Cyber Apocalypse 2026 — Reversing — The Cinder Engine

Category: Reversing
Provided file: `cinder`

FLAG
HTB{c1nd3r_3ng1n3_unw0und}

---

1. The premise

> *"He kept the opcode map in his head and nowhere else. ... Work out its opcode map from how it moves. Pull the firmware apart and find the cipher hiding in its tables."*

The flavour text is an unusually precise spec for the work ahead, and it maps onto three concrete stages:

1. **Work out its opcode map from how it moves** — the binary is an interpreter for an invented instruction set. No symbols, no opcode names, no documentation. The semantics have to be read out of the dispatch code.
2. **Pull the firmware apart and find the cipher hiding in its tables** — the bytecode the interpreter runs lives in `.rodata`, and so do an S-box, nine round keys, a target ciphertext, and a flag mask.
3. **Feed it something worth saying** — the firmware is a keyed block cipher plus an equality check. Invert the cipher, recover the one 32-byte "vow" it accepts, and the engine hands over the flag.

The last stage is the nice bit of design in this challenge: the flag is not a constant sitting in the binary. It is XOR-masked with the correct input, so you cannot print it without first having solved the cipher. There is nothing to grep for.

2. First contact

```
$ file cinder
cinder: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV),
        dynamically linked, interpreter /lib/ld-linux-aarch64.so.1,
        BuildID[sha1]=b3296820..., for GNU/Linux 3.7.0, stripped
```

aarch64, stripped, 14 KB. `strings` is almost silent — the only interesting output in the whole file is:

```
bad opcode
```

That single string is the tell. A binary whose entire error vocabulary is *"bad opcode"* is a virtual machine, and it means the real program is data, not code.

The section table backs this up:

```
$ objdump -h cinder
 13 .text     000005d4  0000000000000880  TEXT
 15 .rodata   00001336  0000000000000e68  DATA
 23 .bss      00010048  0000000000013010  BSS
```

1492 bytes of code against 4918 bytes of read-only data and a 64 KB `.bss`. The code/data ratio is inverted from a normal program. `.text` is the interpreter; `.rodata` is the firmware; `.bss` is the VM's RAM.

`objdump` on macOS handles the aarch64 triple natively, so no cross-toolchain was needed:

```
$ objdump -d cinder > cinder.asm
$ wc -l cinder.asm
458 cinder.asm
```

458 lines total, and the whole VM lives in one function at `0x880`. That is small enough to read line by line, which is exactly what deriving the opcode map requires.

3. Reading the interpreter

3.1 Machine state

The prologue establishes the VM's world:

```
 880: stp   x29, x30, [sp, #-0x50]!
 884: mov   x13, #0x1010
 89c: sub   sp, sp, x13              ; 0x1000-byte input buffer + canary
 8ac: add   x19, sp, #0x8            ; x19 = input buffer
 8b8: ldr   x21, [x21, #0xfd8]       ; x21 = &stdin
...
 8e0: ldr   x0, [x21]
 8e4: bl    getc
 8e8: cmn   w0, #0x1                 ; EOF?
 8d0: add   w22, w22, #0x1           ; w22 = bytes read
 8d4: strb  w0, [x20], #0x1
 8d8: cmp   w22, #0x1, lsl #12       ; cap at 0x1000
```

So the program slurps up to 4096 bytes of stdin into a stack buffer and records the length in `w22`. Then:

```
 8f0: adrp  x23, 0x13000
 8f8: add   x23, x23, #0x18          ; x23 = 0x13018  -> register file
 8f4: adrp  x21, 0x0
 8fc: add   x21, x21, #0xe80         ; x21 = 0xe80    -> bytecode base
 91c: mov   w25, #0x131e             ; = 4894         -> bytecode length
```

Register accesses are all of the form `ldr w0, [x23, x1, lsl #2]` with `x1` masked to 4 bits, which pins down the register file precisely:

Component	Address	Size	Notes
Register file	`0x13018`	16 × `u32`	4-bit register operands
VM RAM	`0x13058`	0x10000 bytes	byte-addressed, address masked `& 0xffff`
Bytecode / ROM	`0xe80`	4894 (`0x131e`) bytes	in `.rodata`, **also readable as data**

The last row matters. `.rodata` runs from `0xe68` to `0x219e`; the bytecode base is `0xe80` and the length constant `0x131e` in `w25` covers `.rodata` all the way to its final byte. The firmware's code and its tables occupy one address space, and one opcode can read from it — this is the "cipher hiding in its tables".

The 64 KB of VM RAM plus the 64-byte register file plus 8 bytes of housekeeping is exactly `0x10048`, the size of `.bss`. The layout is fully accounted for.

3.2 Instruction encoding

The decode block at `0x920` is the crux. It is written in a deliberately confusing way — the four fetched bytes arrive in registers `w3, w1, w2, w0` (not in address order) and are recombined by hand:

```
920: lsl   w2, w2, #16
924: orr   w0, w3, w0, lsl #24
928: orr   w1, w2, w1, lsl #8
92c: add   w20, w20, #4             ; pc += 4  (during decode)
930: orr   w2, w1, w0               ; w2 = full 32-bit instruction word
934: lsr   w0, w0, #24              ; w0 = opcode
938: orr   w3, w2, #0xffff0000
93c: and   w5, w2, #0xffff          ; w5 = imm16
940: tst   x2, #0x8000
944: and   w6, w2, #0xf             ; w6 = rB
948: csel  w4, w3, w5, ne           ; w4 = sign-extended imm16
94c: ubfx  x3, x1, #20, #4          ; w3 = rD
950: ubfx  x1, x1, #16, #4          ; w1 = rA
```

Untangling the `orr` chain: the word is simply the four bytes in little-endian order, `insn = b0 | b1<<8 | b2<<16 | b3<<24`. From there:

Field	Extraction	Meaning
`opcode`	`insn >> 24` (= `b3`)	**top** byte, not the bottom
`rD`	`(insn >> 20) & 0xf`	destination register
`rA`	`(insn >> 16) & 0xf`	first source register
`rB`	`insn & 0xf`	second source register (aliases the immediate!)
`imm16`	`insn & 0xffff`	zero-extended immediate
`simm16`	sign-extended `imm16`	branch displacement
`shift`	`imm16 & 0x1f`	shift amount (ARM masks the shift register)

Two details in here are easy to get wrong and both matter:

- **The opcode is the most-significant byte.** Reading the word as little-endian and taking the low byte gives garbage.
- `rB` **and the immediate overlap.** ALU instructions use `insn & 0xf` as the second source; immediate instructions use `insn & 0xffff`. The same bits serve both roles depending on the opcode, so a decoder that always prints both fields prints nonsense for half the instructions.

`pc += 4` happens *inside* the decode block, so branch targets are relative to the instruction *after* the branch, not to the branch itself:

```
aec: cbz   w26, 0x9a8              ; JZ: not taken -> fall through
af0: add   w20, w20, w4            ; taken -> pc += simm16
```

There is also a startup quirk worth pinning down, because it shifts every address in the firmware. At `0x900` the interpreter pre-loads the decode registers by hand before falling into the decode block:

```
900: mov   w0, #0x3a               ; synthetic opcode
904: mov   w2, #0x0
908: mov   w1, #0x0
90c: mov   w3, #0x0
914: mov   w20, #0x0               ; pc = 0
...falls through into the decoder at 0x920
```

The first pass through the decoder therefore executes a *synthetic* instruction — opcode `0x3a`, `rD = 0`, `imm = 0` — and increments `pc` to 4. Only then does the fetch at `0x9a8` read real bytes from the ROM. The net effect: **the instruction word at ROM offset 0 is never executed**, execution truly begins at offset 4, and the synthetic instruction is a harmless `r0 = 0` on an already-zeroed register file. Getting this wrong desynchronises the entire disassembly by one word.

3.3 Deriving the opcode map

Dispatch is a hand-rolled binary search over `cmp`/`b.eq`/`b.ls` — the compiler turned a `switch` into a decision tree:

```
 954: cmp   w0, #0x7c
 958: b.eq  0xb80
 95c: b.hi  0xa30
 960: cmp   w0, #0x2c
 964: b.eq  0xcfc
 968: b.ls  0x9fc
 ...
 cb4: adrp  x0, 0x0                 ; default case
 cc0: add   x0, x0, #0xe70          ; "bad opcode\n"
 cd0: bl    fwrite
```

Every leaf is a handler, and each handler is three to six instructions of completely transparent ARM. Two examples:

```
; opcode 0x7c
 b80: ldr   w0, [x23, x1, lsl #2]    ; regs[rA]
 b88: ldr   w1, [x23, x6, lsl #2]    ; regs[rB]
 b90: adds  w0, w0, w1               ; add, setting flags
 b94: str   w0, [x23, x3, lsl #2]    ; regs[rD] = result
 b98: cset  w26, eq                  ; Z = (result == 0)
```

`ADD rD, rA, rB`.

```
; opcode 0xc6
 c68: ldr   w0, [x23, x1, lsl #2]    ; regs[rA]
 c74: add   w5, w5, w0               ; imm16 + regs[rA]
 c78: udiv  w0, w5, w25              ; w25 = 4894
 c7c: msub  w0, w0, w25, w5          ; ... % 4894
 c80: ldrb  w0, [x21, x0]            ; read from the BYTECODE region
 c84: str   w0, [x23, x3, lsl #2]
```

A load from the firmware's own instruction stream, modulo the ROM length. This is the table-reading opcode, and it is how the cipher's constants get fetched.

Walking every leaf of the tree gives the complete map. `w26` is the machine's single flag: a zero flag, written by the arithmetic/logic ops and by `CMP`.

Opcode	Handler	Mnemonic	Semantics	Sets Z
`0x00`	`0xb40`	`HALT`	return from `main` (exit 0)	—
`0x11`	`0xa84`	`MOV`	`rD = rA`	no
`0x29`	`0xc98`	`ROL`	`rD = rA <<< (imm & 31)`	no
`0x2a`	`0xc08`	`ROR`	`rD = rA >>> (imm & 31)`	no
`0x2b`	`0xa18`	`SHL`	`rD = rA << (imm & 31)`	no
`0x2c`	`0xcfc`	`SHR`	`rD = rA >> (imm & 31)` (logical)	no
`0x3a`	`0xc20`	`LDI`	`rD = imm16`	no
`0x52`	`0x9d8`	`XOR`	`rD = rA ^ rB`	yes
`0x53`	`0xcdc`	`AND`	`rD = rA & rB`	yes
`0x54`	`0xc2c`	`OR`	`rD = rA | rB`	yes
`0x6b`	`0x988`	`MUL`	`rD = rA * rB`	yes
`0x7c`	`0xb80`	`ADD`	`rD = rA + rB`	yes
`0x7d`	`0xba0`	`SUB`	`rD = rA - rB`	yes
`0x80`	`0xb24`	`ADDI`	`rD = rA + imm16`	yes
`0x90`	`0xc50`	`CMP`	`Z = (rA == rB)`, no write	yes
`0xa0`	`0xaf0`	`JMP`	`pc += simm16`	no
`0xa1`	`0xaec`	`JZ`	`if Z: pc += simm16`	no
`0xa2`	`0xc8c`	`JNZ`	`if !Z: pc += simm16`	no
`0xc4`	`0xbc0`	`LDB`	`rD = mem[(rA + imm16) & 0xffff]`	no
`0xc5`	`0xaa8`	`STB`	`mem[(rA + imm16) & 0xffff] = rD & 0xff`	no
`0xc6`	`0xc68`	`LDROM`	`rD = rom[(rA + imm16) % 4894]`	no
`0xe0`	`0xbe8`	`GETC`	`rD = input[i++]`; on EOF `rD = 0, Z = 1`	yes
`0xe1`	`0xa58`	`PUTC`	`putc(rA & 0xff, stdout)`	no

Twenty-three opcodes out of 256 possible values, scattered across the byte range with no structure to guess from — hence the scribe and his private map. Note the small asymmetries the handlers reveal, which a careless reading would smooth over: `MOV` and the shifts leave `Z` untouched, `CMP` writes `Z` without writing a register, and `GETC` signals EOF through `Z` rather than through a sentinel value.

There is a satisfying confirmation that this reading is right: implementing this table and disassembling the ROM straight through hits its first invalid opcode at **exactly** offset `0x10c4`, and not one byte earlier. 4292 bytes decode

cleanly as instructions; everything past `0x10c4` is data. A wrong opcode map would produce `bad opcode` scattered throughout.

4. Disassembling the firmware

With `vmdis.py` (a ~90-line transcription of the table above) the firmware reads clearly. It is a single flat program with no subroutines.

4.1 Input intake

```
0000: LDI    r0 = 0x0000          <- never executed (synthetic insn covers offset 0)
0004: LDI    r8 = 0x0020          ; 32 = block size
0008: LDI    r4 = 0x0000
000c: GETC   r5 = getc()
0010: STB    mem[r4 + 0x0000] = r5     ; working state
0014: STB    mem[r4 + 0x0040] = r5     ; pristine copy
0018: ADDI   r4 = r4 + 1
001c: CMP    cmp r4, r8
0020: JNZ    0x000c
```

Exactly 32 bytes are consumed, and each byte is written **twice**: into the working state at `mem[0x00]` and into a second buffer at `mem[0x40]`. That second copy is untouched by everything that follows, and it is the mechanism that makes the flag unobtainable without the real answer. Hold that thought.

4.2 The round structure

```
0024: LDI    r7 = 0x11c4          ; round-key pointer into ROM
002c: ADD    r9 = r7 + r4
0030: LDROM  r1 = rom[r9]
0034: LDB    r5 = mem[r4]
0038: XOR    r5 = r5 ^ r1
003c: STB    mem[r4] = r5         ; whitening: state ^= K0
004c: ADDI   r7 = r7 + 0x0020     ; -> K1
0050: LDI    r6 = 0x0001          ; round counter

0054: LDI    r4 = 0x0000          ; ---- top of round loop ----
0058: LDB    r5 = mem[r4]
005c: LDROM  r5 = rom[(r5 + 0x10c4) % 4894]   ; S-box substitution
0060: STB    mem[r4] = r5
0064: ADDI   r4 = r4 + 1
0068: CMP    cmp r4, r8
006c: JNZ    0x0058

0070: ... 4000 bytes of unrolled XOR chains ...    ; linear layer

1018: LDI    r4 = 0x0000          ; copy mem[0x80..0x9f] -> mem[0x00..0x1f]
101c: LDB    r5 = mem[r4 + 0x0080]
1020: STB    mem[r4 + 0x0000] = r5
102c: JNZ    0x101c

1030: LDI    r4 = 0x0000          ; state ^= K[r6]
1034: ADD    r9 = r7 + r4
1038: LDROM  r1 = rom[r9]
103c: LDB    r5 = mem[r4]
1040: XOR    r5 = r5 ^ r1
1044: STB    mem[r4] = r5
1050: JNZ    0x1034

1054: ADDI   r7 = r7 + 0x0020     ; next round key
1058: ADDI   r6 = r6 + 1
105c: LDI    r12 = 0x0009
1060: CMP    cmp r6, r12
1064: JNZ    0x0054               ; ---- loop while r6 != 9 ----
```

A textbook substitution–permutation network:

```
state = input ^ K0
for r in 1..8:
    state = SBOX[state]        (bytewise)
    state = L(state)           (linear layer)
    state = state ^ K[r]
```

`r6` runs 1..8, so **8 rounds** and **9 round keys**. The keys sit consecutively at `0x11c4`, and `0x11c4 + 9*32 = 0x12e4` — which lands exactly on the next table. The arithmetic closes.

4.3 The verdict and the payoff

```
1068: LDI    r11 = 0x0000              ; mismatch accumulator
106c: LDI    r4 = 0x0000
1070: LDROM  r1 = rom[(r4 + 0x12e4) % 4894]   ; target ciphertext
1074: LDB    r5 = mem[r4]
1078: CMP    cmp r5, r1
107c: JZ     0x1084
1080: LDI    r11 = 0x0001              ; any byte differs -> sticky flag
1084: ADDI   r4 = r4 + 1
108c: JNZ    0x1070

1094: CMP    cmp r11, r12              ; r12 = 0
1098: JNZ    0x10c0                    ; mismatch -> straight to HALT

109c: LDI    r4 = 0x0000               ; ---- success path ----
10a0: LDI    r8 = 0x001a               ; 26 bytes
10a4: LDROM  r1 = rom[(r4 + 0x1304) % 4894]   ; flag mask
10a8: LDB    r5 = mem[r4 + 0x0040]     ; <-- the PRISTINE INPUT COPY
10ac: XOR    r5 = r5 ^ r1
10b0: PUTC   putc(r5)
10bc: JNZ    0x10a4
10c0: HALT
```

And there is the design. The flag is emitted as

```
flag[i] = rom[0x1304 + i] XOR original_input[i]      for i in 0..25
```

The mask at `0x1304` is 26 bytes and `0x1304 + 26 = 0x131e` — the exact end of the ROM. Nothing is wasted.

Because the flag is XORed against *the input the user supplied*, the mask on its own is worthless. Feed the engine the wrong vow and it prints nothing at all (the comparison gates the output); feed it a vow that somehow passed but differed, and it would print garbage. The only way to see the flag is to hold the genuine 32-byte preimage. The challenge cannot be shortcut by patching the comparison either — patching `JNZ 0x10c0` to a no-op just makes the engine print `mask XOR whatever_you_typed`, which is noise. **You have to actually break the cipher.** That is a considerably better construction than the usual "flip the branch and win".

Which is also the story the flavour text tells: the engine will only recite the promise to someone who already knows it.

5. The tables

Four tables live past `0x10c4`, and their boundaries are dictated by the code that reads them:

ROM offset	Size	Contents	Read by
`0x10c4`	256	S-box	`LDROM` at `0x005c`
`0x11c4`	9 × 32	round keys K0..K8	`LDROM` at `0x0030`, `0x1038`
`0x12e4`	32	target ciphertext	`LDROM` at `0x1070`
`0x1304`	26	flag mask	`LDROM` at `0x10a4`

S-box (`rom[0x10c4:0x11c4]`):

```
41 6f 9a c0 6b 9d e8 6c 70 07 88 18 bb 9f 0c 77
59 15 86 1a a1 c7 5a b0 10 06 20 5c 3c 47 0d 05
95 f4 91 4c 67 a4 d6 b7 fa be 7d 83 73 37 26 2b
f3 f7 fb 84 08 8b 0f 60 b8 17 5f 3b 35 94 c1 b6
9c 6e 51 16 7e aa 0e 21 d5 bf 50 ca 64 f2 b5 ac
33 53 ba 32 5d 1c 78 62 ea 3e eb 7f 8f a5 cc 24
df 49 63 ce 58 56 52 bd c5 b9 d4 8c e2 4d c2 04
e7 01 2d 3d bc e0 81 98 92 b2 2e da 34 79 39 f5
cb 4f d1 c9 2f 03 cd ff 43 ab 36 69 e3 76 8d a9
02 f0 12 ae 45 e9 5b ec 38 e4 31 22 1d 09 f6 7b
55 8a 66 29 d0 2a c6 ee 4b 97 a0 71 8e db 3a fd
74 ef 1f 5e 90 a2 f8 00 6a f9 fc 2c 7a 65 3f a7
0b 28 e6 1e 57 13 48 61 6d b1 85 de d3 54 dd 14
4e 0a 11 40 fe 82 4a a8 30 46 42 a3 a6 e5 25 c4
80 89 72 96 c3 e1 ed d8 27 75 93 9b c8 d2 cf 44
d7 7c ad dc b3 d9 87 68 19 99 b4 9e 1b f1 af 23
```

It is a genuine permutation of 0..255 — verified, not assumed:

```
assert sorted(sbox) == list(range(256))
```

so inverting it is a matter of turning the table around.

Round keys (`rom[0x11c4:0x12e4]`):

```
K0 = c9d132d3a677c1221e30964bbbbb272c2308c2906c2fd677b2ae89f2978b987b
K1 = 4dbe5e98b79eddde1e603288bbd460fcb2fe293454eed3c9520d7103111810fa
K2 = 2a487d0f48ff03c9843d0cf99048bdf61de6604e8ebfb8fb98df1ee5dc98ab32
K3 = 1d08132ea491d7fb6e0dc3e638d4f1d79155fc3ef3504a19e6f45dc86c562a26
K4 = 2f3102039327b64f6221152384227a3d2a44488e8c9061057ecbe15b4e76ef4d
K5 = 892d48f9c704a181f8c38b5d94777a87f76fd13eb8dce6b1026c5199d23d7602
K6 = 1b6b2687b16abbfac4ad468c935241afa33e8f934089bd601337e0bd841eb1df
K7 = f6c4fa036400a3d23c20aadf47db467b6a9604204518f70d686905855f7b5f09
K8 = ba40d6688af59b0832c1d5aaf0ecad7d68b9689d4777996e78e2cf826e1bc01c
```

Target ciphertext (`rom[0x12e4:0x1304]`):

```
35a1ee02573d33ce3b2537d49ed1c062b2d78d03b3db4e0c4ba630953e19b34b
```

Flag mask (`rom[0x1304:0x131e]`):

```
cbbbe8d7f8ab289fcd7ee9568ee5231eb327564ea22c687d2b26
```

6. The linear layer

The 4000 bytes between `0x0070` and `0x1018` are a fully unrolled diffusion layer. One output byte looks like this:

```
0070: LDB    r5 = mem[r0 + 0x0000]
0074: LDB    r1 = mem[r0 + 0x0001]
0078: XOR    r5 = r5 ^ r1
007c: LDB    r1 = mem[r0 + 0x0003]
0080: XOR    r5 = r5 ^ r1
...
00c4: STB    mem[r0 + 0x0080] = r5
```

That is `out[0] = in[0] ^ in[1] ^ in[3] ^ in[6] ^ in[7] ^ in[9] ^ in[16] ^ in[18] ^ in[19] ^ in[25] ^ in[28]`, repeated 32 times with different tap sets. Critically:

- Inputs come from `mem[0x00..0x1f]`, outputs go to `mem[0x80..0x9f]`, so there is **no aliasing** — every output sees the same input state.
- Every operation is a whole-byte XOR. There are no shifts, rotates or multiplies anywhere in the layer, so **bit positions never mix**. Bit *k* of the output depends only on bit *k* of the input.

That second point is what makes the whole cipher fall. The layer is a single 32×32 matrix over GF(2), applied independently to each of the eight bit-planes. Inverting it means inverting one small binary matrix — not eight, and certainly not a 256×256 one.

Rather than transcribe 1000 instructions by hand, the taps are recovered by walking the decoded instruction stream and accumulating a bitmask per output byte:

```
rows = [None] * 32
for pc in range(0x70, 0x1018, 4):
    op, imm, simm, rD, rA, rB, sh, insn = decode(pc)
    if op == 0xC4 and rA == 0:              # LDB
        if rD == 5:
            cur = 1 << imm                  # r5 <- first term: new output byte
        else:
            pending = imm                   # r1 <- an operand
    elif op == 0x52:                        # XOR r5 = r5 ^ r1
        cur ^= 1 << pending
    elif op == 0xC5 and rA == 0:            # STB -> commit
        rows[imm - 0x80] = cur
    else:
        raise AssertionError('unexpected insn at %04x' % pc)
```

The `raise` is the point of the loop, not an afterthought: the parse only succeeds if that 4000-byte span is *nothing but* this one pattern. It completes without firing, which proves the region is a pure linear layer with no hidden non-linear step tucked into the middle.

The recovered matrix, as 32-bit tap masks (bit *i* set ⇒ output *j* includes input byte *i*):

out	mask	input bytes
0	`0x120d02cb`	0 1 3 6 7 9 16 18 19 25 28
1	`0xb5940bc4`	2 6 7 8 9 11 18 20 23 24 26 28 29 31
2	`0x54e1b571`	0 4 5 6 8 10 12 13 15 16 21 22 23 26 28 30
3	`0x6acdb9d0`	4 6 7 8 11 12 13 15 16 18 19 22 23 25 27 29 30
4	`0xeda4f04d`	0 2 3 6 12 13 14 15 18 21 23 24 26 27 29 30 31
5	`0xa634e20b`	0 1 3 9 13 14 15 18 20 21 25 26 29 31
6	`0x88e9f4a1`	0 5 7 10 12 13 14 15 16 19 21 22 23 27 31
7	`0x5f1b8071`	0 4 5 6 15 16 17 19 20 24 25 26 27 28 30
8	`0xbc2a2a12`	1 4 9 11 13 17 19 21 26 27 28 29 31
9	`0x0a9a75f1`	0 4 5 6 7 8 10 12 13 14 17 19 20 23 25 27
10	`0xd85cd8e1`	0 5 6 7 11 12 14 15 18 19 20 22 27 28 30 31
11	`0x3b22b05b`	0 1 3 4 6 12 13 15 17 21 24 25 27 28 29
12	`0xeed7a35c`	2 3 4 6 8 9 13 15 16 17 18 20 22 23 25 26 27 29 30 31
13	`0x749ce4fb`	0 1 3 4 5 6 7 10 13 14 15 18 19 20 23 26 28 29 30
14	`0x09071135`	0 2 4 5 8 12 16 17 18 24 27
15	`0xadc88b8e`	1 2 3 7 8 9 11 15 19 22 23 24 26 27 29 31
16	`0x8f0ae929`	0 3 5 8 11 13 14 15 17 19 24 25 26 27 31
17	`0xebb209f0`	4 5 6 7 8 11 17 20 21 23 24 25 27 29 30 31
18	`0x8527a234`	2 4 5 9 13 15 16 17 18 21 24 26 31
19	`0xb2fe1754`	2 4 6 8 9 10 12 17 18 19 20 21 22 23 25 28 29 31
20	`0x2abafc6f`	0 1 2 3 5 6 10 11 12 13 14 15 17 19 20 21 23 25 27 29
21	`0x32309fd9`	0 3 4 6 7 8 9 10 11 12 15 20 21 25 28 29
22	`0xf459ade1`	0 5 6 7 8 10 11 13 15 16 19 20 22 26 28 29 30 31
23	`0xb4341f35`	0 2 4 5 8 9 10 11 12 18 20 21 26 28 29 31
24	`0x46bb7095`	0 2 4 7 12 13 14 16 17 19 20 21 23 25 26 30
25	`0xd88421fc`	2 3 4 5 6 7 8 13 18 23 27 28 30 31
26	`0x5da52afb`	0 1 3 4 5 6 7 9 11 13 16 18 21 23 24 26 27 28 30
27		0 3 4 6 7 9 12 18 19 20 22 23 25 26 29 30 31

out	mask	input bytes
	`0xe6dc12d9`	
28	`0x9109cbaa`	1 3 5 7 8 9 11 14 15 16 19 24 28 31
29	`0xd8685504`	2 8 10 12 14 19 21 22 27 28 30 31
30	`0xacb28525`	0 2 5 8 10 15 17 20 21 23 26 27 29 31
31	`0x8ed89e9e`	1 2 3 4 7 9 10 11 12 15 19 20 22 23 25 26 27 31

Row weights range from 11 to 20 — roughly half of 32, as you would want from a diffusion layer.

Inverting over GF(2) is Gaussian elimination on the rows augmented with an identity tag:

```
aug = [(rows[j], 1 << j) for j in range(32)]
piv = {}
for col in range(32):
    r = next(k for k in range(32)
             if k not in piv.values() and (aug[k][0] >> col) & 1)
    piv[col] = r
    for k in range(32):
        if k != r and (aug[k][0] >> col) & 1:
            aug[k] = (aug[k][0] ^ aug[r][0], aug[k][1] ^ aug[r][1])

inv_rows = [0] * 32
for col in range(32):
    row, tag = aug[piv[col]]
    assert row == (1 << col)      # reduced to the identity => matrix was invertible
    inv_rows[col] = tag
```

The matrix is non-singular, so `L` is a bijection and the round function inverts cleanly.

7. Inverting the engine

Every stage is now reversible, so the cipher runs backwards directly. No key recovery, no search, no differential work — the round keys were sitting in `.rodata` the whole time.

```
state = ciphertext
for r in range(8, 0, -1):
    state = [state[i] ^ keys[r][i] for i in range(32)]   # undo AddRoundKey
    state = Linv(state)                                  # undo linear layer
    state = [inv_sbox[b] for b in state]                 # undo S-box
vow = [state[i] ^ keys[0][i] for i in range(32)]         # undo whitening
```

Forward-checking the result closes the loop:

```
assert encrypt(vow) == ciphertext
```

The accepted vow — the 32 bytes the engine has been waiting for:

```
83efaaac9b9a46fbfe0cb665e082127080782320d51c1d134f5bcd157abaf50f
```

It is not printable text, which is worth noting: there was never a passphrase to guess. The engine judges an arbitrary 32-byte block, and the only route to it is the algebra.

XOR the first 26 bytes against the mask at `0x1304`:

```
mask :  cb bb e8 d7 f8 ab 28 9f cd 7e e9 56 8e e5 23 1e b3 27 56 4e a2 2c 68 7d 2b 26
vow  :  83 ef aa ac 9b 9a 46 fb fe 0c b6 65 e0 82 12 70 80 78 23 20 d5 1c 1d 13 4f 5b
XOR  :  48 54 42 7b 63 31 6e 64 33 72 5f 33 6e 67 31 6e 33 5f 75 6e 77 30 75 6e 64 7d
ASCII:  H  T  B  {  c  1  n  d  3  r  _  3  n  g  1  n  3  _  u  n  w  0  u  n  d  }
```

```
HTB{c1nd3r_3ng1n3_unw0und}
```

8. Verification

The target is aarch64 and this analysis was done on an arm64 Mac with no user-space ARM-Linux emulation available (no `qemu-aarch64`, no Docker/Podman/Colima), so the binary could not be executed directly. Rather than leave the result resting on the solver alone, I verified it the stronger way: by writing a **second, independent implementation** — `emulate.py`, a direct transcription of the aarch64 interpreter into Python, opcode by opcode, with no knowledge of the cipher at all. It fetches, decodes and dispatches exactly as the ARM code does, including the synthetic first instruction, the `pc += 4`-during-decode ordering, the `& 0xffff` address mask and the `% 4894` ROM wrap.

Running the unmodified firmware under it:

```
$ python3 emulate.py 83efaaac9b9a46fbfe0cb665e082127080782320d51c1d134f5bcd157abaf50f
[halted after 13812 instructions]
HTB{c1nd3r_3ng1n3_unw0und}

$ python3 emulate.py 83efaaac9b9a46fbfe0cb665e082127080782320d51c1d134f5bcd157abaf50e
[halted after 13660 instructions]
```

One byte changed in the last position and the engine goes silent — the comparison fails and the success path is skipped, 152 instructions short. The emulator halts cleanly on `HALT` in both cases and never reaches the `bad opcode` path, which independently re-confirms the opcode map: a single misread handler would have derailed a 13812-instruction trace long before the end.

This is a genuine end-to-end check of the firmware as shipped — the same bytes, the same control flow, the same tables — and it agrees with the algebraic solver. Two implementations built from different directions producing the same 26 bytes is what makes the answer trustworthy. To reproduce the confirmation on the real binary, `qemu-aarch64 ./cinder` with the vow on stdin will do it:

```
$ printf '\x83\xef\xaa\xac\x9b\x9aF\xfb\xfe\x0c\xb6e\xe0\x82\x12p\x80x# \xd5\x1c\x1d\x130[\xcd\x15z\xba\xf5\x01
HTB{c1nd3r_3ng1n3_unw0und}
```

9. Retrospective

The challenge is well constructed, and its difficulty is placed thoughtfully. Almost all of the work is in the first stage — recovering the opcode map — and the three genuine traps there are the ones the flavour text hints at: the opcode is the *high* byte of a little-endian word, `rB` shares bits with the immediate, and the instruction at offset 0 is never executed. Get any of those wrong and the firmware decodes into noise with no obvious clue as to why.

Once the ISA is right, the firmware is honest about itself. The round structure is legible, the tables announce their own boundaries through the offsets that read them, and the arithmetic checks out at every seam (`0x11c4 + 9*32 = 0x12e4`; `0x1304 + 26 = 0x131e`). The cipher then falls not to cryptanalysis but to bookkeeping: the round keys are in plain sight, the S-box is a permutation, and the diffusion layer's refusal to mix bit positions reduces it to one 32×32 GF(2) matrix inverse.

The detail worth stealing is the output stage. Saving a pristine copy of the input at `mem[0x40]` and emitting `mask XOR input` means the flag is *derived* from the answer rather than guarded by a branch. There is no constant to grep, no comparison to patch, no shortcut past the mathematics — and the mechanism costs the author about six instructions. A dead man's machine that will only recite the promise to someone who already knows it.

Appendix: files

File	Purpose
`vmdis.py`	VM disassembler — the opcode map, implemented
`solve.py`	Table extraction, GF(2) matrix inversion, cipher inversion
`emulate.py`	Independent VM emulator, transcribed from the aarch64 interpreter
`firmware.txt`	Full firmware disassembly (1223 lines)
`cinder.asm`	`objdump -d` of the binary (458 lines)

Flag: `HTB{c1nd3r_3ng1n3_unw0und}`