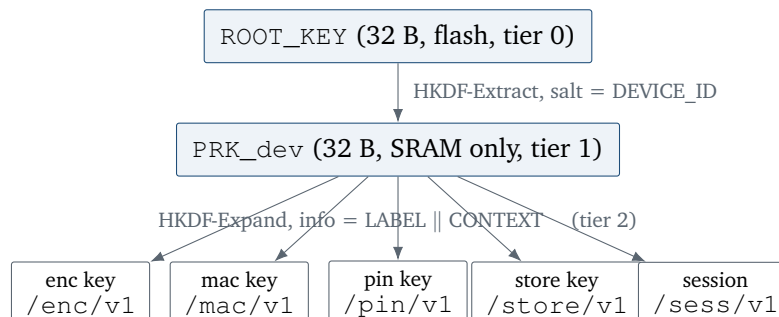
LABEL is a fixed ASCII string naming the purpose. **CONTEXT** is whatever binds the key to the specific operation: a user identifier, a counter value, a record slot number.

Key	LABEL	CONTEXT bound in
Data encryption	"neusense/enc/v1"	record slot, record version
Data authentication	"neusense/mac/v1"	record slot, record version
PIN verification	"neusense/pin/v1"	user id
Storage record MAC	"neusense/store/v1"	slot index, generation counter
Session binding	"neusense/sess/v1"	session nonce

The point

The encryption key and the authentication key for the same record are different keys, derived with different labels. This is the practical expression of “one key, one purpose,” and it is what makes the encrypt-then-MAC construction in Chapter 6 actually sound rather than merely well-shaped.

8.3.4 The whole hierarchy in one picture



8.4 What the hierarchy buys you

Compartmentalisation.

An attacker who somehow recovers one tier-2 key gets exactly that key's purpose, for exactly that context. They cannot climb back up, because HKDF is one-way, and they cannot move sideways, because a different context produces an unrelated key.

Revocation without rekeying.

Bind a generation counter into the context. Increment the counter, and every key derived under the old counter is dead, permanently, without touching the root. Chapter 11 builds revocation on exactly this.

No key storage problem.

Only the root is stored. Everything else is recomputed on demand in a few microseconds. Keys that are never stored cannot be stolen from storage.

Provisioning is one write.

The manufacturing step writes 32 bytes plus an identifier. Simple provisioning is provisioning that is done correctly.

What else I could have done

A flat scheme, with every operational key stored directly in flash, is simpler to reason about and avoids any derivation code. It is what I would do if key derivation itself were the risky part. It loses on every other axis: more secrets in flash, a harder provisioning step, no cheap revocation, and no compartmentalisation. **A deeper hierarchy**, four or five tiers with per-session and per-message layers, gives finer-grained compartments and is what protocols like Signal's do. It costs derivation work per message and more state to track. Three tiers is the point where I stopped getting security back for the complexity I was spending. That judgement is defensible either way, and if your device serves many concurrent users with strong isolation requirements between them, add the tier.

Where Randomness Comes From

9.1 The hardest problem in the document

Every other chapter has a clean answer. This one does not, and I want to be upfront about that, because randomness is where embedded security most often fails and the failures are invisible until somebody looks.

The device needs random bytes for initialisation vectors, for session nonces, and for challenge values. If those bytes are predictable, then IVs repeat, nonces repeat, and challenge-response becomes replayable. None of that produces a visible malfunction. The device works. It is just not secure, and nothing in your test suite tells you.

Get this wrong and the device is broken

The single most common catastrophic failure in embedded cryptography is a weak or repeating random number source. It has broken smart cards, routers, TLS libraries, and cryptocurrency wallets, repeatedly, in production, at scale. It is invisible in testing because the output looks like bytes and the protocol works. **If you take one practical lesson from this document, take this one: treat your entropy source as the thing most likely to be wrong, and test it specifically.**

9.2 What this part gives you, and what it does not

There is no dependable hardware TRNG I am willing to build on. That leaves gathering entropy from weak sources and conditioning it, which is the standard fallback and which must be done carefully to be worth anything.

9.2.1 The sources

Source	Quality	Notes
ADC on a floating pin	Low	Low bits are noisy; upper bits are not. Take low bits only.
Internal temperature sensor	Very low	Drifts slowly. Contributes almost nothing per read.
Oscillator drift	Low–medium	Sample a fast timer against a slow independent one; the jitter between them is real entropy.
Timing of external events	Medium	Inter-arrival times of host bytes. Attacker-influenced, so never trusted alone.
Uninitialised SRAM at boot	One-shot	Genuinely different per device and per power cycle, but only useful once, at boot.
Persistent seed in flash	Carried	Not a source. A memory. See below.

9.2.2 The construction

Weak sources are combined by hashing, never by concatenation or XOR alone, and the pool is carried across reboots so that entropy accumulates over the device's lifetime rather than starting from nothing at every power-on.

```

1  static uint8_t g_pool[32] __attribute__((section(".secrets")));
2  static uint32_t g_ctr;
3
4  /* Stir arbitrary bytes into the pool. Hashing the old pool together
5   * with the new data means adding a known value can never *reduce*
6   * the pool's entropy, which is why we never XOR sources in directly. */
7  static void pool_stir(const uint8_t *data, size_t len)
8  {
9      sha256_ctx c;
10     sha256_init(&c);
11     sha256_update(&c, g_pool, sizeof g_pool);
12     sha256_update(&c, data, len);
13     sha256_final(&c, g_pool);
14 }
15
16 void rng_init(void)
17 {
18     /* 1. Uninitialised SRAM, before anything overwrites it. This must be
19      * called from the reset handler before .bss zeroing to be useful. */
20     pool_stir(uninit_ram_snapshot(), UNINIT_SNAPSHOT_LEN);
21
22     /* 2. The seed carried from the last power cycle. Accumulated entropy
23      * from the device's whole life, not just this boot. */
24     uint8_t seed[32];
25     if (storage_read_seed(seed)) {
26         pool_stir(seed, sizeof seed);
27         secure_zero(seed, sizeof seed);
28     }
29
30     /* 3. Hardware noise, sampled many times because each sample is weak. */
31     for (int i = 0; i < 256; i++) {
32         uint8_t s[4];
33         s[0] = (uint8_t)adc_read_floating(); /* low bits only */
34         s[1] = (uint8_t)timer_jitter_sample();
35         s[2] = (uint8_t)temp_sensor_low_bits();
36         s[3] = (uint8_t)i;
37         pool_stir(s, sizeof s);
38     }
39
40     /* 4. Immediately write a fresh seed back, derived from the pool but
41      * not equal to it. If power is lost now, the next boot still
42      * inherits this boot's entropy. */
43     rng_persist_new_seed();
44 }
45
46 /* Output. Note that we never hand out the pool itself: we hand out
47  * HKDF-Expand of the pool with a counter, so that an attacker who
48  * somehow learns an output cannot roll the pool backwards. */
49 void rng_bytes(uint8_t *out, size_t len)
50 {
51     uint8_t info[8];
52     be32_write(info, g_ctr++);
53     be32_write(info + 4, (uint32_t)len);
54
55     uint8_t prk[32];
56     hkdf_extract(NULL, 0u, g_pool, sizeof g_pool, prk);
57     (void)hkdf_expand(prk, info, sizeof info, out, len);
58     secure_zero(prk, sizeof prk);
59
60     /* Ratchet the pool forward so that this output can never be
61      * reproduced, even by an attacker who dumps the pool one moment later. */

```

```

62 |     pool_stir((const uint8_t *)&g_ctr, sizeof g_ctr);
63 | }

```

Listing 9.1: Entropy pool. The design goal is that compromise of any one source, including the attacker-influenced one, does not compromise the output.

9.3 The rules I follow

1. **Never use a source directly.** Everything goes through the pool. Even a hardware TRNG, if I had one, would be stirred rather than trusted.
2. **Never trust attacker-influenced sources alone.** Host byte timing is stirred in because it cannot hurt, and is never counted toward the entropy estimate.
3. **Persist and accumulate.** A device three years into deployment should have far more entropy in its pool than one out of the box.
4. **Ratchet after every output.** Forward secrecy for the pool. An attacker who reads the pool now cannot compute what it produced ten minutes ago.
5. **Never let a nonce depend on the RNG being good if you can avoid it.** This is the important one and it deserves its own section.

9.4 Designing so that randomness matters less

The best defence against a weak RNG is to need it less. Two techniques do most of the work.

Counter-based IVs. Instead of a random IV, use a monotonically increasing counter, encrypted or derived. The counter is stored in the same anti-rollback structure as everything else in Chapter 11, so it never repeats, and it does not depend on entropy at all. CBC requires IVs to be unpredictable, not merely unique, so the counter is passed through the block cipher rather than used raw.

Synthetic nonces. Derive the nonce from the message content and a key, rather than sampling it. Two identical messages produce the same nonce, which is acceptable, and different messages produce different nonces without touching the RNG.

The point

Every place you can replace “sample randomness” with “derive deterministically from a counter and a key,” you have removed a dependency on the weakest component in an embedded system. Do this wherever the cryptography permits it.

What else I could have done

If the part had a certified TRNG with a health-test suite, I would use it as the primary source, still stirred through the pool, and I would run the continuous health tests from NIST SP 800-90B and treat a health-test failure as a halt condition. If I could add a component, an external secure element with a TRNG and a monotonic counter would solve this chapter and half of Chapter 11 for about a dollar. On a fixed platform I did not choose, the pool is the honest best effort, and I would rather present it as a considered compromise than dress it up as a solution.

PART IV

State That Survives Power Loss

The Secure Flash Filesystem

10.1 Why this is harder than it looks

The device needs to remember a handful of small things: how many times authentication has failed, which capabilities have been revoked, what the current generation counter is, and the entropy seed from Chapter 9. That is perhaps a few hundred bytes of state.

A first attempt looks like this: reserve a flash sector, define a struct, read it at boot, write it when it changes. That design is wrong in four separate ways, and walking through each one is the fastest route to understanding why the real design looks the way it does.

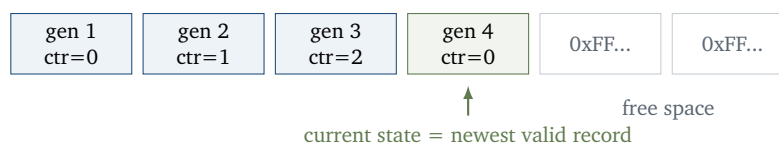
1. **You cannot write it.** Flash writes only clear bits. To change a byte from `0x05` to `0x06` you must erase the whole sector first, because that requires setting a bit.
2. **The erase is not atomic.** Cut power during the erase and the sector holds undefined bytes. Your failure counter is now whatever the silicon felt like, which might be `0xFF` and might be `0x00`. An attacker with A-3 capability will aim for exactly this.
3. **The sector wears out.** Erase before every write means one erase per failed PIN attempt. A patient attacker submits bad PINs until the sector dies, and now the device cannot record failures at all.
4. **Nothing stops modification.** An attacker who can write flash directly can set the counter to zero. Nothing in the struct detects it.

The point

Every one of those four problems has the same root cause: treating flash like RAM. Flash is not slow RAM. It is a write-once, erase-in-blocks, wears-out medium, and a storage design that does not respect that will fail in the field in ways that never appear on the bench.

10.2 The design: an append-only authenticated log

The structure that solves all four problems at once is a log. You never modify a record. You append a new one. The current state is whatever the newest valid record says.



Reading the state means scanning forward to the last record that authenticates. Writing means appending one record into erased space. Erase happens once per sector, when the sector fills, not once per write.

10.2.1 How this fixes each problem

Problem	How the log fixes it
Cannot write in place	We never write in place. We append into erased bytes, which is exactly what flash supports natively.
Erase is not atomic	A torn append leaves an invalid record, which fails its MAC and is ignored. The previous record is untouched and remains current.
Wear	One erase per sector-full of records instead of one per write. With small records that is a two to three order of magnitude improvement.
No integrity	Every record carries a MAC over its contents <i>and</i> its position. Chapter 8's storage key derives it.

10.3 The record format

Field	Bytes	Meaning
magic	2	0x4E53. Distinguishes a written record from erased 0xFF bytes.
version	1	Record format version. Lets a future firmware read old records.
type	1	Which state variable this record carries.
seq	4	Monotonic sequence number, big-endian. The newest valid record is the one with the highest <code>seq</code> .
payload	24	The state itself. Fixed size so that record boundaries are computable without parsing.
mac	32	HMAC-SHA-256 over all preceding fields <i>plus the record's own flash offset</i> .

Total 64 bytes, which is a convenient size: it aligns to flash write granularity on most parts, it divides sector sizes evenly, and it means record n lives at offset $64n$ with no bookkeeping.

10.3.1 Why the offset is in the MAC

This is the detail that people leave out and it is the one that matters most.

Get this wrong and the device is broken

If the MAC covers only the record contents, an attacker can copy a valid old record byte-for-byte into a later slot. The MAC still verifies, because the contents did not change. The record's `seq` is old, but if your scan logic takes “the last record with a valid MAC” rather than “the highest `seq`,” you have just been rolled back. Including the flash offset in the MAC input makes a record that is valid at offset X invalid at offset Y , and the copy attack dies immediately.

```

1 #define REC_SIZE 64u
2 #define REC_MAGIC 0x4E53u
3 #define REC_HDR_LEN 32u /* everything before the mac field */
4
5 static void record_mac(uint32_t flash_offset,
```

```

6         const uint8_t hdr[REC_HDR_LEN],
7           uint8_t out[32])
8     {
9         uint8_t store_key[32];
10        derive_storage_key(flash_offset, store_key); /* Chapter 8, tier 2 */
11
12        uint8_t off_be[4];
13        be32_write(off_be, flash_offset);
14
15        hmac_sha256_ctx c;
16        hmac_sha256_init(&c, store_key, 32u);
17        hmac_sha256_update(&c, off_be, 4u); /* position binding first */
18        hmac_sha256_update(&c, hdr, REC_HDR_LEN);
19        hmac_sha256_final(&c, out);
20
21        secure_zero(store_key, sizeof store_key);
22    }

```

Listing 10.1: Record MAC. The offset goes in first, so that a truncated MAC computation cannot silently omit it.

10.4 Reading: the boot-time scan

At boot, `storage_init` walks the log and reconstructs current state. The walk must be robust against every kind of damage an A-3 attacker can leave behind.

```

1 bool storage_init(void)
2 {
3     state_reset_to_safe_defaults(&g_state); /* fail-closed, R-05 */
4     uint32_t best_seq = 0u;
5     bool found = false;
6
7     for (uint32_t off = LOG_BASE; off + REC_SIZE <= LOG_END; off += REC_SIZE) {
8         record_t r;
9         flash_read(off, (uint8_t *)&r, REC_SIZE);
10
11         if (be16_read(r.magic) != REC_MAGIC) {
12             continue; /* erased, or a torn write. Keep scanning. */
13         }
14
15         uint8_t expect[32];
16         record_mac(off, (const uint8_t *)&r, expect);
17         if (!ct_eq(expect, r.mac, 32u)) {
18             continue; /* forged, corrupted, or relocated. Ignore. */
19         }
20
21         uint32_t seq = be32_read(r.seq);
22         if (!found || seq > best_seq) {
23             best_seq = seq;
24             state_apply_record(&g_state, &r);
25             found = true;
26         }
27     }
28
29     g_next_seq = found ? (best_seq + 1u) : 1u;
30     g_write_off = find_first_erased_slot();
31     return true; /* an empty log is a valid state: a fresh device */
32 }

```

Listing 10.2: The scan. Note that it never stops early on encountering a bad record.

Three properties of that loop are deliberate and worth naming.

It does not stop at the first bad record. A torn write in the middle of the log must not hide the valid records after it. Stopping early is how a single well-timed power cut becomes a

permanent rollback.

It selects by highest seq, not by last position. Combined with the offset binding in the MAC, this makes rollback require forging a MAC, which requires the storage key, which requires the root.

It starts from safe defaults. If the log is empty, damaged, or entirely forged, the device comes up in the most restrictive state it has, not the most permissive. Fail-closed is not only a permission-model idea; it applies to state reconstruction too.

10.5 Writing: the append

```

1 bool storage_append(uint8_t type, const uint8_t payload[24])
2 {
3     if (g_write_off + REC_SIZE > LOG_END) {
4         if (!log_compact()) { return false; } /* see next section */
5     }
6
7     record_t r;
8     memset(&r, 0, sizeof r);
9     be16_write(r.magic, REC_MAGIC);
10    r.version = REC_VERSION;
11    r.type = type;
12    be32_write(r.seq, g_next_seq);
13    memcpy(r.payload, payload, 24u);
14    record_mac(g_write_off, (const uint8_t *)&r, r.mac);
15
16    /* Write the whole 64 bytes in one flash program operation where the
17     * part allows it. If the part programs in smaller units, write the
18     * MAC LAST, so that a torn write leaves a record whose MAC is still
19     * erased 0xFF and therefore cannot verify. */
20    if (!flash_program(g_write_off, (const uint8_t *)&r, REC_SIZE)) {
21        return false;
22    }
23
24    /* Read back and verify. Flash program failures are not always
25     * reported. A record we think we wrote but did not is a silent
26     * rollback waiting to happen. */
27    record_t check;
28    flash_read(g_write_off, (uint8_t *)&check, REC_SIZE);
29    if (memcmp(&check, &r, REC_SIZE) != 0) {
30        return false;
31    }
32
33    g_write_off += REC_SIZE;
34    g_next_seq += 1u;
35    return true;
36 }

```

Listing 10.3: Append. The ordering here is what makes power loss safe.

The point

Write the MAC last. Then any interruption produces a record whose MAC field is still erased, which fails verification, which means the scan ignores it and the previous record remains authoritative. Power-loss atomicity falls out of field ordering rather than requiring a journal.

10.6 Compaction: the one dangerous moment

When the log fills, you must erase and start again. This is the only point in the design where valid state is not present in flash, and it needs care.

The technique is two sectors used alternately. Never one.

1. Sector A is full. Sector B is erased and empty.
2. Write the current state as the first record of sector B, with a `seq` higher than anything in A.
3. Verify it reads back correctly.
4. Only now erase sector A.

At every instant during that sequence, at least one sector holds a valid, authenticated, current record. Power loss between steps 2 and 4 leaves both sectors holding valid records, and the highest-`seq` rule picks the right one. Power loss during step 4 leaves sector A damaged and sector B correct, which is exactly the intended outcome.

Get this wrong and the device is broken

The tempting single-sector version is: read state to RAM, erase the sector, write state back. There is a window of several milliseconds during which the only copy of your failure counter is in volatile RAM. An attacker with A-3 capability who cuts power inside that window resets the device to a fresh state with an empty log. They do not need to be precise; the window is enormous by their standards. **Never let the only copy of security-critical state live in RAM, not even briefly.**

What else I could have done

A dedicated FRAM or EEPROM part with byte-level writes and effectively unlimited endurance eliminates this entire chapter. It is an extra component and an extra bus, and on a fixed platform it was not available. **A wear-levelling filesystem** like LittleFS solves the wear and power-loss problems with far more generality than I need, at a cost of several kilobytes of code and a large amount of behaviour I would have to audit. For a few hundred bytes of state, a hand-written append-only log is about two hundred lines and I can read every one of them, which for security-critical storage is the trade I want.

Counters That Cannot Go Backwards

11.1 Why counters are the whole game

Three of the six attacker goals from Chapter 2 reduce to a counter. G-4 is literally counter rollback. G-5, replay, is defeated by a counter. And the rate limiting that defeats G-2 is a counter. If your counters are sound, a large part of your security is sound; if they can be rolled back, essentially nothing else you built matters.

The point

On an embedded device with no hardware monotonic counter, the anti-rollback design is the security architecture. Everything else is downstream of it.

11.2 The counters this device keeps

Counter	Increments on	Protects against
<code>fail_count</code>	Failed auth attempt	Brute force (G-2). Reset only on success.
<code>generation</code>	Explicit revocation	Reuse of revoked capabilities (G-4). Never resets.
<code>msg_counter</code>	Every accepted command	Replay of a previously accepted command (G-5).
<code>boot_count</code>	Every power-on	Reconnaissance; detecting an attacker power-cycling repeatedly.

Note that `fail_count` resets on success and the others never do. A counter that resets needs extra care, because the reset path is itself an attack target: if an attacker can trigger the reset without a genuine success, the rate limit is gone.

11.3 The increment-before-check rule

This is the most important paragraph in the chapter and possibly in the document, so I will state it as plainly as I can.

The point

When checking a secret, **write the incremented failure counter to persistent storage, and verify the write, before you perform the comparison.** Reset it afterwards only if the comparison succeeded.

The reason is an attacker with A-3 capability. Consider the natural ordering:

```

1 FUNCTION authenticate(candidate_pin)
2   IF check_pin(candidate_pin) THEN
3     fail_count = 0
4     RETURN success
5   ELSE
6     fail_count = fail_count + 1 # attacker cuts power HERE

```

```

7     store(fail_count)
8     RETURN failure
9     END
10    END

```

Listing 11.1: The wrong order. This is what almost everybody writes first.

The attacker submits a wrong PIN and cuts power at the marked point, before the store completes. The device reboots. `fail_count` is unchanged. They try the next PIN. They have unlimited attempts, and the rate limit you carefully designed does nothing at all. This attack requires no precision beyond "cut power somewhere in a several-microsecond window," which is trivial with a MOSFET and a microcontroller.

The correct ordering removes the window entirely:

```

1  FUNCTION authenticate(candidate_pin)
2      IF fail_count >= LOCKOUT_THRESHOLD THEN
3          apply_backoff_delay(fail_count)
4      END
5
6      # Charge for the attempt BEFORE making it, and confirm the charge stuck.
7      fail_count = fail_count + 1
8      IF NOT store_and_verify(fail_count) THEN
9          RETURN failure # cannot account for the attempt, so refuse it
10     END
11
12     IF constant_time_check_pin(candidate_pin) THEN
13         fail_count = 0
14         store_and_verify(fail_count) # refund only on genuine success
15         RETURN success
16     END
17
18     RETURN failure
19 END

```

Listing 11.2: The right order. Power loss at any point leaves the counter incremented, which is the safe direction to fail.

Now trace the attacker's options. Cut power before the store: the attempt never happened, so nothing was gained. Cut power after the store but before the comparison: counter incremented, attempt wasted. Cut power after a failed comparison: counter already incremented. Cut power after a *successful* comparison but before the reset: counter stays incremented, which costs a legitimate user one attempt and costs the attacker nothing they wanted. There is no window where an attempt is free.

Get this wrong and the device is broken

The reset on success is the remaining subtlety. If an attacker can make the reset happen without a genuine success, the whole scheme collapses. That is why the reset lives inside the success branch of a constant-time comparison and why the comparison result is a single value that is checked once. Chapter 17 adds a redundant check here specifically because this branch is the highest-value glitch target on the entire device.

11.4 Replay defence with the message counter

`msg_counter` makes every accepted command unique. The host includes the counter value it believes is current; the device accepts only a value strictly greater than what it has stored, then stores the new value.

```

1 bool session_check_replay(const command_t *cmd)
2 {
3     uint32_t stored = state_get_msg_counter(&g_state);
4
5     /* Strictly greater. Equal means a replay of the immediately
6      * preceding message, which is the easiest replay to mount. */
7     if (cmd->counter <= stored) {
8         return false;
9     }
10
11     /* Bound the forward jump. Without this, an attacker sends
12      * counter = 0xFFFFFFFF once and permanently wedges the device,
13      * which is a protocol-level permanent denial of service (R-14). */
14     if (cmd->counter - stored > MSG_COUNTER_MAX_SKIP) {
15         return false;
16     }
17
18     /* Commit before executing. If we execute first and lose power
19      * before committing, the command is replayable. */
20     return state_set_msg_counter(&g_state, cmd->counter);
21 }

```

Listing 11.3: Replay check. Strictly greater, not greater-or-equal, and stored before the command executes.

The forward-jump bound is the kind of detail that only appears after somebody attacks your device. Without it, the replay defence itself becomes the denial-of-service vector, which is a nicely ironic way to fail R-14.

11.5 Revocation through the generation counter

Revocation is where the counter design and the key hierarchy meet, and the result is more elegant than either alone.

Recall from Chapter 8 that operational keys are derived as

$$K = \text{HKDF-Expand}(\text{PRK}_{\text{dev}}, \text{LABEL} \parallel \text{CONTEXT}, 32)$$

Put the generation counter into CONTEXT. Now every key in the device is implicitly tagged with the generation under which it was derived.

To revoke everything, increment the generation. That is the entire operation. Every key derived under generation n becomes unreachable because the device will only ever derive under generation $n + 1$, and HKDF gives no way to compute one from the other.

```

1 bool revoke_all_capabilities(void)
2 {
3     uint32_t gen = state_get_generation(&g_state);
4
5     if (gen == UINT32_MAX) {
6         return false; /* refuse to wrap; wrapping un-revokes everything */
7     }
8
9     /* Persist first. A generation increment that is lost on power failure
10      * means the revoked keys come back to life. */
11     if (!state_set_generation(&g_state, gen + 1u)) {
12         return false;
13     }
14
15     /* Any cached derived key belongs to the old generation. Drop them. */
16     scrub_secrets();
17     session_reset();
18     return true;

```

19 | }

Listing 11.4: Revocation. Eleven lines, because the hierarchy did the work.

The point

Revocation costs one counter increment and no key transport, no revocation list, no certificate handling, and no communication with an authority. That falls directly out of deriving keys from context rather than storing them. It is the clearest example in this document of a structural decision paying for itself three chapters later.

What else I could have done

A **revocation list** of individually revoked identifiers is finer grained: you can revoke one capability without invalidating the rest. It costs storage that grows without bound, a lookup on every operation, and a decision about what to do when the list fills. **Short-lived keys with expiry** avoid revocation entirely but require a trusted clock, which this device does not have and cannot get. Generation counters give coarse but absolutely reliable revocation for four bytes of state, and coarse-but-reliable beats fine-but-fragile on a device with no clock and no unbounded storage.

PART V

Deciding Who Gets In

PIN Authentication and Rate Limiting

12.1 The unavoidable weakness of a PIN

A PIN is a short secret chosen by a human. A six-digit PIN has one million possibilities, which is twenty bits of entropy on a generous day and considerably less in practice because humans pick birthdays. Against a machine that can test a million candidates per second, twenty bits is nothing.

So the security of PIN authentication does not come from the PIN. It comes entirely from making guesses expensive. That reframing matters because it tells you where to spend your effort:

The point

You are not protecting the PIN. You are rationing attempts. Every design decision in this chapter follows from that: the storage format, the comparison, the counter ordering, and the backoff schedule all exist to make the n th guess cost more than the attacker is willing to pay.

12.2 Never store the PIN

The device stores a verifier, not the PIN. The verifier is:

$$V = \text{HKDF-Expand}(\text{PRK}_{\text{dev}}, \text{"neusense/pin/v1"} \parallel \text{user_id} \parallel \text{salt}, 32) \text{ keyed over the PIN}$$

Concretely, the PIN is stretched and then MACed under a device-bound key, so that a verifier lifted out of flash is useless without the device's root secret.

```

1  #define PIN_STRETCH_ROUNDS 20000u
2
3  /* Derive the verifier for a candidate PIN.
4   *
5   * Device binding: the pin key comes from PRK_dev, so a verifier copied
6   * to another device will never match there. That removes the offline
7   * attack where somebody dumps flash and cracks the PIN on a GPU.
8   *
9   * Stretching: 20000 HMAC iterations costs the legitimate user a few tens
10  * of milliseconds, which is invisible, and costs an attacker who somehow
11  * did get the pin key a factor of 20000 on every guess.
12  */
13  static void pin_verifier(const uint8_t *pin, size_t pin_len,
14                          uint8_t user_id, const uint8_t salt[16],
15                          uint8_t out[32])
16  {
17      uint8_t pin_key[32];
18      derive_pin_key(user_id, pin_key); /* tier 2, Chapter 8 */
19
20      /* Round 0: bind PIN, salt, and user together. */
21      hmac_sha256_ctx c;
22      hmac_sha256_init(&c, pin_key, 32u);
23      hmac_sha256_update(&c, salt, 16u);
24      hmac_sha256_update(&c, &user_id, 1u);
25      hmac_sha256_update(&c, pin, pin_len);
26      hmac_sha256_final(&c, out);
27
28      /* Rounds 1..N: iterate. Each round depends on the previous, so the

```

```

29  * work cannot be parallelised within a single guess. */
30  for (uint32_t i = 1u; i < PIN_STRETCH_ROUNDS; i++) {
31      hmac_sha256_init(&c, pin_key, 32u);
32      hmac_sha256_update(&c, out, 32u);
33      hmac_sha256_final(&c, out);
34  }
35
36  secure_zero(pin_key, sizeof pin_key);
37 }

```

Listing 12.1: PIN verifier derivation. Two properties matter: it is device-bound, and it is slow on purpose.

Get this wrong and the device is broken

The iteration count must not depend on the PIN, and the loop must run the full count on every call including calls that will fail. It is tempting to add an early exit for obviously-wrong input. Do not. The wall-clock time of a failed authentication must be identical to the time of a successful one, or the timing itself becomes the oracle you spent this whole chapter trying to avoid.

12.3 The comparison

`memcmp` returns as soon as it finds a difference. That means comparing a 32-byte verifier takes a time proportional to how many leading bytes matched. An attacker measuring that time recovers the verifier one byte at a time, in 32×256 trials instead of 2^{256} .

The fix is a comparison that always reads every byte and whose result is computed with arithmetic rather than branches.

```

1  /* Returns true if the two buffers are equal, in time depending only on
2  * n and not on the contents.
3  *
4  * How it works: accumulate the bitwise differences of every byte pair.
5  * If any byte differed, some bit in diff is set. Then convert
6  * "diff is zero" into a boolean without a branch.
7  */
8  bool ct_eq(const uint8_t *a, const uint8_t *b, size_t n)
9  {
10     uint8_t diff = 0u;
11
12     for (size_t i = 0u; i < n; i++) {
13         diff |= (uint8_t)(a[i] ^ b[i]); /* no branch, no early exit */
14     }
15
16     /* Fold the 8 bits of diff down to bit 0, then invert.
17     * If diff == 0 the result is 1; otherwise 0. No comparison,
18     * so no branch for the compiler to turn into a jump. */
19     return (bool)((uint8_t)((((uint32_t)diff - 1u) >> 8) & 1u));
20 }

```

Listing 12.2: Constant-time equality. This function appears in every serious cryptographic codebase and it is worth understanding completely.

There is a second-order concern here that is worth naming because it catches experienced people: **the compiler is allowed to undo your work**. An optimiser that recognises the pattern may emit a branch anyway. The defences are to keep this function in a separate translation unit compiled without aggressive optimisation, to check the disassembly, and, in the Rust version in Chapter 20, to use a volatile read barrier that the optimiser is not permitted to see through. Chapter 18 covers this properly.

12.4 The backoff schedule

Once you are counting failures reliably (Chapter 11) and comparing safely, the remaining question is what to do with the count. My schedule:

Consecutive fails	Delay before next attempt	Reasoning
1–3	none	Typos happen. Punishing a legitimate user for three mistakes is bad design, and three free guesses out of a million is not a meaningful attacker gain.
4–6	1 second	Enough to make automated guessing tedious, short enough that a flustered human does not notice much.
7–10	10 seconds	Now clearly adversarial.
11–20	60 seconds	About 500 attempts per day maximum.
21+	300 seconds	Roughly 288 attempts per day. Exhausting a six-digit space takes about 9 years.

12.4.1 Why the delay grows but never becomes permanent

A permanent lockout after n failures satisfies **R-07** elegantly and violates **R-14** completely. Anybody who can reach the device can brick it by submitting bad PINs, which converts an authentication control into a denial-of-service weapon pointed at the device's owner. Growing backoff gives you the same practical protection, because 288 guesses a day against a million-element space is not an attack anybody runs, and it leaves the legitimate user a way back.

Get this wrong and the device is broken

The delay must be enforced by the device, using the device's own timer, and it must survive reset. A delay implemented as "the host waits before sending the next command" is not a delay, it is a suggestion. And a delay whose remaining time lives only in RAM is defeated by a power cycle, which is why the failure count, not a countdown, is the persistent quantity: the device recomputes the required delay from the count at every boot.

```

1 static uint32_t backoff_ms(uint32_t fails)
2 {
3     if (fails <= 3u) { return 0u; }
4     if (fails <= 6u) { return 1000u; }
5     if (fails <= 10u) { return 10000u; }
6     if (fails <= 20u) { return 60000u; }
7     return 300000u;
8 }
9
10 /* Busy-wait on the device's own timer. Interrupts stay enabled so the
11  * link layer does not lose bytes, but no command is processed. */
12 static void apply_backoff(uint32_t fails)
13 {
14     uint32_t ms = backoff_ms(fails);
15     uint32_t start = timer_millis();
16     while ((timer_millis() - start) < ms) {
17         /* Deliberately does nothing. A sleep mode would be nicer for
18          * power, but wake sources are attacker-influenced and I would

```

```

19     * rather burn current than add an attacker-controlled wake path. */
20     }
21 }

```

Listing 12.3: Backoff. Derived from a persistent count, applied before the attempt is processed.

12.5 The complete authenticate path

Assembling Chapters 11 and 12 into the actual function:

```

1 bool cmd_authenticate(uint8_t user_id, const uint8_t *pin, size_t pin_len)
2 {
3     /* Structural validation first. Length is not secret. */
4     if (pin_len < PIN_MIN_LEN || pin_len > PIN_MAX_LEN) {
5         return false;
6     }
7
8     uint32_t fails = state_get_fail_count(&g_state);
9     apply_backoff(fails); /* R-07 */
10
11     /* Charge for the attempt before making it. R-06. */
12     if (!state_set_fail_count(&g_state, fails + 1u)) {
13         return false; /* cannot account for it, so refuse it */
14     }
15
16     /* Load the stored verifier and salt for this user. */
17     uint8_t stored[32], salt[16];
18     if (!state_get_pin_record(&g_state, user_id, stored, salt)) {
19         /* Unknown user. Still do the full work, so that the timing of an
20          * unknown user is indistinguishable from a wrong PIN. Otherwise
21          * we have handed the attacker a user enumeration oracle. */
22         rng_bytes(stored, 32u);
23         rng_bytes(salt, 16u);
24     }
25
26     uint8_t candidate[32];
27     pin_verifier(pin, pin_len, user_id, salt, candidate);
28
29     bool ok = ct_eq(candidate, stored, 32u);
30
31     secure_zero(candidate, sizeof candidate);
32     secure_zero(stored, sizeof stored);
33
34     if (ok) {
35         (void)state_set_fail_count(&g_state, 0u); /* reset on success */
36         session_authenticate(user_id);
37         return true;
38     }
39     return false;
40 }

```

Listing 12.4: Authenticate. Read this against the pseudocode in Chapter 11; the ordering is identical and it is the ordering that matters.

The unknown-user branch deserves attention. If an unknown user short-circuits, the device answers "no such user" faster than it answers "wrong PIN," and an attacker learns which user identifiers exist. Doing the full verification against random bytes costs nothing and closes the enumeration channel. This pattern appears in every well-built authentication system and it is left out of most homegrown ones.

What else I could have done

A **PAKE**, such as OPAQUE or SRP, would let the host prove knowledge of the PIN without ever sending it, which protects the PIN from a compromised host and from anybody watching the link. That is a real improvement and it is what I would use if the host were untrusted in a deployment sense rather than merely in a threat-model sense. It requires public key operations, which Chapter 6 explains I do not have. A **hardware-backed lockout**, where the secure element enforces the counter, is strictly better than my software counter and requires a secure element. **Longer PINs or passphrases** would help more than any of this, and are a user-experience decision rather than an engineering one, but if you get to choose, choose more entropy.

The Fail-Closed Permission Model

13.1 What fail-closed means, precisely

Fail-closed means the answer to "may this operation proceed?" is *no* unless a specific, positively-matched rule says yes. Not "no unless something went wrong." Not "yes unless denied." No, by default, in every code path, including paths that were never anticipated.

The distinction is easiest to see in code. Here are two functions that behave identically on well-formed input:

```

1 bool permission_allows_BAD(const session_t *s, uint8_t op)
2 {
3     for (size_t i = 0; i < s->rule_count; i++) {
4         if (s->rules[i].op == op && s->rules[i].effect == DENY) {
5             return false;
6         }
7     }
8     return true; /* <-- anything not explicitly denied is allowed */
9 }

```

Listing 13.1: Fail-open. Looks correct. Is not.

```

1 bool permission_allows(const session_t *s, uint8_t op)
2 {
3     if (!s->authenticated) { return false; }
4     if (s->rule_count > MAX_RULES) { return false; } /* corrupt state */
5     if (s->generation != state_get_generation(&g_state)) { return false; }
6
7     for (size_t i = 0; i < s->rule_count; i++) {
8         if (s->rules[i].op == op && s->rules[i].effect == ALLOW) {
9             return true;
10        }
11    }
12    return false; /* <-- anything not explicitly allowed is denied */
13 }

```

Listing 13.2: Fail-closed. The default is the safe answer.

Now consider what happens when the rule table is corrupted by a glitch, or when `rule_count` is zero because a record failed to load, or when a new opcode is added to the firmware and nobody updated the policy. The first function grants. The second denies. Those are the same inputs producing opposite security outcomes, and the difference is one keyword.

The point

Write your permission function so that *every* unexpected condition falls through to deny. Corruption, uninitialised state, unknown opcodes, empty tables, and paths you have not thought of yet should all take you to the same place, and that place should be a refusal.

13.2 The model: capabilities, not roles

There are two common ways to express permissions.

Role-based assigns each user a role, and each role a set of allowed operations. It is compact and it is how most enterprise systems work.

Capability-based gives the session a list of specific things it may do, with no indirection.

I chose capabilities, for one reason that matters on a small device: **there is no lookup**. A role-based check has to resolve role to permission set, which means a table, which means an index, which means an index that could be out of range, which means another failure mode. A capability list is checked directly against what is in front of you. Less indirection, fewer paths, smaller attack surface.

Capability	Bit	Permits
CAP_ENCRYPT	0	Encrypt data under a derived key
CAP_DECRYPT	1	Decrypt data under a derived key
CAP_MAC	2	Compute an authentication tag
CAP_READ_STATE	3	Read non-secret device state
CAP_REVOKE	4	Increment the generation counter
CAP_SET_PIN	5	Change a PIN verifier

Six capabilities in a bitmask fits in one byte with room to spare, and the check is a single mask-and-test.

13.3 Redundancy against fault injection

A single `if` statement is a single branch instruction, and a single branch instruction is exactly one glitch away from going the wrong way. Chapter 17 covers glitching in depth; here is the countermeasure as it applies to the permission decision.

```

1  /* Deliberately not booleans. A glitched bool is 50/50 to land on true.
2   * These patterns have high Hamming distance, so a single flipped bit
3   * produces neither value and the check fails closed. */
4  #define PERM_GRANTED 0xA5C33C5Au
5  #define PERM_DENIED 0x5A3CC3A5u
6
7  uint32_t permission_check_hardened(const session_t *s, uint8_t op)
8  {
9      uint32_t v1 = PERM_DENIED;
10     uint32_t v2 = PERM_DENIED;
11
12     /* First evaluation. */
13     if (s->authenticated && ((s->caps >> op) & 1u) != 0u) {
14         v1 = PERM_GRANTED;
15     }
16
17     /* Second, independent evaluation, structured differently so that a
18      * glitch tuned to the first is unlikely to affect the second. */
19     uint32_t mask = 1u << op;
20     if (((s->caps & mask) == mask) && (s->authenticated == true)) {
21         v2 = PERM_GRANTED;
22     }
23
24     /* Both must agree, and both must equal the granted pattern.
25      * Any disagreement, any corruption, any glitch: denied. */
26     if (v1 != v2) { return PERM_DENIED; }
27     if (v1 != PERM_GRANTED) { return PERM_DENIED; }
28
29     return PERM_GRANTED;
30 }
31
32 /* Call sites compare against the pattern, never against "not denied". */
33 void execute_command(const command_t *cmd)
34 {

```

```

35     if (permission_check_hardenened(&g_session, cmd->opcode) != PERM_GRANTED) {
36         respond_generic_error();
37         return;
38     }
39     dispatch(cmd);
40 }

```

Listing 13.3: Redundant permission check. Three independent mechanisms must agree before the operation proceeds.

Two details carry the weight. The sentinel values are chosen so that flipping any single bit produces a value equal to neither, which means a single-bit fault denies rather than grants. And the call site tests for equality with `PERM_GRANTED` rather than inequality with `PERM_DENIED`, so that a glitched return value of any third pattern is refused.

13.4 Control-flow integrity, cheaply

A related attack skips the permission check entirely by glitching the branch that calls it. A cheap partial defence is a counter that every security-relevant stage increments, checked before the operation commits.

```

1  static volatile uint32_t g_flow;
2
3  #define FLOW_RESET() do { g_flow = 0u; } while (0)
4  #define FLOW_STAGE(n) do { g_flow += (n); } while (0)
5  #define FLOW_EXPECT(n) ((g_flow) == (n))
6
7  void handle_one_command(void)
8  {
9      FLOW_RESET();
10
11     if (!parse_command(...)) { goto fail; } FLOW_STAGE(1u);
12     if (!session_check_replay(...)) { goto fail; } FLOW_STAGE(2u);
13     if (permission_check_hardenened(...) != PERM_GRANTED) { goto fail; }
14     FLOW_STAGE(4u);
15
16     /* 1 + 2 + 4. If any stage was skipped by a glitch, the sum is wrong
17      * and we never reach dispatch. */
18     if (!FLOW_EXPECT(7u)) { goto fail; }
19
20     dispatch(...);
21     return;
22
23 fail:
24     respond_generic_error();
25 }

```

Listing 13.4: Control-flow counter. Not a substitute for hardware CFI, but it catches skipped stages.

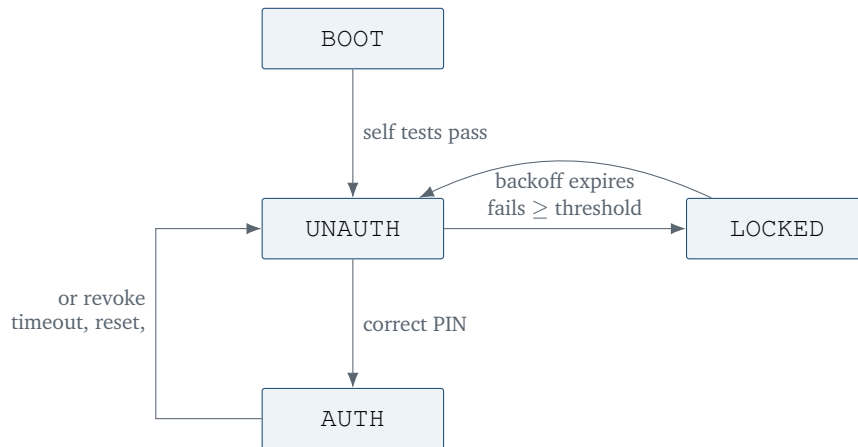
What else I could have done

Real control-flow integrity needs hardware support: a shadow stack, landing-pad instructions, or a memory protection unit enforcing execute-only regions. None of that exists here. The counter is a poor substitute that raises the cost of a skip attack from one well-placed glitch to two correlated ones. I would not claim it as a defence against a determined attacker with a good glitcher, and I do not want anybody reading this to think it is more than it is.

Session Lifecycle

14.1 Four states, and no others

The session state machine is small on purpose. Every state a security state machine can be in is a state somebody has to reason about, and states multiply faster than reasoning does.



State	What is true in it
BOOT	Self tests running. No link, no commands accepted, no keys derived.
UNAUTH	Link up. Only unprivileged commands accepted. No capability bits set. No keys in SRAM.
AUTH	A user has proven their PIN. Capability bits set from their record. Keys derived on demand and scrubbed after each use.
LOCKED	Too many failures. Backoff in progress. Behaves exactly like UNAUTH except that authenticate is delayed.

14.2 The rules that hold in every state

1. **The device powers on in UNAUTH.** There is no persistent authentication. A device that remembers you were authenticated last time is a device that an attacker authenticates by power cycling.
2. **Any reset returns to UNAUTH.** Including watchdog resets, hard fault resets, and brown-out resets. There is exactly one code path that sets AUTH and it runs only after a successful comparison.
3. **The session records the generation it was created under.** If the generation changes mid-session, every subsequent permission check fails, because the check compares them. Revocation takes effect immediately, not at next login.
4. **A session times out.** Not because of any threat in the model, but because a device left plugged in and authenticated overnight is a real-world failure that no protocol design prevents.

```

1  typedef struct {
2      bool authenticated;
3      uint8_t user_id;
4      uint8_t caps; /* capability bitmask, Chapter 13 */
5      uint32_t generation; /* generation this session was born under */
6      uint32_t last_activity_ms;
7      uint8_t session_nonce[16];
8  } session_t;
9
10 static session_t g_session;
11
12 void session_reset(void)
13 {
14     /* Whole-struct zeroing rather than field-by-field, so a field added
15      * later is cleared automatically rather than silently persisting. */
16     secure_zero(&g_session, sizeof g_session);
17
18     g_session.authenticated = false; /* explicit, for the reader */
19     scrub_secrets();
20     aes256_clear();
21 }
22
23 void session_tick(void)
24 {
25     if (g_session.authenticated) {
26         uint32_t idle = timer_millis() - g_session.last_activity_ms;
27         if (idle > SESSION_TIMEOUT_MS) {
28             session_reset();
29         }
30         if (g_session.generation != state_get_generation(&g_state)) {
31             session_reset(); /* revoked mid-session */
32         }
33     }
34 }

```

Listing 14.1: Session structure and reset. Reset is written so that adding a new field without adding it here causes a compile error.

The point

`session_reset` zeroes the whole struct rather than setting fields individually. This is a small habit with a large payoff: six months from now somebody adds a `bool is_admin` field, forgets to clear it, and a stale value survives a reset. Whole-struct clearing makes that class of bug impossible rather than merely unlikely.

PART VI

Talking to the Host

The Command Protocol

15.1 Design goals, in priority order

The protocol is the only thing an A-0 or A-1 attacker can touch, which makes its design the highest-leverage security work in the project. My priorities, ordered, and the order is the interesting part:

1. **Unambiguous.** Every byte sequence has exactly one interpretation, or is rejected. Ambiguity is where parser differentials live.
2. **Bounded.** Every field has a maximum size known at compile time. No length is unbounded, ever.
3. **Authenticated.** Every command carries a MAC binding it to this device and this message counter.
4. **Simple enough to parse without a state machine.** If parsing needs a state machine, parsing needs testing that I cannot exhaustively do.
5. **Efficient.** Last, and a distant last. This device processes a handful of commands per second at most.

The point

Efficiency is at the bottom of that list on purpose. Almost every protocol design mistake I have seen in embedded work traces back to somebody optimising bytes on the wire and introducing a variable-length encoding, an implicit length, or an optional field to save four bytes. Those four bytes cost you a parser you cannot verify.

15.2 Frame format

Field	Bytes	Notes
sync	2	0x4E 0x53. Lets the receiver resynchronise after garbage.
version	1	Protocol version. A mismatch is rejected outright, never negotiated.
opcode	1	The command. Unknown values are rejected.
counter	4	Message counter, big-endian. Chapter 11.
length	2	Payload length, big-endian, validated against <code>HSM_MAX_PAYLOAD_LEN</code> before use.
payload	0–1024	Opaque to the framing layer.
mac	32	HMAC-SHA-256 over every preceding byte, under the session key.

Fixed header, one variable field, trailing MAC. That is the whole thing.

15.2.1 Why big-endian

Because it removes a class of bug. On a little-endian core, a big-endian wire format forces you to write explicit conversion functions, and explicit conversion functions are code you can read and review. Native-endian wire formats invite `memcpy` into a struct, which works perfectly until somebody builds for a different target, and which hides alignment assumptions that will fault on a Cortex-M0+ where unaligned access is not permitted.

```

1 static inline uint16_t be16_read(const uint8_t *p)
2 {
3     return (uint16_t)((uint16_t)p[0] << 8) | (uint16_t)p[1];
4 }
5
6 static inline uint32_t be32_read(const uint8_t *p)
7 {
8     return ((uint32_t)p[0] << 24) | ((uint32_t)p[1] << 16)
9         | ((uint32_t)p[2] << 8) | (uint32_t)p[3];
10 }
11
12 static inline void be32_write(uint8_t *p, uint32_t v)
13 {
14     p[0] = (uint8_t)(v >> 24); p[1] = (uint8_t)(v >> 16);
15     p[2] = (uint8_t)(v >> 8); p[3] = (uint8_t)(v);
16 }

```

Listing 15.1: Byte-order helpers. Boring, explicit, and impossible to get subtly wrong.

Note that these read byte at a time. No casting a `uint8_t*` to a `uint32_t*`, which would be both an alignment fault and a strict-aliasing violation. This is the kind of thing that works on your development board and faults in the field.

15.3 One response, always

R-11 demands that every error look the same. The response format is therefore deliberately uninformative:

Field	Bytes	Notes
sync	2	0x4E 0x53
status	1	0x00 success, 0x01 failure. Two values. That is the entire vocabulary.
length	2	Zero on failure, always.
payload	0–1024	Present only on success.
mac	32	Over everything preceding.

Get this wrong and the device is broken

Resist the urge to add error codes. "Bad MAC," "unknown opcode," "not authenticated," "insufficient permission," and "replay detected" are five extremely useful pieces of information, and you would be giving all five of them to the attacker for free. During development, build a debug variant that emits detailed errors over a separate channel, and make it structurally impossible for that variant to be built for release. I gate it behind a compile-time flag that the release build's linker script refuses to resolve, so a mistake is a link error rather than a shipped device.

Uniform responses are only half of **R-11**. The other half is uniform *timing*, which is harder: a rejection at MAC verification naturally takes longer than a rejection at length validation. My approach is a floor. Every command path, success or failure, is padded to a fixed minimum duration measured from frame receipt.

```

1  #define RESPONSE_FLOOR_MS 25u
2
3  void respond(bool ok, const uint8_t *payload, uint16_t len)
4  {
5      /* g_cmd_start_ms was captured the instant the frame completed. */
6      uint32_t elapsed = timer_millis() - g_cmd_start_ms;
7      while (elapsed < RESPONSE_FLOOR_MS) {
8          elapsed = timer_millis() - g_cmd_start_ms;
9      }
10
11     if (!ok) { payload = NULL; len = 0u; }
12     frame_send(ok ? 0x00u : 0x01u, payload, len);
13 }

```

Listing 15.2: Response timing floor. Crude, effective, and cheap on a device that does not need throughput.

The floor must exceed the slowest legitimate path, including the PIN stretching from Chapter 12. Measure it, then add margin, then measure it again on a cold board, because flash wait states and clock trim vary.

15.4 Session binding and replay

The command MAC uses the session key, derived per session from a nonce the device generates:

$$K_{\text{sess}} = \text{HKDF-Expand}(\text{PRK}_{\text{dev}}, \text{"neusense/sess/v1"} \parallel \text{nonce} \parallel \text{generation}, 32)$$

This binds each command to three things at once: the device, because PRK_{dev} is device specific; the session, because the nonce is fresh; and the generation, because revocation changes the derivation. A command captured from one session cannot be replayed into another, a command from one device cannot be replayed into another, and a command from before a revocation cannot be replayed after it. All three properties come from putting the right things in `info`, which is a nice demonstration of why Chapter 8’s structure was worth building.

The Parser Is the Most Dangerous Code You Will Write

16.1 Why this chapter exists

Every control in this document assumes the attacker cannot execute code inside the device. That is assumption five of the threat model, and I flagged it there as load-bearing.

The parser is where that assumption is won or lost. It is the first code to touch attacker-controlled bytes, it runs before authentication, and its input is entirely adversarial. If there is a memory-safety bug in it, the attacker gets code execution, and once they have code execution your key hierarchy, your constant-time comparisons, and your fail-closed permission model are all just data they can read.

The point

Rank your code by risk and the parser is at the top, alone, by a wide margin. It deserves more review time than the cryptography, because the cryptography is standard algorithms with published test vectors, and the parser is code you wrote yourself with no reference implementation to check against.

16.2 Total parsing

The discipline I follow is that the parser is a *total function*: for every possible input, of every possible length, containing every possible byte, it terminates and returns either a valid parsed structure or a failure. There is no input that causes it to read out of bounds, write out of bounds, loop forever, or leave a partially-filled output that a caller might use.

That is a strong claim and it is achievable only if the parser is written a particular way.

16.2.1 Rule one: validate length before touching anything

```

1 bool parse_command(const frame_t *f, command_t *out)
2 {
3     /* Clear the output first. A caller that ignores our return value
4      * gets a safe zeroed structure rather than uninitialised stack. */
5     memset(out, 0, sizeof *out);
6
7     /* 1. Is there even enough data for a header? Check before reading
8      * any header field. */
9     if (f->len < HDR_LEN + MAC_LEN) {
10         return false;
11     }
12
13     /* 2. Sync and version. Cheap rejects first. */
14     if (be16_read(&f->data[0]) != FRAME_SYNC) { return false; }
15     if (f->data[2] != PROTO_VERSION) { return false; }
16
17     /* 3. Read the declared payload length, then validate it against
18      * BOTH the protocol maximum AND the actual bytes we received.
19      * Missing either check is the classic overflow. */
20     uint16_t plen = be16_read(&f->data[8]);
21     if (plen > HSM_MAX_PAYLOAD_LEN) { return false; }
22     if ((size_t)HDR_LEN + plen + MAC_LEN != f->len) { return false; }
23
24     /* 4. Opcode must be one we know. Unknown opcodes are rejected here,
25      * not later by a default case in a switch. */

```

```

26     uint8_t op = f->data[3];
27     if (!opcode_is_known(op)) { return false; }
28
29     /* 5. Verify the MAC before copying anything out. Nothing downstream
30      * ever sees bytes that failed authentication. */
31     const uint8_t *mac = &f->data[HDR_LEN + plen];
32     if (!verify_command_mac(f->data, HDR_LEN + plen, mac)) {
33         return false;
34     }
35
36     /* 6. Only now populate the output. Every value copied here has been
37      * range-checked above. */
38     out->opcode = op;
39     out->counter = be32_read(&f->data[4]);
40     out->len = plen;
41     memcpy(out->payload, &f->data[HDR_LEN], plen); /* plen <= sizeof */
42     return true;
43 }

```

Listing 16.1: The parse entry point. Every check happens before the corresponding field is read.

Step three is the one that kills devices. Checking `plen` against the protocol maximum but not against the received byte count means a frame that *claims* 1024 bytes but *contains* 20 causes a read 1004 bytes past the end of the receive buffer. Checking against the received count but not the maximum means a frame claiming 60000 bytes overflows the destination. You need both, and they are different checks, and it is entirely normal to write one and believe you have written both.

16.2.2 Rule two: no pointer arithmetic on attacker-derived values

Every index in that function is either a compile-time constant or a value that has already been range-checked. There is no `ptr += something_from_the_wire`. Where an offset is needed it is computed and checked as an integer first, and the check uses addition on values that cannot overflow because they have already been bounded.

Get this wrong and the device is broken

Watch for integer overflow in the checks themselves. `if (offset + len > buf_size)` is wrong if `offset + len` can wrap. On a 16-bit length and a 32-bit `size_t` it cannot, but the habit of writing the check as `if (len > buf_size - offset)`, with `offset <= buf_size` already established, is worth forming, because the day you move to 32-bit lengths the first form silently becomes exploitable.

16.2.3 Rule three: no recursion, no unbounded loops

Every loop in the parser has a bound that is a compile-time constant or a value checked against one. There is no recursion anywhere in the firmware. This means the worst-case stack depth is computable statically, which Chapter 19 shows how to do, which in turn means the guard band from Chapter 5 is a backstop rather than a primary defence.

16.2.4 Rule four: fixed-size output, always

`command_t` contains a fixed `uint8_t payload[HSM_MAX_PAYLOAD_LEN]`. It does not contain a pointer into the receive buffer. This costs a kilobyte of SRAM and buys the guarantee that nothing downstream holds a reference into a buffer that will be overwritten by the next frame. Use-after-free without a heap is still possible if you pass around pointers into reused buffers, and it is a bug that is very hard to see in review.

16.3 Fuzzing is not optional

Careful writing gets you most of the way. Fuzzing gets you the rest, and it finds things review does not, reliably, every time.

The parser is written so that it can be compiled and run on a host machine with no hardware dependencies. That is a deliberate architectural choice: `parse_command` takes a buffer and fills a struct, and touches no peripheral. This makes it fuzzable with standard tooling.

```

1  /* Build:
2   * clang -g -O1 -fsanitize=fuzzer,address,undefined \
3   * -DHSM_HOST_BUILD parser.c fuzz_parser.c -o fuzz_parser
4   * Run:
5   * ./fuzz_parser corpus/ -max_len=2048 -runs=50000000
6   */
7  #include <stdint.h>
8  #include <string.h>
9  #include "hsm_parser.h"
10
11 int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
12 {
13     if (size > HSM_MAX_FRAME_LEN) { return 0; }
14
15     frame_t f;
16     memset(&f, 0, sizeof f);
17     memcpy(f.data, data, size);
18     f.len = size;
19
20     command_t cmd;
21     if (parse_command(&f, &cmd)) {
22         /* If we accepted it, the invariants the rest of the firmware
23          * relies on must hold. Assert them here so the fuzzer treats a
24          * violation as a crash. */
25         assert(cmd.len <= HSM_MAX_PAYLOAD_LEN);
26         assert(opcode_is_known(cmd.opcode));
27     }
28     return 0;
29 }

```

Listing 16.2: libFuzzer harness. Twenty lines that will find bugs no amount of reading finds.

Run it with AddressSanitizer and UndefinedBehaviorSanitizer both enabled. ASan catches the out-of-bounds accesses; UBSan catches the integer overflows that lead to them. Seed the corpus with valid frames so the fuzzer gets past the sync check quickly, and let it run for hours, not minutes.

The point

Architect the parser so it can run on a host with no hardware. That one decision is what makes fuzzing, unit testing, and coverage measurement possible at all. A parser welded to a UART peripheral is a parser you can only test by plugging in a board, which means you will test it far less than it needs.

16.4 The receive path

Framing sits below parsing and has its own hazards, mainly around what happens when the attacker sends fewer bytes than they promised, or more, or none, or a continuous stream forever.

```

1 bool link_receive_frame(frame_t *out)
2 {

```

```

3  memset(out, 0, sizeof *out);
4  uint32_t start = timer_millis();
5
6  /* 1. Hunt for sync. Bounded: after this many bytes of garbage we
7   * give up rather than spinning forever on a hostile stream. */
8  size_t discarded = 0u;
9  uint8_t b0 = 0u, b1 = 0u;
10 for (;;) {
11     if (timer_millis() - start > FRAME_TIMEOUT_MS) { return false; }
12     if (discarded > MAX_RESYNC_BYTES) { return false; }
13     if (!uart_read_byte_timeout(&b1, FRAME_TIMEOUT_MS)) { return false; }
14     if (b0 == 0x4Eu && b1 == 0x53u) { break; }
15     b0 = b1;
16     discarded++;
17 }
18 out->data[0] = 0x4Eu; out->data[1] = 0x53u;
19
20 /* 2. Read the fixed header. Fixed count, so no attacker influence. */
21 for (size_t i = 2u; i < HDR_LEN; i++) {
22     if (timer_millis() - start > FRAME_TIMEOUT_MS) { return false; }
23     if (!uart_read_byte_timeout(&out->data[i], FRAME_TIMEOUT_MS)) {
24         return false;
25     }
26 }
27
28 /* 3. Read the declared payload length and bound it BEFORE using it
29  * as a loop count. This is the same check as in the parser, done
30  * again, because defence in depth costs three lines here. */
31 uint16_t plen = bel6_read(&out->data[8]);
32 if (plen > HSM_MAX_PAYLOAD_LEN) { return false; }
33
34 size_t total = HDR_LEN + (size_t)plen + MAC_LEN;
35 if (total > HSM_MAX_FRAME_LEN) { return false; } /* belt and braces */
36
37 for (size_t i = HDR_LEN; i < total; i++) {
38     if (timer_millis() - start > FRAME_TIMEOUT_MS) { return false; }
39     if (!uart_read_byte_timeout(&out->data[i], FRAME_TIMEOUT_MS)) {
40         return false;
41     }
42 }
43
44 out->len = total;
45 return true;
46 }

```

Listing 16.3: Frame receive. Every loop is bounded in both bytes and time.

The overall timeout, checked inside every loop, is what stops an attacker from holding the device hostage by sending one byte per second forever. Without it, a single malicious connection denies service indefinitely at essentially no cost, which fails [R-14](#).

What else I could have done

COBS or SLIP framing with an escaped delimiter removes the resync hunt entirely, because the delimiter byte cannot appear inside a frame. It is a genuinely better framing scheme and I would use it in a design without a fixed-length header. **A length-prefixed stream with no sync bytes** is simpler still but has no recovery path if the two ends ever disagree about where a frame starts, and on a serial link with an attacker injecting bytes, they will disagree. **TLV encoding** for the payload would give flexibility for future commands, and it is exactly the flexibility I did not want: every TLV parser I have reviewed had a length-handling bug in it somewhere.

PART VII

Surviving Physical Access

Side Channels and Fault Injection

17.1 The shift in mindset

Everything up to this point treated the device as a black box that receives bytes and sends bytes. Chapters 17 and 18 drop that assumption. The attacker now has the board on their bench, with probes on it, and they are not sending it messages so much as watching it think.

This is the part of embedded security that surprises people coming from web or application security, because the attack does not involve a bug. Your code can be entirely correct, your cryptography can be textbook, and you still lose, because correct code running on real silicon emits information about what it is computing.

The point

A side channel is not a bug in your program. It is a property of your program's execution on physical hardware. You cannot fix it by fixing a mistake; you fix it by changing what the execution looks like from outside.

17.2 The channels, and what leaks through each

Channel	What the attacker measures	What leaks
Timing	Wall-clock duration of an operation	Branch decisions on secret data; early-exit comparisons; data-dependent loop counts
Power	Current draw over time, sampled fast	Hamming weight of values on the bus; which instructions ran; number of rounds
Electromagnetic	Near-field emissions	Same as power, without needing to break the power rail; often better spatially
Fault	Nothing, they inject	Not a leak; a way to make the device compute the wrong thing on purpose

17.3 Timing: the one you must fix

Timing is the channel to take most seriously, because it is the cheapest to exploit. It requires no laboratory equipment, it can sometimes be measured across the serial link, and on a core with no cache and no branch predictor the relationship between code and duration is close to exact, which is bad news for you and excellent news for the attacker.

The sources of secret-dependent timing, in rough order of how often I have seen them:

1. **Early-exit comparison.** `memcmp` on a MAC or a verifier. Chapter 12 covers the fix.
2. **Branching on a secret.** `if (key[i] == 0) { ... }`. Two paths, two durations.
3. **Table lookups indexed by a secret.** Harmless on a cacheless core, dangerous the moment you port to one with a cache. [R-16](#) forbids it anyway so that the code stays portable.
4. **Loop counts derived from secrets.** Counting set bits by shifting until zero.
5. **Division or modulo on secrets.** Chapter 4 covers why this is worse on this core than on most.

6. **Early rejection in a validation chain.** Rejecting a bad length faster than a bad MAC. Chapter 15's response floor handles this.

17.4 Power analysis, honestly assessed

Simple power analysis reads the trace directly and identifies operations by their shape. On a device doing software AES, the sixteen S-box lookups of a round are often visible to the naked eye on an oscilloscope.

Differential power analysis is far more powerful. The attacker collects thousands of traces with varying inputs, guesses one byte of the key, predicts the power consumption at a chosen intermediate value under that guess, and correlates the prediction against the measurements. The correct guess correlates; the other 255 do not. Then they move to the next byte. The full key falls in 16×256 guesses rather than 2^{128} , and the whole thing is a weekend with an off-the-shelf toolkit.

17.4.1 What I do about it

Use the hardware AES engine. Chapter 4 made this argument and I will not repeat it beyond the summary: the engine computes in dedicated parallel logic over few cycles, exposing far less key-correlated structure than a software implementation running through the CPU's register file and data bus.

Use DMA for the data path. Plaintext and ciphertext never pass through the core, so the bus activity around the core carries less of your data.

Limit operations per key. DPA needs many traces under the same key. Deriving a fresh key per record, per context, from the hierarchy in Chapter 8, means the attacker cannot accumulate thousands of traces under one key by asking the device to do the same thing repeatedly. This is a genuine benefit of the key hierarchy that I did not design it for, and it is worth stating because it is the kind of thing that makes good architecture pay off twice.

The point

Differential power analysis needs volume. Anything that reduces the number of operations an attacker can observe under a single key raises the cost proportionally. Per-context key derivation is therefore a side-channel countermeasure as much as a compartmentalisation measure.

17.4.2 What I do not do, and what it would take

I do not implement masking. Masking splits every secret value into random shares so that no single measured value correlates with the secret, and it is the real defence against DPA. Done properly it costs a large multiple in execution time, a substantial amount of randomness per operation, and a great deal of care, because a masking scheme with one unmasked intermediate value provides no protection at all while looking like it does.

Given the entropy situation in Chapter 9, a masking scheme that consumes fresh randomness per operation would be resting on the weakest component in the system. I judged that a well-used hardware engine plus limited operations per key was the better allocation of my effort. **If you are building this for production against a funded attacker, you need masking, and you need it evaluated by someone with a lab.**

17.5 Fault injection

Faults are the active version of the attack. Instead of listening, the attacker briefly pushes the chip outside its operating envelope so that an instruction produces the wrong result, and they time that push to hit an instruction that matters.

Method	Equipment	Typical effect
Voltage glitch	MOSFET and a microcontroller; tens of dollars	Instruction skipped or corrupted; register value wrong
Clock glitch	Signal generator	Setup-time violation; a fetched instruction decodes wrong
EM pulse	Coil and pulse generator; hundreds of dollars	Localised fault without touching the package
Laser	Optical bench, decapsulated die	Precise, single-bit, expensive; A-4 territory

The attacker’s dream target on this device is the branch that follows the PIN comparison. Skip one instruction and `if (!ok) return false;` becomes a no-op, and an unauthenticated attacker is authenticated. One glitch, no cryptography broken, complete bypass.

17.5.1 The countermeasures I use

Redundant checks with high-Hamming-distance sentinels.

Chapter 13’s `permission_check_hardened`. Two independent evaluations, compared, with sentinel values chosen so no single bit flip lands on a valid one.

Control-flow counters.

Chapter 13’s `FLOW_STAGE` macros. Skipping a stage produces a wrong sum, and the sum is checked before anything commits.

Random delays before critical operations.

Glitching requires triggering at a precise moment. A jittered delay before the comparison means the attacker’s trigger lands in a different place each time, so they must average over many attempts, and the failure counter from Chapter 11 is charging them for every one.

Double-checking the result after acting on it.

After the success branch sets `authenticated = true`, re-verify the condition that justified it. A glitch that got you into the branch has to also survive the recheck.

```

1  /* Jitter: the attacker cannot align a glitch to a moving target
2   * without averaging, and averaging costs them failure-counter
3   * increments they cannot avoid. */
4  random_delay_cycles(rng_byte() & 0x3Fu);
5
6  bool ok1 = ct_eq(candidate, stored, 32u);
7  random_delay_cycles(rng_byte() & 0x1Fu);
8  bool ok2 = ct_eq(candidate, stored, 32u); /* independent recompute */
9
10 secure_zero(candidate, sizeof candidate);
11 secure_zero(stored, sizeof stored);
12
13 /* Both must agree AND both must be true. A single glitched compare
14  * disagrees with the other and we fail closed. */
15 if (ok1 != ok2) { return false; }
```

```

16     if (!ok1) { return false; }
17
18     (void)state_set_fail_count(&g_state, 0u);
19     session_authenticate(user_id);
20
21     /* Post-condition recheck. If a glitch carried us here without a
22      * genuine match, this tears the session back down. */
23     if (!g_session.authenticated || g_session.user_id != user_id) {
24         session_reset();
25         return false;
26     }
27     return true;

```

Listing 17.1: The hardened authentication tail. Compare this against the plain version in Chapter 12.

Get this wrong and the device is broken

None of this makes the device glitch-proof. It raises the cost from one well-timed pulse to several correlated ones, which against a determined attacker with a good setup buys you time, not immunity. Real fault resistance needs hardware: voltage and clock monitors, redundant execution units, error-detecting memory. On a general-purpose microcontroller you are doing damage limitation, and a document that claimed otherwise would be lying to you.

What else I could have done

Duplicate the entire authentication path and compare final results, rather than duplicating individual checks. Stronger, and roughly doubles the code and the time. **Compute a checksum over the session structure** and verify it before every use, so that a glitched `authenticated` flag is caught. Cheap, and I would add it in a second revision. **Move the decision into the cryptography**, so that failing authentication means you cannot derive the key rather than meaning a flag is false. This is the strongest option by a distance, because there is no branch to skip: the key is either derivable or it is not. It requires restructuring so that the PIN is an input to key derivation rather than a gate before it, which is a design I would seriously consider if I built this again.

Constant-Time Discipline

18.1 The rule, and why it is a rule rather than a technique

The point

The execution time, the sequence of memory addresses accessed, and the sequence of branches taken must depend only on public values. Never on a secret. Not "usually not." Never.

It is a rule rather than a technique because it has to be applied everywhere or it does not help. One leaky comparison in one error path undoes every other careful line, and the leaky path is always the one nobody reviewed because it "just returns an error."

18.2 The toolkit

Constant-time programming means replacing control flow with arithmetic. Here is the complete set of primitives I use; everything else is built from these.

```

1  /* Build a mask: all ones if condition is true, all zeros if false,
2   * without a branch. The trick is that (0 - 1) underflows to all-ones
3   * in unsigned arithmetic. */
4  static inline uint32_t ct_mask(uint32_t cond_nonzero)
5  {
6      /* Normalise any nonzero value to exactly 1 first. */
7      uint32_t nz = (cond_nonzero | (0u - cond_nonzero)) >> 31;
8      return 0u - nz; /* 1 -> 0xFFFFFFFF, 0 -> 0x00000000 */
9  }
10
11 /* Select a or b without branching. Equivalent to cond ? a : b. */
12 static inline uint32_t ct_select(uint32_t cond_nonzero, uint32_t a, uint32_t b)
13 {
14     uint32_t m = ct_mask(cond_nonzero);
15     return (a & m) | (b & ~m);
16 }
17
18 /* Conditionally copy src over dst, in constant time, touching every
19 * byte regardless of whether the copy "happens". */
20 static inline void ct_copy(uint32_t cond_nonzero,
21                          uint8_t *dst, const uint8_t *src, size_t n)
22 {
23     uint8_t m = (uint8_t)ct_mask(cond_nonzero);
24     for (size_t i = 0u; i < n; i++) {
25         dst[i] = (uint8_t)((src[i] & m) | (dst[i] & (uint8_t)~m));
26     }
27 }
28
29 /* Constant-time "is a less than b" for unsigned 32-bit values.
30 * Returns 1 or 0 with no branch. */
31 static inline uint32_t ct_lt(uint32_t a, uint32_t b)
32 {
33     return (uint32_t)((((uint64_t)a - (uint64_t)b) >> 63) & 1u);
34 }

```

Listing 18.1: Constant-time primitives. Small, and worth memorising.

18.3 The compiler is not on your side

This is the part that catches people who have done everything else right.

A C compiler is permitted to transform your code into anything with the same observable behaviour as defined by the language standard, and **execution time is not observable behaviour as far as the standard is concerned**. An optimiser that recognises $(a \ \& \ m) \mid (b \ \& \ \sim m)$ as a conditional select is entirely within its rights to emit a branch, and modern optimisers are good at exactly this kind of recognition.

Four defences, in increasing order of reliability:

1. **Put constant-time code in its own translation unit** and compile it at `-O1`, so the aggressive passes that do this recognition never run on it.
2. **Use `volatile` to create an optimisation barrier** at the point where the value would be branched on.
3. **Read the disassembly**. Not once; as part of the build. Chapter 22 describes a build step that disassembles the constant-time module and fails the build if a conditional branch appears in it.
4. **Write it in assembly**. Total control, zero portability. Reserved for the two or three functions where it genuinely matters.

```

1  /* Force the compiler to materialise a value and forget what it knows
2  * about it. Without this, an optimiser that can prove diff is zero on
3  * some path is free to specialise that path into a branch. */
4  static inline uint8_t ct_barrier_u8(uint8_t v)
5  {
6      volatile uint8_t tmp = v;
7      return tmp;
8  }
9
10 bool ct_eq(const uint8_t *a, const uint8_t *b, size_t n)
11 {
12     uint8_t diff = 0u;
13     for (size_t i = 0u; i < n; i++) {
14         diff |= (uint8_t)(a[i] ^ b[i]);
15     }
16     diff = ct_barrier_u8(diff);
17     return (bool)((uint8_t)((((uint32_t)diff - 1u) >> 8) & 1u));
18 }

```

Listing 18.2: Optimisation barrier. Ugly, and necessary.

18.4 Secure zeroing, and why `memset` is not it

R-03 requires key material to be erased from SRAM after use. The obvious implementation does not work.

```

1  /* WRONG. The compiler observes that key is never read after the memset
2  * and that key is about to go out of scope, concludes the memset has no
3  * observable effect, and deletes it entirely. This is a well-known
4  * optimisation, it is standards-conforming, and it means your key is
5  * still sitting in RAM. This exact bug has shipped in real products. */
6  void bad_scrub(void)
7  {
8      uint8_t key[32];
9      derive_key(key);
10     use_key(key);
11     memset(key, 0, sizeof key); /* silently removed */
12 }
13

```

```

14  /* RIGHT. The volatile pointer makes each write an observable side
15   * effect that the compiler is not permitted to elide. */
16  void secure_zero(void *p, size_t n)
17  {
18      volatile uint8_t *v = (volatile uint8_t *)p;
19      while (n--) {
20          *v++ = 0u;
21      }
22  }

```

Listing 18.3: Why the obvious version fails, and what to do instead.

Three notes on this that matter in practice. C11 offers `memset_s` and it is the right answer where your toolchain has it, which on embedded targets is often not the case. `explicit_bzero` exists on some platforms and is equally good. And on ARM, a compiler barrier such as `__asm__ __volatile__("" ::: "memory")` after the loop is worth adding as belt and braces.

Get this wrong and the device is broken

Scrubbing the buffer you know about is the easy half. The hard half is the copies you did not think about: the value the compiler spilled to the stack, the register that still holds a key byte, the DMA buffer that was not scrubbed because it is "just a transfer buffer," and the stack frame of a function that returned. This is why Chapter 5 puts all secrets in a dedicated linker section and scrubs the whole region, and why the stack itself is worth scrubbing after any operation that handled key material.

18.5 Verifying that it worked

Reasoning is not evidence. Two ways to get evidence:

Statistical timing measurement. Run the operation many thousands of times with two classes of input, one fixed and one random, and test whether the timing distributions differ. This is the `dudect` methodology and it is the standard approach. A Welch t-test over the two samples gives you a number, and a $|t| > 4.5$ says you have a leak.

Cycle-accurate simulation. Run the code in an instruction-set simulator and count cycles exactly for different inputs. Slower to set up and gives an exact answer rather than a statistical one.

```

1  /* Measure two classes of input and compare the distributions. If the
2   * function is constant time, the two samples come from the same
3   * distribution and the t-statistic stays small. */
4  void test_ct_eq_timing(void)
5  {
6      enum { TRIALS = 100000 };
7      uint8_t fixed[32], random_in[32], reference[32];
8
9      memset(reference, 0xAA, sizeof reference);
10     memset(fixed, 0xAA, sizeof fixed); /* class 0: equal */
11
12     double t = dudect_run(TRIALS,
13         /* class 0 */ ^{ ct_eq(fixed, reference, 32); },
14         /* class 1 */ ^{ rng_fill(random_in, 32);
15             ct_eq(random_in, reference, 32); });
16
17     /* |t| > 4.5 is the conventional threshold for "there is a leak here". */
18     assert(fabs(t) < 4.5);
19 }

```

Listing 18.4: Timing leak test, host side. Run it against every function that touches a secret.

What else I could have done

Formal verification of the constant-time property is possible with tools such as ct-verif or the constant-time checkers built into some verification frameworks. They prove the property rather than sampling it, which is strictly better. They also require your code to fit the tool's supported subset, which embedded code with volatile peripheral access usually does not without work. For the two or three functions that genuinely matter, the effort would be justified in a production build, and it is on the list of things I would do with more time.

PART VIII

Building It

The C Implementation

19.1 Project layout

The structure below is not decoration. Each boundary exists so that a specific class of code can be tested independently of the hardware, and so that a reviewer can tell at a glance which files are dangerous.

```

1 neusense/
2   include/
3     hsm_types.h # shared types, no logic
4     hsm_config.h # ALL protocol and buffer constants, one place
5 portable/ # builds on host AND target. Fuzzable. Testable.
6   sha256.c # software SHA-256
7   hmac.c # HMAC-SHA-256
8   hkdf.c # RFC 5869
9   ct.c # constant-time primitives (compile at -O1)
10  parser.c # frame parsing (fuzz this)
11  record.c # storage record format
12  session.c # session state machine
13  permission.c # capability checks
14 target/ # target only. Touches registers. Not fuzzable.
15  startup.c # reset handler, vector table
16  clock.c
17  aes_hw.c # AESADV driver
18  dma.c
19  flash.c # program / erase / read
20  uart.c
21  entropy.c # Chapter 9 sources
22 app/
23   main.c # the loop from Chapter 5
24   commands.c # dispatch
25 test/
26   vectors/ # NIST KATs, checked in as data
27   unit/ # runs on host
28   fuzz/ # libFuzzer harnesses
29 tools/
30   provision.py # Chapter 24
31   check_ct.py # disassembly checker, Chapter 18
32 linker/
33 neusense.ld

```

Listing 19.1: Directory layout. The split between portable and target is the important one.

The point

Everything in portable/ compiles and runs on your laptop. That is what makes unit testing, fuzzing, coverage measurement, and timing analysis possible. If security-critical logic lives in a file that only builds for the target, you will test it far less than it deserves. Push logic out of target/ until only register access remains.

19.2 The single configuration header

Every size, every limit, every timeout, in one file, with static assertions tying them together. When somebody changes one, the assertions catch the combinations that no longer make sense.

```

1 #ifndef HSM_CONFIG_H

```

```

2  #define HSM_CONFIG_H
3
4  /* ---- Protocol ----- */
5  #define HSM_PROTO_VERSION 0x01u
6  #define HSM_FRAME_SYNC 0x4E53u
7  #define HSM_HDR_LEN 10u /* sync..length inclusive */
8  #define HSM_MAC_LEN 32u
9  #define HSM_MAX_PAYLOAD_LEN 1024u
10 #define HSM_MAX_FRAME_LEN (HSM_HDR_LEN + HSM_MAX_PAYLOAD_LEN + HSM_MAC_LEN)
11
12 /* ---- Authentication ----- */
13 #define HSM_PIN_MIN_LEN 6u
14 #define HSM_PIN_MAX_LEN 16u
15 #define HSM_PIN_STRETCH_ROUNDS 20000u
16 #define HSM_LOCKOUT_THRESHOLD 4u
17 #define HSM_SESSION_TIMEOUT_MS 300000u
18
19 /* ---- Storage ----- */
20 #define HSM_REC_SIZE 64u
21 #define HSM_REC_PAYLOAD_LEN 24u
22 #define HSM_LOG_SECTORS 2u
23 #define HSM_SECTOR_SIZE 1024u
24 #define HSM_RECS_PER_SECTOR (HSM_SECTOR_SIZE / HSM_REC_SIZE)
25
26 /* ---- Timing ----- */
27 #define HSM_RESPONSE_FLOOR_MS 25u
28 #define HSM_FRAME_TIMEOUT_MS 2000u
29 #define HSM_MAX_RESYNC_BYTES 4096u
30
31 /* ---- Consistency checks. These fire at compile time. ----- */
32 _Static_assert(HSM_MAX_FRAME_LEN <= 2048u,
33               "frame buffer would exceed the SRAM budget");
34 _Static_assert((HSM_SECTOR_SIZE % HSM_REC_SIZE) == 0u,
35               "records must tile a sector exactly, or the last one straddles");
36 _Static_assert(HSM_REC_PAYLOAD_LEN + 8u + HSM_MAC_LEN == HSM_REC_SIZE,
37               "record fields must sum to the record size");
38 _Static_assert(HSM_PIN_MIN_LEN >= 6u,
39               "shorter PINs are not defensible even with backoff");
40 _Static_assert(HSM_RESPONSE_FLOOR_MS >= 20u,
41               "floor must exceed the slowest legitimate path; measure it");
42
43 #endif

```

Listing 19.2: hsm_config.h. Change a number here and the build tells you what else is now wrong.

19.3 Command dispatch

```

1  typedef bool (*cmd_handler_fn)(const command_t *cmd, response_t *rsp);
2
3  typedef struct {
4      uint8_t opcode;
5      uint8_t required_cap; /* capability bit index, Chapter 13 */
6      bool needs_auth;
7      cmd_handler_fn handler;
8  } cmd_entry_t;
9
10 /* The table is const, so it lives in flash and cannot be modified by a
11  * stray write. A dispatch table in RAM is an attractive target. */
12 static const cmd_entry_t CMD_TABLE[] = {
13     { OP_GET_VERSION, CAP_NONE, false, handle_get_version },
14     { OP_GET_NONCE, CAP_NONE, false, handle_get_nonce },
15     { OP_AUTHENTICATE, CAP_NONE, false, handle_authenticate },
16     { OP_ENCRYPT, CAP_ENCRYPT, true, handle_encrypt },
17     { OP_DECRYPT, CAP_DECRYPT, true, handle_decrypt },
18     { OP_MAC, CAP_MAC, true, handle_mac },

```

```

19 { OP_READ_STATE, CAP_READ_STATE, true, handle_read_state },
20 { OP_REVOKE, CAP_REVOKE, true, handle_revoke },
21 { OP_SET_PIN, CAP_SET_PIN, true, handle_set_pin },
22 };
23 #define CMD_TABLE_LEN (sizeof CMD_TABLE / sizeof CMD_TABLE[0])
24
25 /* Linear scan, not an array index by opcode. A scan cannot be pushed
26  * out of bounds by an unexpected opcode value, and with nine entries
27  * the cost is irrelevant. */
28 static const cmd_entry_t *lookup(uint8_t opcode)
29 {
30     for (size_t i = 0; i < CMD_TABLE_LEN; i++) {
31         if (CMD_TABLE[i].opcode == opcode) {
32             return &CMD_TABLE[i];
33         }
34     }
35     return NULL;
36 }
37
38 void execute_command(const command_t *cmd)
39 {
40     response_t rsp;
41     memset(&rsp, 0, sizeof rsp);
42
43     const cmd_entry_t *e = lookup(cmd->opcode);
44     if (e == NULL) {
45         respond(false, NULL, 0u); /* generic. R-11. */
46         return;
47     }
48
49     if (e->needs_auth) {
50         if (permission_check_hardened(&g_session, e->required_cap)
51             != PERM_GRANTED) {
52             respond(false, NULL, 0u);
53             return;
54         }
55     }
56
57     bool ok = e->handler(cmd, &rsp);
58
59     /* Scrub before responding, not after. If the response path faults,
60      * the keys are already gone. */
61     scrub_secrets();
62     aes256_clear();
63
64     respond(ok, rsp.data, rsp.len);
65     secure_zero(&rsp, sizeof rsp);
66 }

```

Listing 19.3: Dispatch. A table, not a switch, so that adding a command means adding a row with an explicit capability.

Two choices in that listing are security-relevant. The table is `const` so it sits in flash rather than RAM, which means a stray write cannot repoint a handler. And lookup is a linear scan rather than `CMD_TABLE[opcode]`, which removes any possibility of an out-of-range index turning an unknown opcode into a jump to arbitrary memory.

19.4 A complete handler

```

1 static bool handle_encrypt(const command_t *cmd, response_t *rsp)
2 {
3     /* Payload layout: [2 bytes slot][2 bytes plaintext len][plaintext] */
4     if (cmd->len < 4u) { return false; }
5

```

```

6  uint16_t slot = be16_read(&cmd->payload[0]);
7  uint16_t pt_len = be16_read(&cmd->payload[2]);
8
9  /* Validate both bounds. Chapter 16 explains why one is not enough. */
10 if (slot >= HSM_MAX_SLOTS) { return false; }
11 if (pt_len == 0u) { return false; }
12 if ((size_t)pt_len + 4u != cmd->len) { return false; }
13 if (pt_len > HSM_MAX_PAYLOAD_LEN - 4u - 16u - 32u) { return false; }
14
15 const uint8_t *pt = &cmd->payload[4];
16
17 /* Derive the two keys for this slot under the current generation.
18  * Different labels: R-01's compartmentalisation in practice. */
19 uint8_t enc_key[32], mac_key[32];
20 uint32_t gen = state_get_generation(&g_state);
21 derive_op_key("neusense/enc/v1", slot, gen, enc_key);
22 derive_op_key("neusense/mac/v1", slot, gen, mac_key);
23
24 /* Counter-derived IV, so we do not lean on the RNG. Chapter 9. */
25 uint8_t iv[16];
26 derive_iv(slot, gen, state_get_msg_counter(&g_state), iv);
27
28 /* PKCS#7 padding into a static scratch buffer of known size. */
29 size_t padded_len;
30 if (!pad_apply(pt, pt_len, g_scratch, sizeof g_scratch, &padded_len)) {
31     goto fail;
32 }
33
34 /* Encrypt, then MAC. Never the other way round. Chapter 6. */
35 aes256_load_key(enc_key);
36 aes256_set_iv(iv);
37 aes256_cbc(AES_ENCRYPT, g_scratch, rsp->data + 16, padded_len);
38 memcpy(rsp->data, iv, 16u);
39
40 hmac_sha256_ctx c;
41 hmac_sha256_init(&c, mac_key, 32u);
42 hmac_sha256_update(&c, rsp->data, 16u + padded_len);
43 hmac_sha256_final(&c, rsp->data + 16u + padded_len);
44
45 rsp->len = (uint16_t)(16u + padded_len + 32u);
46
47 secure_zero(enc_key, sizeof enc_key);
48 secure_zero(mac_key, sizeof mac_key);
49 secure_zero(g_scratch, sizeof g_scratch);
50 aes256_clear();
51 return true;
52
53 fail:
54     secure_zero(enc_key, sizeof enc_key);
55     secure_zero(mac_key, sizeof mac_key);
56     secure_zero(g_scratch, sizeof g_scratch);
57     aes256_clear();
58     return false;
59 }

```

Listing 19.4: `handle_encrypt`, end to end. Every step traces to a requirement.

Get this wrong and the device is broken

Count the cleanup calls. There are two identical blocks, one on each exit path, and there are five `goto fail` sites in the full version of this function. Every one of them has to scrub. This is the structural weakness of C for this kind of code: correctness depends on a human remembering, at every exit, in every future edit. Chapter 20 exists because Rust

makes the compiler remember instead.

19.5 Measuring worst-case stack depth

Chapter 5 promised a method. Because there is no recursion and no function pointers outside the const dispatch table, the call graph is static and the worst-case depth is computable.

```
1 # 1. Ask the compiler to emit per-function stack usage.
2 arm-none-eabi-gcc -fstack-usage -c src/*.c
3
4 # 2. Each .su file has: file:line:col function_name bytes qualifier
5 # "static" means the compiler knows the exact number.
6 # "dynamic" or "bounded" means it does not, and you must fix that.
7 grep -h "dynamic" *.su && echo "FAIL: variable-length stack use found"
8
9 # 3. Walk the call graph and sum the worst path.
10 # Add the interrupt frame size, because an interrupt can land at the
11 # deepest point of the deepest path.
12 python3 tools/stack_depth.py --su-dir build/ --entry main \
13 --isr-overhead 64 --limit 4096
```

Listing 19.5: Static stack analysis. Run it in CI and fail the build if the number grows.

The `dynamic` check is the one that matters. A variable-length array or an `alloca` makes the depth input-dependent, which means an attacker chooses your stack usage, which means the guard band from Chapter 5 is your only defence. Ban both at the coding-standard level.

The Rust Implementation

20.1 Why a second implementation

Not to be fashionable. Three specific properties of Rust map onto three specific failure modes of the C code above, and I want to show the mapping concretely rather than assert that Rust is safer.

C failure mode	What Rust does about it
Buffer overflow in the parser	Slices carry lengths; indexing is checked; the compiler rejects code that cannot prove the bound
Forgotten key scrub on an error path	<code>Drop</code> runs on every exit, including early returns and panics. You cannot forget.
Using an uninitialised or stale session	Type-state: an unauthenticated session is a different <i>type</i> from an authenticated one, and the privileged function will not compile against the wrong one

The point

The value is not that Rust is memory safe in the abstract. It is that the three specific things this design most needs a human to remember are all things Rust's compiler can be made to remember instead. That converts a review burden into a build error.

20.2 Zeroize on drop

The C version needed two identical cleanup blocks and five `goto fail` sites. The Rust version needs none.

```

1  use core::ptr;
2  use core::sync::atomic::{compiler_fence, Ordering};
3
4  /// A 32-byte secret. Zeroed automatically whenever it goes out of
5  /// scope, by any route: normal return, early return, or panic unwind.
6  pub struct SecretKey([u8; 32]);
7
8  impl SecretKey {
9      pub fn new(bytes: [u8; 32]) -> Self {
10         SecretKey(bytes)
11     }
12
13     /// Deliberately not 'Deref'. Callers must ask explicitly, which
14     /// makes every use of raw key bytes visible in review.
15     pub fn expose(&self) -> &[u8; 32] {
16         &self.0
17     }
18 }
19
20 impl Drop for SecretKey {
21     fn drop(&mut self) {
22         // write_volatile is not removable by the optimiser, unlike a
23         // plain assignment or a memset the compiler can prove is dead.
24         for byte in self.0.iter_mut() {
25             unsafe { ptr::write_volatile(byte, 0u8) };
26         }
27     }
28 }

```

```

27 // Stop the compiler moving later memory operations before the
28 // zeroing, which would defeat the point.
29 compiler_fence(Ordering::SeqCst);
30 }
31 }
32
33 // No Copy and no Clone. A key that can be copied silently is a key that
34 // exists in places you did not scrub.

```

Listing 20.1: A key type that cannot outlive its own erasure.

Now compare the two encrypt functions. The C version had cleanup on every path. Here it is structural.

```

1 pub fn handle_encrypt(
2     session: &Session<Authenticated>,
3     payload: &[u8],
4     out: &mut [u8],
5 ) -> Result<usize, Error> {
6     // Slicing with checked patterns: if the payload is too short, this
7     // returns None and '?' turns it into an error. No manual length check
8     // to forget, and no possibility of reading past the end.
9     let slot = u16::from_be_bytes(payload.get(0..2).ok_or(Error::Malformed)?
10                                     .try_into().unwrap());
11     let pt_len = u16::from_be_bytes(payload.get(2..4).ok_or(Error::Malformed)?
12                                     .try_into().unwrap()) as usize;
13
14     if slot as usize >= MAX_SLOTS { return Err(Error::Malformed); }
15     let pt = payload.get(4..4 + pt_len).ok_or(Error::Malformed)?;
16
17     let gen = storage::generation();
18
19     // Both keys are SecretKey. From this line on, every 'return' and
20     // every '?' erases them on the way out. There is no path that does not.
21     let enc_key = derive_op_key(b"neusense/enc/v1", slot, gen)?;
22     let mac_key = derive_op_key(b"neusense/mac/v1", slot, gen)?;
23
24     let iv = derive_iv(slot, gen, storage::msg_counter())?;
25
26     let mut scratch = [0u8; MAX_PAYLOAD_LEN];
27     let padded = pad_apply(pt, &mut scratch)?; // early return: keys erased
28
29     let (iv_out, rest) = out.split_at_mut(16);
30     iv_out.copy_from_slice(&iv);
31
32     let ct = rest.get_mut(..padded.len()).ok_or(Error::TooLong)?;
33     aes::cbc_encrypt(&enc_key, &iv, padded, ct)?;
34
35     let tag = hmac::compute(&mac_key, &out[..16 + padded.len()]);
36     out[16 + padded.len()..16 + padded.len() + 32].copy_from_slice(&tag);
37
38     Ok(16 + padded.len() + 32)
39     // enc_key and mac_key drop here, and on every '?' above.
40 }

```

Listing 20.2: handle_encrypt in Rust. Five early returns, zero cleanup code, and the keys are erased on all six exits.

20.3 Type-state for the session

This is the technique I find most valuable and the one least often shown, so it gets a full treatment.

The idea: encode the session's state in the type system, so that calling a privileged operation on an unauthenticated session is a compile error rather than a runtime check that somebody

might forget or that a glitch might skip.

```

1  use core::marker::PhantomData;
2
3  pub struct Unauthenticated;
4  pub struct Authenticated;
5
6  pub struct Session<S> {
7      user_id: u8,
8      caps: u8,
9      generation: u32,
10     last_activity_ms: u32,
11     _state: PhantomData<S>,
12 }
13
14 impl Session<Unauthenticated> {
15     pub fn new() -> Self {
16         Session { user_id: 0, caps: 0, generation: 0,
17                   last_activity_ms: 0, _state: PhantomData }
18     }
19
20     /// Consumes the unauthenticated session. On success you get an
21     /// authenticated one; on failure you get the unauthenticated one
22     /// back. Either way the old value is gone, so there is no way to
23     /// keep using a stale session object.
24     pub fn authenticate(
25         self,
26         user_id: u8,
27         pin: &[u8],
28     ) -> Result<Session<Authenticated>, Session<Unauthenticated>> {
29         match verify_pin(user_id, pin) {
30             Ok(caps) => Ok(Session {
31                 user_id,
32                 caps,
33                 generation: storage::generation(),
34                 last_activity_ms: timer::millis(),
35                 _state: PhantomData,
36             }),
37             Err(_) => Err(self),
38         }
39     }
40 }
41
42 impl Session<Authenticated> {
43     pub fn has_cap(&self, cap: Capability) -> bool {
44         // Still checked at runtime, because capabilities are data.
45         // But *authentication* is checked at compile time.
46         (self.caps >> (cap as u8)) & 1 == 1
47         && self.generation == storage::generation()
48     }
49
50     pub fn logout(self) -> Session<Unauthenticated> {
51         Session::new()
52     }
53 }
54
55 // The signature enforces the rule. There is no runtime check to skip,
56 // no flag to glitch, and no way to call this with the wrong session.
57 pub fn handle_revoke(session: &Session<Authenticated>) -> Result<(), Error> {
58     if !session.has_cap(Capability::Revoke) {
59         return Err(Error::Denied);
60     }
61     storage::increment_generation()
62 }

```

Listing 20.3: Type-state sessions. The unauthenticated session has no method that performs a privileged operation, so no such call can be written.

The point

`handle_revoke` takes `&Session<Authenticated>`. There is no way to obtain one except by a successful `authenticate`. The check that Chapter 13 spent a page hardening against fault injection has, for this one property, been moved from runtime to compile time, where a glitch cannot reach it. That is not a small difference.

20.4 Constant-time code in Rust

Rust’s optimiser is LLVM, and LLVM has exactly the same freedom to turn your branchless code into a branch that a C compiler has. The language does not help here. What helps is the same discipline plus the same barriers.

```

1  use core::ptr;
2
3  /// Compare two byte slices in time depending only on their length.
4  /// Returns false immediately if the lengths differ, because length is
5  /// not secret and pretending otherwise would be theatre.
6  pub fn ct_eq(a: &[u8], b: &[u8]) -> bool {
7      if a.len() != b.len() {
8          return false;
9      }
10
11     let mut diff: u8 = 0;
12     for i in 0..a.len() {
13         diff |= a[i] ^ b[i];
14     }
15
16     // read_volatile stops LLVM reasoning about diff's value and
17     // specialising a branch on it.
18     let diff = unsafe { ptr::read_volatile(&diff) };
19     diff == 0
20 }
21
22 /// Branchless select. Equivalent to 'if c { a } else { b }'.
23 #[inline(always)]
24 pub fn ct_select_u32(cond: bool, a: u32, b: u32) -> u32 {
25     let mask = (cond as u32).wrapping_neg(); // true -> !0, false -> 0
26     (a & mask) | (b & !mask)
27 }

```

Listing 20.4: Constant-time equality with an explicit optimisation barrier.

In production I would use the `subtle` crate, which provides `ConstantTimeEq` and `Choice` with the barriers already in place and reviewed by people who do this full time. I have written it out here so you can see what it does rather than trusting it blindly.

20.5 Panics, and `no_std`

Two Rust-specific hazards on this target.

Panics. A Rust panic on an embedded target reaches a `panic_handler` you write. The default behaviour of most handlers is to loop forever, which for us is an attacker-triggerable permanent denial of service, violating [R-14](#). My handler scrubs secrets and resets.

```

1  #![no_std]
2  #![no_main]
3
4  use core::panic::PanicInfo;
5
6  #[panic_handler]

```

```

7 fn panic(_info: &PanicInfo) -> ! {
8     // Do not format or transmit _info. It can contain array indices,
9     // lengths, and file paths, all of which are reconnaissance.
10    unsafe {
11        secrets::scrub_all();
12        aes::clear_engine();
13    }
14    // Reset rather than loop. A hung device is a bricked device, and
15    // a bricked device is the attacker's win condition G-6.
16    cortex_m::peripheral::SCB::sys_reset()
17 }

```

Listing 20.5: The panic handler. It must never leak, and it must never hang.

Build with `panic = "abort"` in the release profile so that no unwinding machinery is linked in. It costs code space we do not have, and unwinding through a function holding a key is a code path I would rather not have to reason about.

Bounds-check cost. Rust inserts bounds checks on slice indexing. On this core they cost a compare and a branch, and there is a temptation to reach for `get_unchecked` to remove them. Do not do this in the parser. The measured cost of bounds checking on a device processing a few commands per second is invisible, and `get_unchecked` in a parser reintroduces exactly the bug class you switched languages to eliminate.

What else I could have done

Stay in C and use MISRA C plus static analysis. A well-configured analyser catches a real fraction of what Rust's compiler catches, and it does not require your team to learn a language. It does not catch the forgotten scrub, and it does not give you type-state. **Use Ada or SPARK**, which gives you provable absence of runtime errors, and which is genuinely stronger than Rust for this. The obstacle is tooling and ecosystem for this class of part. **Write the parser in Rust and everything else in C**, linking them together. This is a real strategy, it concentrates the benefit where the risk is highest, and it is what I would recommend to a team that cannot rewrite everything.

Porting This Design

21.1 What is portable and what is not

Portable without change	Must be reconsidered per platform
Threat model structure (Ch. 2)	Which attacker level you can actually resist
Requirements (Ch. 3)	Nothing; these should survive any port
Key hierarchy (Ch. 8)	Nothing; HKDF is HKDF
Encrypt-then-MAC (Ch. 6)	Whether a better AEAD is available
Append-only log (Ch. 10)	Erase granularity, program granularity, endurance
Counter ordering (Ch. 11)	Nothing; the ordering rule is universal
Fail-closed permissions (Ch. 13)	Nothing
Parser discipline (Ch. 16)	Nothing
Constant-time primitives (Ch. 18)	Compiler barriers; cache behaviour

The point

The architecture ports completely. The hardware assumptions do not. When you move this design, rewrite Chapter 4 for your part first, and let the differences you find there drive every other change. Do not start by porting the code.

21.2 Porting to a different microcontroller

A checklist, in the order I would work through it:

1. **Does it have a hardware AES engine?** If yes, write the driver behind the four-function interface from Chapter 7 and nothing above changes. If no, you need a constant-time software AES, and you should budget a week for implementation plus testing against the full NIST vector set.
2. **Does it have a real TRNG with health tests?** If yes, delete most of Chapter 9 and use it, still stirred through a pool. If no, keep Chapter 9 and take the counter-based-IV advice seriously.
3. **What is the flash program granularity?** If you cannot program 64 bytes atomically, reorder the record so the MAC is written in a separate, later operation. The design already assumes this; make sure the code does too.
4. **Is there an MPU?** If yes, use one region for a stack guard, one to make the secrets section inaccessible outside crypto code, and one to make RAM non-executable. This is the single biggest security improvement available on most parts.
5. **Is there a cache?** If yes, **R-16** stops being free. Audit every table lookup for secret-dependent indices.
6. **Is there a hardware monotonic counter or a secure element?** If yes, move the anti-rollback state there and delete half of Chapter 11.
7. **What is the debug interface and can it be permanently disabled?** Chapter 24.

21.3 Porting to another language

21.3.1 Python: for the host tools and the model

Python has no place in the firmware, and a large place in the toolchain. I write a complete reference model of the device in Python and use it as the oracle for differential testing: feed the same commands to the model and the firmware, and any divergence is a bug in one of them.

```

1 import hashlib, hmac, struct
2
3 def hkdf_extract(salt: bytes, ikm: bytes) -> bytes:
4     if not salt:
5         salt = b"\x00" * 32
6     return hmac.new(salt, ikm, hashlib.sha256).digest()
7
8 def hkdf_expand(prk: bytes, info: bytes, length: int) -> bytes:
9     """RFC 5869 expand. Written for clarity, not speed: this is the
10    thing the firmware gets compared against, so it must be readable
11    enough that a reviewer can confirm it matches the RFC by eye."""
12    okm, t, counter = b"", b"", 1
13    while len(okm) < length:
14        t = hmac.new(prk, t + info + bytes([counter]), hashlib.sha256).digest()
15        okm += t
16        counter += 1
17    return okm[:length]
18
19 class DeviceModel:
20     """Mirrors the firmware's state machine exactly. Any behaviour the
21     firmware has that this does not is undocumented behaviour."""
22
23     def __init__(self, root_key: bytes, device_id: bytes):
24         self.prk = hkdf_extract(salt=device_id, ikm=root_key)
25         self.generation = 0
26         self.fail_count = 0
27         self.msg_counter = 0
28         self.authenticated = False
29         self.caps = 0
30
31     def derive_op_key(self, label: bytes, slot: int) -> bytes:
32         info = label + struct.pack(">HI", slot, self.generation)
33         return hkdf_expand(self.prk, info, 32)
34
35     def authenticate(self, user_id: int, pin: bytes) -> bool:
36         # Increment before checking. Same ordering as the firmware,
37         # because the model must reproduce the firmware's behaviour
38         # under power loss too, not just under normal operation.
39         self.fail_count += 1
40         expected = self._verifier(user_id, pin)
41         if hmac.compare_digest(expected, self._stored_verifier(user_id)):
42             self.fail_count = 0
43             self.authenticated = True
44             self.caps = self._caps_for(user_id)
45             return True
46         return False
47
48     def revoke(self) -> None:
49         self.generation += 1
50         self.authenticated = False
51         self.caps = 0

```

Listing 21.1: The reference model. Slow, obviously correct, and the ground truth for every test.

21.3.2 Go: for the host-side application

Go is a good fit for the host application that drives the device: strong standard-library cryptography, easy concurrency for a service, and memory safety. It is not a fit for the firmware, because

the garbage collector makes timing unpredictable and it needs a runtime we cannot afford.

```

1 package hsm
2
3 import (
4     "crypto/hmac"
5     "crypto/sha256"
6     "encoding/binary"
7     "errors"
8 )
9
10 const (
11     ProtoVersion = 0x01
12     FrameSync    = 0x4E53
13     HdrLen       = 10
14     MacLen       = 32
15     MaxPayloadLen = 1024
16 )
17
18 // BuildFrame assembles and authenticates a command frame.
19 // The counter must be strictly greater than the last one the device
20 // accepted, or it will be rejected as a replay.
21 func BuildFrame(sessionKey []byte, opcode uint8,
22     counter uint32, payload []byte) ([]byte, error) {
23
24     if len(payload) > MaxPayloadLen {
25         return nil, errors.New("payload exceeds protocol maximum")
26     }
27
28     buf := make([]byte, HdrLen+len(payload)+MacLen)
29     binary.BigEndian.PutUint16(buf[0:2], FrameSync)
30     buf[2] = ProtoVersion
31     buf[3] = opcode
32     binary.BigEndian.PutUint32(buf[4:8], counter)
33     binary.BigEndian.PutUint16(buf[8:10], uint16(len(payload)))
34     copy(buf[HdrLen:], payload)
35
36     m := hmac.New(sha256.New, sessionKey)
37     m.Write(buf[:HdrLen+len(payload)])
38     copy(buf[HdrLen+len(payload):], m.Sum(nil))
39
40     return buf, nil
41 }
42
43 // VerifyResponse checks the device's reply. hmac.Equal is constant time;
44 // using bytes.Equal here would leak the tag one byte at a time to
45 // anybody who can measure the host, which in some deployments is
46 // exactly the attacker.
47 func VerifyResponse(sessionKey, frame []byte) error {
48     if len(frame) < 5+MacLen {
49         return errors.New("short frame")
50     }
51     body := frame[:len(frame)-MacLen]
52     tag := frame[len(frame)-MacLen:]
53
54     m := hmac.New(sha256.New, sessionKey)
55     m.Write(body)
56     if !hmac.Equal(m.Sum(nil), tag) {
57         return errors.New("authentication failed")
58     }
59     return nil
60 }

```

Listing 21.2: Host-side frame construction. Note the explicit big-endian encoding, matching Chapter 15.

21.3.3 Zig: a middle path

Zig deserves a mention because it targets exactly this niche: manual memory management like C, no hidden control flow, no runtime, and compile-time bounds checking in safe build modes. It gives you a meaningful fraction of Rust’s safety benefit with a much smaller conceptual jump from C, and it has excellent support for freestanding targets.

```

1  const std = @import("std");
2
3  pub const ParseError = error{ Malformed, BadVersion, BadMac };
4
5  pub fn parseCommand(frame: []const u8, out: *Command) ParseError!void {
6      // Length checks first, exactly as in the C version.
7      if (frame.len < HDR_LEN + MAC_LEN) return ParseError.Malformed;
8      if (std.mem.readIntBig(u16, frame[0..2]) != FRAME_SYNC)
9          return ParseError.Malformed;
10     if (frame[2] != PROTO_VERSION) return ParseError.BadVersion;
11
12     const plen = std.mem.readIntBig(u16, frame[8..10]);
13     if (plen > MAX_PAYLOAD_LEN) return ParseError.Malformed;
14     if (HDR_LEN + @as(usize, plen) + MAC_LEN != frame.len)
15         return ParseError.Malformed;
16
17     // Slicing is bounds-checked in Debug and ReleaseSafe. Build the
18     // firmware in ReleaseSafe, not ReleaseFast: the checks are the
19     // reason you chose this language.
20     const body = frame[0 .. HDR_LEN + plen];
21     const mac = frame[HDR_LEN + plen ..];
22
23     if (!verifyMac(body, mac)) return ParseError.BadMac;
24
25     out.opcode = frame[3];
26     out.counter = std.mem.readIntBig(u32, frame[4..8]);
27     out.len = plen;
28     @memcpy(out.payload[0..plen], frame[HDR_LEN..][0..plen]);
29 }

```

Listing 21.3: Zig parser fragment. Slices carry their length, and the arithmetic is checked in safe modes.

21.3.4 What not to use

Language	Why not, for the firmware
Python, JS	Interpreted; no deterministic timing; runtime far larger than the flash
Go	Garbage collector makes timing unpredictable, which breaks Chapter 18
Java, C#	Same, plus a runtime measured in megabytes
C++	Usable, but exceptions and hidden allocations in the standard library need careful exclusion; if you go this way, use a strict freestanding subset and ban the STL

21.4 Porting the architecture, not the code

If you take this design somewhere genuinely different, a server-side HSM service or a mobile secure enclave, most of the code is irrelevant and most of the document still applies. What travels is:

1. Write the threat model first, with an explicit capability ladder and an explicit line you are not defending past.

2. Number the requirements and make every one testable.
3. One key, one purpose, derived not stored.
4. Encrypt then MAC, verify before decrypt.
5. Increment the failure counter before the check, persistently.
6. Deny by default, in every path, including the ones you did not anticipate.
7. Make the parser total, and fuzz it.
8. Never branch on a secret.
9. Erase secrets on every exit path, and prefer a language that does it for you.

Those nine survive any port. Everything else in this document is the specific shape those nine took on one particular small chip.

PART IX

Proving It

Test Strategy

22.1 Five layers, each catching what the one below misses

Layer	Runs on	Catches
Known-answer tests	Host and target	A cryptographic primitive computing the wrong thing
Unit tests	Host	Logic errors in parsing, state machines, permissions
Property tests	Host	Invariant violations across generated input, including cases nobody thought of
Fuzzing	Host	Memory-safety bugs and unhandled input
Hardware-in-loop	Target	Timing, power-loss, and glitch behaviour that only exists on real silicon

The layers are ordered by cost and inverted by coverage: the cheap ones run on every commit, the expensive ones run nightly, and the top one runs before a release.

22.2 Known-answer tests

Use published vectors. Never generate your own with the implementation you are testing, because that verifies only that your code agrees with itself.

Primitive	Source	Minimum coverage
AES-256-CBC	NIST SP 800-38A	Both directions, multi-block, all provided vectors
SHA-256	NIST CAVP / FIPS 180-4	Empty input, one block, boundary at 55 and 56 bytes, multi-block
HMAC-SHA-256	RFC 4231	All seven test cases, including short and over-long keys
HKDF	RFC 5869	All test cases, including empty salt and empty info

The 55 and 56 byte SHA-256 cases are worth calling out specifically. That is where the length padding crosses a block boundary, and it is the single most common place a hand-written SHA-256 is wrong while passing every other test.

22.3 Property-based tests

Unit tests check the cases you thought of. Property tests check thousands of cases the generator thought of, against invariants you state once.

```

1 from hypothesis import given, strategies as st
2
3 @given(st.binary(min_size=1, max_size=1000))
4 def test_roundtrip(plaintext):
5     """Encrypt then decrypt returns the original. The obvious one, and
```

```

6     still worth having, because padding bugs hide at odd lengths."""
7     ct = model.encrypt(slot=0, data=plaintext)
8     assert model.decrypt(slot=0, data=ct) == plaintext
9
10    @given(st.binary(min_size=1, max_size=1000), st.integers(0, 999))
11    def test_tampering_always_detected(plaintext, bit_index):
12        """Flipping any single bit of any ciphertext must fail the MAC.
13        This is the property CBC alone does not give you."""
14        ct = bytearray(model.encrypt(slot=0, data=plaintext))
15        idx = bit_index % (len(ct) * 8)
16        ct[idx // 8] ^= (1 << (idx % 8))
17        assert model.decrypt(slot=0, data=bytes(ct)) is None
18
19    @given(st.integers(0, 255), st.integers(0, 255))
20    def test_distinct_slots_give_distinct_keys(slot_a, slot_b):
21        """Compartmentalisation, stated as a property. If this ever fails,
22        the key hierarchy's context binding is broken."""
23        if slot_a != slot_b:
24            assert model.derive_op_key(b"enc", slot_a) != \
25                model.derive_op_key(b"enc", slot_b)
26
27    @given(st.binary(max_size=3000))
28    def test_parser_never_accepts_garbage(data):
29        """The parser either rejects, or accepts something that satisfies
30        every invariant the rest of the firmware relies on. There is no
31        third outcome."""
32        result = model.parse(data)
33        if result is not None:
34            assert result.len <= MAX_PAYLOAD_LEN
35            assert result.opcode in KNOWN_OPCODES
36
37    @given(st.lists(st.integers(0, 2**32 - 1), min_size=1, max_size=50))
38    def test_counter_never_decreases(sequence):
39        """Anti-rollback, as a property. Feed arbitrary counter values and
40        confirm the stored value is monotonic regardless."""
41        last = model.msg_counter
42        for c in sequence:
43            model.try_command(counter=c)
44            assert model.msg_counter >= last
45            last = model.msg_counter

```

Listing 22.1: Properties worth stating. Each one has caught a real bug in something, somewhere.

22.4 Hardware-in-the-loop

Some properties do not exist off the target. Three test rigs cover them.

22.4.1 The power-cut rig

A microcontroller controls a MOSFET on the device's power rail, and a GPIO on the device signals when the operation under test begins. The rig cuts power at a delay that sweeps across the whole operation, in small increments, and after each cut it powers the device back up and reads its state.

```

1    def test_power_cut_sweep_authenticate():
2        """Cut power at every point across the authenticate command and
3        verify that the failure counter never went backwards.
4
5        This is the test for the attack in Chapter 11: an attacker who cuts
6        power before the counter is written gets a free guess. If our
7        ordering is right, no cut point produces a free guess."""
8
9        baseline = device.read_fail_count()
10
11        for delay_us in range(0, AUTH_MAX_DURATION_US, 10):

```

```

12     rig.power_on()
13     device.wait_ready()
14     before = device.read_fail_count()
15
16     rig.arm_cut_after_trigger(delay_us)
17     device.send_authenticate(user=0, pin=b"000000") # deliberately wrong
18     rig.wait_for_cut()
19
20     rig.power_on()
21     device.wait_ready()
22     after = device.read_fail_count()
23
24     # The counter may have been incremented, or the whole attempt may
25     # have been lost before it started. It must never have gone DOWN,
26     # and the device must never be usable without the attempt counting.
27     assert after >= before, (
28         f"ROLLBACK at {delay_us} us: {before} -> {after}")
29
30     assert device.read_fail_count() >= baseline

```

Listing 22.2: Power-cut sweep. This is the test that validates R-06 and R-13, and nothing else can.

22.4.2 The timing rig

A logic analyser on a GPIO the firmware toggles at the start and end of the operation under test. Thousands of trials, two input classes, a Welch t-test on the durations. Chapter 18 has the method; the rig is what makes it run against real silicon rather than a simulator.

22.4.3 The glitch rig

A voltage glitcher, a trigger from the same GPIO, and a sweep across glitch offset and glitch width. Success is measured by whether the device ever reports authentication success for a wrong PIN. Expect this to take a long time and expect to find something; that is what the countermeasures in Chapter 17 are for and the rig is how you learn whether they held.

Attacking Your Own Device

23.1 Why you do this yourself first

Somebody is going to run these attacks. It is much better for that person to be you, on your own bench, three weeks before the deadline, than for it to be another team on the day.

Each attack below states the goal from Chapter 2, the method, the expected result if the defence works, and what it means if it does not.

23.2 The playbook

23.2.1 A-1: parser abuse

Attack	Expected result
Length field claims 1024, frame contains 20 bytes	Rejected at the length cross-check. No read past the buffer.
Length field claims 0xFFFF	Rejected against <code>HSM_MAX_PAYLOAD_LEN</code> .
Valid header, MAC truncated by one byte	Rejected. Total length mismatch.
Unknown opcode with a valid MAC	Rejected at the dispatch lookup, generic error.
Frame sent one byte per second	Frame timeout fires. Device remains responsive to other traffic afterwards.
Ten megabytes of 0x00	Resync limit fires. Device recovers.

23.2.2 G-4: rollback

1. Fail authentication three times. Read the counter. Power cycle. Confirm it is still three.
2. Erase the log sector externally with a debugger. Confirm the device comes up in the most restrictive state, not the most permissive.
3. Copy a valid old record into a later slot with a debugger. Confirm it fails verification because of the offset binding in the MAC.
4. Run the power-cut sweep from Chapter 22 across the counter write. Confirm no cut point yields a free attempt.

23.2.3 G-5: replay

1. Capture a valid encrypt command. Resend it verbatim. Confirm rejection.
2. Resend it after a power cycle. Confirm rejection, because the counter is persistent.
3. Send it to a second device provisioned from the same root. Confirm rejection, because the session key binds the device identity.
4. Send a command with the counter set to 0xFFFFFFFF. Confirm rejection at the forward-jump bound, and confirm the device still accepts the next legitimate command afterwards.

23.2.4 A-2: timing

1. Measure the response time for a wrong PIN differing in the first byte versus the last byte. Ten thousand trials each. Expect no statistically significant difference.
2. Measure a known user identifier against an unknown one. Expect no difference, because of the dummy-verification path in Chapter 12.
3. Measure a MAC failure against a length failure. Expect no difference, because of the response floor in Chapter 15.

Get this wrong and the device is broken

When a timing test comes back showing no leak, check that your measurement is actually sensitive enough to detect one. Insert a deliberate leak, an early-exit `memcmp`, and confirm your rig detects it. A test that passes because it cannot see anything is worse than no test, because it produces false confidence. I have watched a team pass a timing audit with a rig that was measuring the host's scheduler rather than the device.

23.2.5 A-3: fault injection

1. Sweep glitch offset across the whole authenticate command, at several widths. Look for any authentication success with a wrong PIN.
2. Target the permission check specifically, using the trigger GPIO to align. Look for any privileged operation succeeding from an unauthenticated session.
3. Target the flash write during the counter increment. Look for a torn record that still verifies.
4. Reset the device at high frequency during operation and confirm it always comes up in UNAUTH.

23.3 What to do when you find something

Write it down before you fix it. The record should say what you did, what happened, why the defence failed, what you changed, and how you now test for it. That last field is the one people skip and it is the one that stops the bug coming back.

Field	Content
Finding	One sentence. What can an attacker do that they should not be able to?
Goal	Which of G-1 to G-6 it achieves.
Capability	Which attacker level is required. A finding needing A-4 is a documented limit; one needing A-1 is an emergency.
Root cause	The specific line or design decision, not "insufficient validation."
Fix	What changed.
Regression test	The test that now fails on the old code and passes on the new. Not optional.

Build, Provisioning, and Release

24.1 Reproducible builds

If two builds of the same source produce different binaries, you cannot tell whether the binary on a device is the one you reviewed. For a security device that is a real problem, and the fix is mostly mechanical.

```

1 CFLAGS = \
2   -mcpu=cortex-m0plus -mthumb \
3   -Os \
4   -std=c11 -Wall -Wextra -Werror \
5   -Wconversion -Wsign-conversion \
6   -Wshadow -Wcast-align -Wundef \
7   -Wstrict-prototypes \
8   -fno-common \
9   -ffunction-sections -fdata-sections \
10  -fstack-usage \
11  -fno-strict-aliasing \
12  -ffile-prefix-map=$(PWD)=. \
13  -Werror=vla \
14  -Werror=implicit-function-declaration
15
16  # The constant-time module is built separately, at -O1, so that the
17  # aggressive passes that turn branchless code into branches never run.
18  CT_CFLAGS = $(filter-out -Os,$(CFLAGS)) -O1
19
20  LDFLAGS = -Wl,--gc-sections -Wl,-Map=build/neusense.map -nostartfiles

```

Listing 24.1: Build flags. Every one of these is either a security control or a reproducibility control.

Flag	Why it is there
-Wconversion	Catches silent narrowing, which is how a validated 32-bit length becomes an unvalidated 16-bit one
-Werror=vla	Variable-length arrays make stack depth input-dependent; see Chapter 19
-fno-strict-aliasing	Removes a class of surprising optimisation in code that does byte-level manipulation
-ffile-prefix-map	Strips absolute paths so the binary does not depend on where it was built
-fno-common	Makes duplicate definitions a link error rather than a silent merge
-fstack-usage	Emits the data Chapter 19's depth analysis needs

```

1  # Build twice, in different directories, and compare.
2  make clean && make && sha256sum build/neusense.bin > /tmp/a
3  make clean && make && sha256sum build/neusense.bin > /tmp/b
4  diff /tmp/a /tmp/b || { echo "BUILD NOT REPRODUCIBLE"; exit 1; }
5
6  # Confirm the constant-time module contains no conditional branches.
7  arm-none-eabi-objdump -d build/ct.o \
8  | python3 tools/check_ct.py --forbid beq,bne,bcs,bcc,bmi,bpl,bhi,bls \
9  || { echo "CONDITIONAL BRANCH IN CONSTANT-TIME CODE"; exit 1; }

```

```

10
11 # Confirm no secret ended up outside the .secrets section.
12 python3 tools/check_secrets_section.py build/neusense.map

```

Listing 24.2: Reproducibility check. Run it in CI; a mismatch fails the build.

24.2 Provisioning

Provisioning happens once per device, on a trusted machine, and it is the moment when the entire security of the device is established. Assumption one of the threat model rests here.

```

1 import os, secrets, hashlib, hmac, struct, json
2
3 def provision(device_id: bytes, initial_pin: bytes, out_manifest: str):
4     """Generate and install one device's root secret.
5
6     The root key is generated here, written to the device, and then
7     discarded from this process's memory. It is stored only in the
8     encrypted manifest, and only because a lost root means a device
9     that can never be re-provisioned or audited.
10    """
11    assert len(device_id) == 16
12
13    # os.urandom, not random. The whole device rests on this line.
14    root_key = secrets.token_bytes(32)
15
16    # Compute the initial PIN verifier here, so the plaintext PIN never
17    # travels over the provisioning link.
18    prk = hmac.new(device_id, root_key, hashlib.sha256).digest()
19    pin_key = hkdf_expand(prk, b"neusense/pin/v1" + b"\x00", 32)
20    salt = secrets.token_bytes(16)
21    verifier = stretch(pin_key, salt, initial_pin, rounds=20000)
22
23    # Write, then read back a derived proof rather than the secret
24    # itself. The device proves it has the right root by MACing a
25    # challenge; it never sends the root, and there is no command
26    # that would let it.
27    programmer.write_region(ROOT_ADDR, root_key)
28    programmer.write_region(DEVID_ADDR, device_id)
29    programmer.write_region(PIN_ADDR, salt + verifier)
30
31    challenge = secrets.token_bytes(32)
32    response = device.prove_root(challenge)
33    expected = hmac.new(prk, challenge, hashlib.sha256).digest()
34    if not hmac.compare_digest(response, expected):
35        raise RuntimeError("provisioning verification failed; device rejected")
36
37    # Permanently disable the debug port. AFTER verification, because
38    # this is irreversible and a device that failed verification should
39    # be diagnosable.
40    programmer.set_debug_lock_fuse()
41
42    # Confirm the lock took. A device shipped with an open debug port
43    # is a device with no security at all.
44    if programmer.debug_accessible():
45        raise RuntimeError("debug lock did not take; do not ship this unit")
46
47    save_encrypted_manifest(out_manifest, {
48        "device_id": device_id.hex(),
49        "root_key": root_key.hex(), # encrypted at rest by the caller
50        "provisioned_at": iso_now(),
51        "firmware_sha256": firmware_digest(),
52    })

```

Listing 24.3: Provisioning tool. Note what it does not do: it never writes the root key to disk unencrypted, and it never reads it back from the device.

Get this wrong and the device is broken

The debug lock is the last step and it is the one people skip in a hurry, because a locked device is harder to debug and the deadline is tomorrow. A device with an accessible debug port has no security whatsoever: an attacker attaches a debugger, halts the core, and reads the root key out of flash in about four seconds. Every other control in this document is defeated by that one omission. Automate the lock, verify the lock, and refuse to ship any unit where the verification did not pass.

24.3 Release checklist

Nothing ships until every line is checked by a person and initialled.

Item
☐ All known-answer tests pass on the target, not just the host
☐ Fuzzer has run at least 50 million executions with no new findings
☐ Static stack analysis is under the limit, with no <code>dynamic</code> entries
☐ Constant-time modules disassembled and confirmed branch-free
☐ Timing tests show $ t < 4.5$ on every secret-handling function
☐ Power-cut sweep passes across authenticate and every flash write
☐ Glitch sweep run, findings documented and accepted
☐ No debug or verbose-error build flags in the release configuration
☐ Debug port lock fuse set and verified on a sample unit
☐ Build reproduces byte-for-byte across two clean builds
☐ <code>.secrets</code> section contains everything it should and nothing else
☐ Threat model reviewed against the final feature set
☐ Requirement traceability matrix complete, no unmapped requirements
☐ Every finding from the self-attack playbook has a regression test

24.4 What a real product would need that this does not have

I want to end this part honestly, because a competition device and a product are different things and it would be easy to let the reader forget that.

Secure firmware update.

The hardest remaining problem. You need a signature over the image, a bootloader that verifies it before jumping, anti-rollback on firmware version so old vulnerable images cannot be reinstalled, and a recovery path for a failed update that does not itself become the bypass. This is easily as much design work as everything in this document, and it is the single most attacked surface on shipped embedded devices.

Key rotation and a real key management story.

Right now the root is fixed for the life of the device. A product needs a defined lifetime, a rotation procedure, and an answer to what happens when the provisioning authority's key is compromised.

Certification.

FIPS 140-3 or Common Criteria if the deployment demands it. Both would require documented evidence for most of what this document asserts, and both would reject the entropy design in Chapter 9 as it stands.

Manufacturing security.

Who has access to the provisioning machine, how are root keys backed up, what stops a malicious operator from provisioning extra devices, and how do you detect it if they do.

Field diagnostics that do not leak.

Chapter 15 removed all error detail for good reasons. A product still needs to be supportable, which means an authenticated diagnostic mode, which means another privileged path to design carefully.

The point

The gap between "secure against a competition adversary" and "secure as a product" is mostly update, key management, and manufacturing. Those three are less intellectually interesting than side-channel work, and they are where real deployments actually fail.

PART X

The Decision Register

Every Alternative, In One Place

25.1 Why this chapter exists separately

Each technical chapter has its own alternatives box, which is useful while you are reading that chapter and useless when somebody asks you a question three months later. This chapter collects all of them into one table so that the whole set of trade-offs can be reviewed at once, and so that a person evaluating this design can see the shape of my judgement rather than one instance of it.

I have added a column that the individual boxes do not have: **what would change my mind**. A design decision without a stated reversal condition is a preference, not a decision.

25.2 The register

#	Chose	Over	Reason, and what would change my mind
D-01	Symmetric only	Public key	<i>Reason:</i> device and authority already share a secret from manufacture, so asymmetric solves a problem I do not have, at a cost of kilobytes and constant-time bignum risk. <i>Reversal:</i> an offline authority, a fleet needing forward secrecy against authority compromise, or a part with an ECC accelerator.
D-02	AES-256	AES-128	<i>Reason:</i> four extra rounds on a hardware engine is free; post-quantum margin is not. <i>Reversal:</i> nothing realistic. Keep 256.
D-03	CBC + HMAC	GCM / CCM	<i>Reason:</i> the engine does CBC natively, keeping chaining out of the CPU; GHASH in software on a core with no carryless multiply is slow and awkward to make constant time; GCM's nonce-reuse failure is catastrophic and my entropy is weak. <i>Reversal:</i> a GCM-capable engine plus a real TRNG. Then use GCM.
D-04	Encrypt-then-MAC	MAC-then-encrypt	<i>Reason:</i> only ordering where forged ciphertext never reaches the unpadding code. <i>Reversal:</i> none. This is not a preference.
D-05	Hardware AES	Bitsliced software AES	<i>Reason:</i> side-channel resistance, not speed; and my verification effort is better spent elsewhere than on a hand-rolled cipher. <i>Reversal:</i> no hardware engine on the target. Then bitslice, and budget a week for NIST vectors.
D-06	Three-tier HKDF	Flat stored keys	<i>Reason:</i> one stored secret instead of many, free revocation, compartmentalisation, and fewer traces per key against DPA. <i>Reversal:</i> none for the flat scheme.

#	Chose	Over	Reason, and what would change my mind
D-07	Three tiers	Four or more	<i>Reason:</i> the point where extra complexity stopped buying security for this threat model. <i>Reversal:</i> many concurrent users needing isolation from each other, or per-message forward secrecy requirements.
D-08	Entropy pool	Trusting one source	<i>Reason:</i> no dependable TRNG on the part; pooling and persisting is the honest fallback. <i>Reversal:</i> a certified TRNG with SP 800-90B health tests. Use it, still stirred.
D-09	Counter-derived IVs	Random IVs	<i>Reason:</i> removes a dependency on the weakest component in the system. <i>Reversal:</i> a strong TRNG makes random IVs equally safe and slightly simpler.
D-10	Append-only log	Read-modify-write struct	<i>Reason:</i> fixes write-in-place, atomicity, wear, and integrity in one structure. <i>Reversal:</i> an FRAM or EEPROM part with byte writes; then a plain struct with a MAC is fine.
D-11	Hand-written log	LittleFS or similar	<i>Reason:</i> two hundred lines I can read entirely, for a few hundred bytes of state. <i>Reversal:</i> state growing past a few kilobytes, or needing real file semantics.
D-12	MAC binds flash offset	MAC over contents only	<i>Reason:</i> closes the record-relocation attack. <i>Reversal:</i> none.
D-13	Increment before check	Increment after check	<i>Reason:</i> removes the power-cut window that gives an attacker free guesses. <i>Reversal:</i> none. This is the most important ordering in the document.
D-14	Growing backoff	Permanent lockout	<i>Reason:</i> same practical protection without converting authentication into a denial-of-service weapon. <i>Reversal:</i> a deployment where bricking on abuse is genuinely preferred, e.g. a payment card.
D-15	Device-bound PIN verifier	Plain salted hash	<i>Reason:</i> a verifier lifted from flash is useless without the device root, which kills offline cracking. <i>Reversal:</i> none.
D-16	Capability bitmask	Role table	<i>Reason:</i> no indirection, so no index to push out of range; nine commands do not need roles. <i>Reversal:</i> a policy space large enough that a bitmask stops fitting.
D-17	Fail-closed default	Deny-list	<i>Reason:</i> unknown states must be safe states, and a deny-list grants anything nobody thought to forbid. <i>Reversal:</i> none.
D-18	Redundant checks with sentinels	Single boolean	<i>Reason:</i> a glitched boolean is a coin flip; high-Hamming-distance sentinels fail closed under a single bit flip. <i>Reversal:</i> hardware fault detection in silicon.
D-19	One generic error	Detailed error codes	<i>Reason:</i> five useful error codes are five oracles handed to the attacker. <i>Reversal:</i> none for release builds. Debug builds get detail on a separate channel.

#	Chose	Over	Reason, and what would change my mind
D-20	Response timing floor	Best-effort timing	<i>Reason:</i> equalises paths that are naturally unequal, cheaply, on a device with no throughput requirement. <i>Reversal:</i> a throughput requirement. Then equalise properly rather than by padding.
D-21	Big-endian wire	Native-endian struct copy	<i>Reason:</i> forces explicit conversion code you can review, and avoids alignment faults on a core that does not permit unaligned access. <i>Reversal:</i> none.
D-22	Fixed header + one length	TLV encoding	<i>Reason:</i> every TLV parser I have reviewed had a length bug; flexibility here is the flexibility I do not want. <i>Reversal:</i> a genuine need for extensible commands, and then only with a formally specified grammar.
D-23	Linear dispatch scan	Array indexed by opcode	<i>Reason:</i> a scan cannot be pushed out of bounds by an unexpected opcode. <i>Reversal:</i> enough commands that the scan cost matters, which for this device is never.
D-24	Static allocation only	Heap	<i>Reason:</i> no allocation failure paths, no fragmentation-driven input-dependent behaviour, no freed blocks holding secrets. <i>Reversal:</i> none on a device this size.
D-25	Fixed-size parsed output	Pointers into the RX buffer	<i>Reason:</i> costs a kilobyte of SRAM, buys the guarantee that nothing downstream references a buffer about to be overwritten. <i>Reversal:</i> severe SRAM pressure, and then only with a documented lifetime discipline.
D-26	Guard band	MPU stack guard	<i>Reason:</i> the MPU was not available. The guard band is the weaker option and I want that on record. <i>Reversal:</i> any part with a usable MPU. Spend a region on this immediately.
D-27	No field update	Signed update path	<i>Reason:</i> a device that cannot be updated cannot be maliciously updated; the competition did not require it. <i>Reversal:</i> any real product. Then design it properly, and expect it to be as much work as this whole document.
D-28	No masking	Masked AES	<i>Reason:</i> masking needs fresh randomness per operation, and my randomness is the weakest link; a partially masked implementation provides no protection while looking like it does. <i>Reversal:</i> a real TRNG plus a lab to evaluate the result. For production against a funded attacker, you need this.
D-29	Rust as second impl	C only	<i>Reason:</i> the three things this design most needs a human to remember are three things the compiler can remember instead. <i>Reversal:</i> a team that cannot adopt it; then put the parser alone in Rust and link it.

#	Chose	Over	Reason, and what would change my mind
D-30	Accept A-4 exposure	Claim tamper resistance	<i>Reason:</i> the part has no mesh, no shield, no zeroisation. Claiming otherwise costs more credibility than the claim would buy. <i>Reversal:</i> a secure element or a part with tamper features.

25.3 The four decisions I would revisit first

If somebody handed me another month, this is the order I would work in.

1. **D-28, masking.** It is the largest real gap against a capable physical attacker, and everything else on this list is smaller. It depends on fixing D-08 first, which is why it is one item rather than two.
2. **D-08, entropy.** The chapter I am least happy with. An external component with a TRNG and a monotonic counter would fix this chapter and half of Chapter 11 for about a dollar, and on a platform I chose that is what I would do.
3. **The authentication restructure from Chapter 17's alternatives box.** Making the PIN an input to key derivation rather than a gate before it removes the branch that fault injection targets. There is no flag to glitch if the key is either derivable or not. This is the most elegant available improvement and I did not have time to do it properly.
4. **D-07, hierarchy depth** — which is also the open item in Appendix F. Before I discuss this design in an interview I need to read my own derivation code and confirm what shipped.

The point

Thirty decisions, and the honest summary is that about four of them are places where I traded real security for available time or available hardware, and the rest are places where I think the choice is correct on the merits. Being able to tell those two categories apart, out loud, is most of what a design review is for.

APPENDIX A

Requirement Traceability Matrix

Every requirement from Chapter 3, the control that satisfies it, and the test that demonstrates it. A requirement with no test is an aspiration. A control with no requirement is code nobody asked for.

Req	Control	Where	Test
R-01	No key output path exists in any handler	Ch. 8, 16, 19	Command fuzzing with key-pattern scanner on all output
R-02	Only the root is stored; all else derived	Ch. 8	Flash image inspection after a full operational cycle
R-03	<code>secure_zero</code> on every exit; <code>.secrets</code> section	Ch. 18, 19, 20	RAM inspection under debugger, debug build
R-04	Type-state session; <code>UNAUTH</code> on every reset	Ch. 14, 20	Every privileged command attempted from cold boot
R-05	<code>permission_allows</code> denies by default; redundant check	Ch. 13	Fault injection into policy record; corruption sweep
R-06	Increment-before-check with verified persistence	Ch. 11, 12	Power-cut sweep across the authenticate window
R-07	Growing backoff from persistent count	Ch. 12	Measured wall-clock delay across a failure sequence
R-08	Record MAC binds flash offset	Ch. 10	Slot-swap harness with a debugger
R-09	Highest- <code>seq</code> selection; no decrement path	Ch. 10, 11	Rollback attempted via every command and via flash rewrite
R-10	Session key binds device, nonce, generation; counter check	Ch. 11, 15	Cross-device and same-device replay harness
R-11	Single generic error; response timing floor	Ch. 15	Differential timing and response comparison per failure cause
R-12	Total parser; static buffers; no pointer arithmetic	Ch. 16, 19, 20	Coverage-guided fuzzing, ASan + UBSan, 50M+ executions
R-13	MAC written last; two-sector compaction	Ch. 10	Power-cut sweep across every flash write
R-14	Backoff never permanent; bounded loops; panic resets	Ch. 12, 16, 20	Long-running adversarial command sequence
R-15	<code>ct_eq</code> ; no secret-dependent branches	Ch. 12, 18	Statistical timing analysis, $ t < 4.5$
R-16	No secret-indexed memory access	Ch. 18	Source audit plus disassembly review

APPENDIX B

Test Vectors

B.1 Where to get them

Use published vectors only. Vectors you generated with the code under test verify that your code agrees with itself, which is not a property anybody cares about.

Primitive	Authoritative source
AES-256-CBC	NIST SP 800-38A, Appendix F.2. Also the CAVP AESAVS files.
SHA-256	FIPS 180-4 examples; NIST CAVP SHAVS byte-oriented vectors.
HMAC-SHA-256	RFC 4231, test cases 1 through 7.
HKDF-SHA-256	RFC 5869, Appendix A, test cases 1 through 3.

B.2 The minimum set to embed on-target

Full vector suites belong in host tests. On the device, at boot, run one vector per primitive: enough to detect a broken accelerator, small enough to fit.

```
1  /* AES-256-CBC, NIST SP 800-38A F.2.5 / F.2.6, first block only. */
2  extern const uint8_t KAT_AES_KEY[32]; /* 603deb10...0914dff4 */
3  extern const uint8_t KAT_AES_IV[16]; /* 000102...0e0f */
4  extern const uint8_t KAT_AES_PT[16]; /* 6bcb1bee2...7393172a */
5  extern const uint8_t KAT_AES_CT[16]; /* f58c4c04...5f7bfbdb */
6
7  /* SHA-256 of "abc", FIPS 180-4. */
8  static const uint8_t KAT_SHA_IN[3] = { 'a', 'b', 'c' };
9  static const uint8_t KAT_SHA_OUT[32] = {
10     0xba,0x78,0x16,0xbf,0x8f,0x01,0xcf,0xea, 0x41,0x41,0x40,0xde,0x5d,0xae,0x22,0x23,
11     0xb0,0x03,0x61,0xa3,0x96,0x17,0x7a,0x9c, 0xb4,0x10,0xff,0x61,0xf2,0x00,0x15,0xad
12 };
13
14 /* HMAC-SHA-256, RFC 4231 test case 2: key "Jefe", data "what do ya want
15  * for nothing?". Chosen because the key is shorter than the block,
16  * which exercises the zero-padding path where implementations go wrong. */
17 static const uint8_t KAT_HMAC_KEY[4] = { 'J','e','f','e' };
18 static const uint8_t KAT_HMAC_MSG[28] = "what do ya want for nothing?";
19 static const uint8_t KAT_HMAC_OUT[32] = {
20     0x5b,0xdc,0xc1,0x46,0xbf,0x60,0x75,0x4e, 0x6a,0x04,0x24,0x26,0x08,0x95,0x75,0xc7,
21     0x5a,0x00,0x3f,0x08,0x9d,0x27,0x39,0x83, 0x9d,0xec,0x58,0xb9,0x64,0xec,0x38,0x43
22 };
23
24 bool selftest_all(void)
25 {
26     /* Order matters: SHA underpins HMAC, HMAC underpins HKDF. Test
27     * bottom-up so a failure points at the lowest broken layer. */
28     if (!selftest_sha256()) { return false; }
29     if (!selftest_hmac()) { return false; }
30     if (!selftest_hkdf()) { return false; }
31     if (!selftest_aes()) { return false; }
32     return true;
33 }
```

Listing B.1: The on-target vector set. Roughly 200 bytes of flash, and it closes the silent-failure class from Chapter 7.

B.3 Vectors you must generate yourself

Nobody publishes vectors for your key hierarchy, your record format, or your frame format. Generate these with the Python reference model from Chapter 21, check them into the repository as data files, and treat any change to them as a protocol-breaking change requiring review.

Vector family	What it pins down
Tier-1 derivation	Known root and device id produce a known <code>PRK_dev</code>
Tier-2 derivation	Each label and context produces a known operational key
Record MAC	A known record at a known offset produces a known tag
Frame MAC	A known command under a known session key produces a known tag
PIN verifier	A known PIN, salt, and user id produce a known verifier
Padding	Every input length from 1 to 32 produces the expected padded output

Memory Map and Constants

C.1 Flash layout

Region	Access	Contents
Vector table	Read-only	Initial SP, reset handler, exception handlers
.text	Read-only	Firmware code
.rodata	Read-only	Constants, KAT vectors, the <code>const</code> dispatch table
Provisioning	Read-only after provisioning	ROOT_KEY (32 B), DEVICE_ID (16 B), PIN records
Log sector A	Read/erase	Append-only state records, Chapter 10
Log sector B	Read/erase	Alternate sector for compaction

C.2 SRAM layout

Region	Size	Contents
.data	small	Initialised globals, copied from flash at reset
.bss	small	Zeroed globals
.secrets	128 B	All key material, in one contiguous region so the scrub is one loop and the linker map shows it at a glance
Frame buffers	2 × 1066 B	<code>g_rx_frame</code> , <code>g_tx_frame</code>
Scratch	1024 B	<code>g_scratch</code> , used for padding and intermediate crypto
Guard band	64 B	Painted 0xDEADBEEF, checked each loop, Chapter 5
Stack	remainder	Grows down toward the guard band

C.3 Linker script fragment

```

1 MEMORY {
2   FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 256K
3   RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 32K
4 }
5
6 SECTIONS {
7   /* All secrets in one place. The scrub routine walks from
8    * __secrets_start to __secrets_end, so a secret declared outside this
9    * section is a secret that never gets erased. tools/check_secrets_
10   * section.py verifies at build time that nothing escaped. */
11   .secrets (NOLOAD) : ALIGN(4) {
12     __secrets_start = .;
13     KEEP(*(.secrets))
14     __secrets_end = .;
15   } > RAM
16

```

```
17  /* Guard band immediately below the stack. Painted at reset,  
18   * checked every loop iteration. Chapter 5 is honest about the fact  
19   * that this detects overflow rather than preventing it. */  
20  .stack_guard (NOLOAD) : ALIGN(4) {  
21    _sstack_guard = .;  
22    . = . + 64;  
23    _estack_guard = .;  
24  } > RAM  
25  
26  _estack = ORIGIN(RAM) + LENGTH(RAM);  
27  }  
28  
29  ASSERT(__secrets_end - __secrets_start <= 256,  
30    "secrets section grew unexpectedly; audit what was added");
```

Listing C.1: The parts of the linker script that carry security meaning.

Glossary

AEAD

Authenticated encryption with associated data. A construction giving confidentiality and integrity in one primitive. GCM and CCM are AEADs; CBC is not, which is why Chapter 6 wraps it in HMAC.

Bare metal

Firmware running with no operating system. Chapter 5.

Constant time

Execution whose duration and memory access pattern depend only on public values. Chapter 18.

DPA

Differential power analysis. Statistical recovery of a key from many power traces. Chapter 17.

Encrypt-then-MAC

Authenticating the ciphertext rather than the plaintext, so that forged input is rejected before decryption. Chapter 6.

Fail-closed

Denying by default, so that unexpected conditions produce refusal rather than permission. Chapter 13.

Fault injection

Deliberately pushing the chip outside its operating envelope to make an instruction produce the wrong result. Chapter 17.

HKDF

The extract-and-expand key derivation function of RFC 5869. Chapter 8.

HMAC

Keyed hash message authentication code. Immune to the length-extension weakness of a bare hash. Chapter 6.

HSM

Hardware security module. A device holding secrets that the host it serves cannot read. Chapter 1.

IKM

Input keying material. The possibly-non-uniform secret fed to HKDF-Extract.

KAT

Known-answer test. Running a primitive against a published vector to confirm it computes what it claims. Chapter 7.

Known-plaintext, chosen-plaintext

Attack models describing what an adversary may request. Both are assumed available to A-0 here.

Monotonic counter

A counter that can only increase, through any interface, across power loss. Chapter 11.

Nonce

A value used once under a given key. Reuse is catastrophic for some modes and merely bad for others.

PRK

Pseudorandom key. The uniform output of HKDF-Extract.

Rollback

Restoring persistent state to an earlier value, usually to undo a counter increment. Attacker goal G-4.

Side channel

Information leaked by the physical execution of correct code. Chapter 17.

SPA

Simple power analysis. Reading operations directly off a single power trace.

Type-state

Encoding a runtime state in the type system so invalid operations fail to compile. Chapter 20.

Zeroisation

Erasing secrets from memory so a later dump finds nothing. Chapter 18.

Review Checklists

E.1 Reviewing a change to the parser

-
- ☐ Every length is checked against both the protocol maximum and the received byte count
 - ☐ No arithmetic on a length can overflow before the check that uses it
 - ☐ No pointer is advanced by an unvalidated value
 - ☐ Every loop bound is a constant or a checked value
 - ☐ The output structure is zeroed before any field is written
 - ☐ The MAC is verified before any payload byte is copied out
 - ☐ Every rejection returns the same generic failure
 - ☐ The change is covered by a new fuzzing seed
 - ☐ The fuzzer has been re-run for at least an hour on the change
-

E.2 Reviewing a change that touches key material

-
- ☐ The key is declared in the `.secrets` section
 - ☐ Every exit path, including error paths, scrubs it
 - ☐ `aes256_clear` is called if the key reached the engine
 - ☐ No copy of the key was made that is not also scrubbed
 - ☐ The derivation context is unique to this purpose
 - ☐ The generation counter is bound into the context
 - ☐ Nothing derived from the key can be returned to the host
 - ☐ The linker map still shows the secrets section within its limit
-

E.3 Reviewing a change that touches persistent state

-
- ☐ The record is appended, never modified in place
 - ☐ The MAC covers the record's flash offset
 - ☐ The MAC field is written last, or the whole record atomically
 - ☐ The write is read back and verified
 - ☐ The value cannot decrease through any interface
 - ☐ Power loss at any point leaves either fully old or fully new state
 - ☐ The power-cut sweep has been re-run against the change

-
- ☐ Sector wear has been reconsidered if the write frequency changed
-

E.4 Reviewing a change that adds a command

-
- ☐ A row was added to `CMD_TABLE` with an explicit required capability
 - ☐ `needs_auth` is set correctly, and defaults to true if in doubt
 - ☐ The handler validates every payload field before use
 - ☐ The handler returns the generic error on every failure
 - ☐ The handler scrubs any key material on every exit
 - ☐ The response cannot exceed the transmit buffer
 - ☐ The command's timing has been measured against the response floor
 - ☐ The threat model has been re-read to check the command does not open a new goal
 - ☐ The reference model in Python implements the same command identically
-

Open Items

F.1 Things I need to confirm against my own repository

This appendix is short and it exists because I would rather write down what I do not currently know than let a reader assume I know it.

F-1. Key hierarchy depth. (Highest priority.)

This document describes a three-tier hierarchy: root, device key, operational keys. My resume, portfolio, and outreach all say three tiers. I have a note to myself questioning whether the shipped firmware added intermediate derivation steps, which would make it deeper. **Action: read the derivation code in the repository and settle this before discussing the project in any technical interview.** The architecture in Chapter 8 is what I would build again either way; the open question is purely what the implementation did.

F-2. Exact AESADV register sequence.

Chapter 7 describes the engine's contract rather than its registers, deliberately, so that the document does not rot. If you are implementing on this exact part, the register sequence must come from the current TI technical reference manual, not from memory and not from this document.

F-3. Measured response floor.

Chapter 15 sets the floor at 25 ms and says to measure it. I have not recorded a measurement in this document. **Action: measure the slowest legitimate path on a cold board, including PIN stretching, and set the constant from data rather than from a guess.**

F-4. Flash program granularity on this part.

Chapter 10 says to write the 64-byte record atomically where the part allows, and to write the MAC last otherwise. Which of those applies here needs confirming from the datasheet.

F-5. Stack depth number.

Chapter 19 gives the method and the limit of 4096 bytes. I have not recorded the actual measured worst case. It should be in the repository's CI output and it should be quoted here.

F-6. Whether the debug lock was verified per unit.

Chapter 24 insists on verifying that the lock took. For competition devices this may not have been done systematically. Worth confirming before claiming it as a control.

F.2 Things I would add with more time

1. Masking for the AES data path, contingent on fixing the entropy source first (D-28, D-08).
2. Restructuring authentication so the PIN feeds key derivation rather than gating it, removing the branch that fault injection targets (Chapter 17).
3. A session-structure checksum verified before every use, so a glitched `authenticated` flag is caught.
4. Formal verification of the constant-time property on `ct_eq` and the padding routines.
5. A full secure firmware update design, which is the largest single gap between this and a product (D-27).

End of document. Version 1.0, August 2026.