CyberApocalypse 2026 — Reversing — First Mark

Challenge: First Mark **Category:** Reversing **File:** `first-mark.elf` **Flag:** `HTB{cut_f0r_th3_P1NT}`

---

1. Summary

`first-mark.elf` is a tiny (4,992-byte) statically-linked **32-bit RISC-V bare-metal firmware image**. It reads a 16-byte candidate from a fixed RAM address, pushes each byte through a four-stage transformation, and compares the result against a 16-byte digest baked into `.rodata`. If every byte matches it prints `ACCEPTED`, otherwise it stays silent.

The twist — and the whole challenge — is that all four transformation stages are **custom (non-standard) RISC-V instructions**. They use the RISC-V *custom-0* (`0x0b`) and *custom-1* (`0x2b`) opcode spaces, so no off-the-shelf disassembler knows what they do; every tool decodes the surrounding RV32I fine and then stalls at exactly those four words. The challenge text hands us the semantics of the third rune only, and tells us to infer the other three "from the company they keep".

The solve is:

1. Disassemble the RV32I skeleton to recover the loop, the three lookup tables and the digest.
2. Use the *given* semantics of rune 3 to run its state/carry chain backwards, which yields the exact intermediate value entering rune 3 on every round — with **no guessing**.
3. Infer rune 1 and rune 2 from context (their operands, and the masking the compiler emits around them), then invert them. Rune 1 is `ROTR8`, rune 2 is a **GF(2^8) multiply** with the AES polynomial `0x11B`.
4. Rune 4 turns out to be an "attest" instruction — a compare that traps on mismatch. It never needs to be inverted.

Because every stage is a bijection, the passing input is **unique**.

---

2. Reconnaissance

```
$ file first-mark.elf
first-mark.elf: ELF 32-bit LSB executable, UCB RISC-V, soft-float ABI,
                version 1 (SYSV), statically linked, stripped
```

No `readelf` / `objdump` for RISC-V on the box, so the headers were parsed with `pyelftools`:

Section	VMA	Size
`.text`	`0x20000000`	`0x178` (376 B)
`.rodata`	`0x20000178`	`0x0bc` (188 B)
`.bss`	`0x80000000`	`0x010` (16 B)
`.riscv.attributes`	—	`0x02a`

Two `PT_LOAD` segments (`0x20000000` R-X, `0x80000000` RW-), entry point `0x20000000`, and a `PT_GNU_STACK`. The load addresses are the giveaway: `0x2000_0000` for flash and `0x8000_0000` for RAM is the classic QEMU RISC-V bare-metal layout. There is no libc and no OS — this is firmware.

The architecture attributes are worth reading, because they are the first hint:

```
A)...riscv...rv32i2p1_m2p0_zmmul1p0
```

The ISA string is plain `rv32i` + `m` / `zmmul`. **No custom extension is declared.** So whatever the four mystery words are, they are not part of any advertised extension — they are undocumented on purpose, exactly as the challenge text says.

2.1 `.rodata`

```
20000178  74 68 65 20 73 74 6f 6e 65 20 77 69 74 6e 65 73  |the stone witnes|
20000188  73 65 73 2e 20 69 74 20 64 6f 65 73 20 6e 6f 74  |ses. it does not|
20000198  20 62 61 72 67 61 69 6e 2e 20 72 65 61 64 20 74  | bargain. read t|
200001a8  68 65 20 66 6f 75 72 20 6d 61 72 6b 73 20 6f 72  |he four marks or|
200001b8  20 62 65 20 63 61 73 74 20 6f 75 74 2e 00 00 00  | be cast out....|
200001c8  41 43 43 45 50 54 45 44 3a 20 54 68 65 20 46 69  |ACCEPTED: The Fi|
200001d8  72 73 74 20 4d 61 72 6b 20 77 61 73 20 63 75 74  |rst Mark was cut|
200001e8  20 69 6e 20 73 74 65 65 6c 2e 00 00 03 07 01 05  | in steel.......|
200001f8  02 06 04 00 03 07 01 05 02 06 04 00 03 02 03 02  |................|
20000208  05 07 02 03 05 07 02 03 05 07 02 03 11 7a 35 90  |.............z5.|
20000218  7e 88 b0 59 79 7f 56 6a 3a 10 e9 05 20 6b 65 65  |~..Yy.Vj:... kee|
20000228  70 20 79 6f 75 72 20 73 74 65 65 6c              |p your steel|
```

Past the two strings there are four 16-byte blobs:

Address	Contents	Role (confirmed later from code)
`0x200001f4`	`03 07 01 05 02 06 04 00` ×2	**rot table** — rune 1's operand
`0x20000204`	`03 02 03 02 05 07 02 03 05 07 02 03 05 07 02 03`	**mul table** — rune 2's operand
`0x20000214`	`11 7a 35 90 7e 88 b0 59 79 7f 56 6a 3a 10 e9 05`	**target digest**
`0x20000224`	`" keep your steel"`	**decoy** — never referenced by code

That last blob is a nice piece of misdirection: it is exactly 16 bytes, has **no NUL terminator** (so it cannot be a printed string), and sits right after the digest — it looks exactly like a 16-byte key. It is even a pun on the challenge text's "learn what those four runes do from the company they *keep*". I tested it as a known plaintext

early on; it matched nothing. Cross-referencing the code proves it is dead data: the only `.rodata` addresses the program ever forms are `0x20000178`, `0x200001c8`, `0x200001f4`, `0x20000204` and `0x20000214`.

Note the structure of the mul table: `[3,2,3,2]` then `[5,7,2,3]` repeated three times. The rot table has period 8. This means rounds 12–15 use *identical* table parameters to rounds 4–7 — a detail that becomes a useful sanity check on the final answer.

---

3. Disassembly

Since no RISC-V binutils were available, I wrote a ~40-line RV32IM disassembler that decodes the standard opcodes and, critically, **flags anything in the custom opcode spaces instead of bailing out**:

```
if op in (0x0b, 0x2b, 0x5b, 0x7b):
    cn = {0x0b:'CUSTOM0', 0x2b:'CUSTOM1', 0x5b:'CUSTOM2', 0x7b:'CUSTOM3'}[op]
    return ">>> %s f3=%d f7=%#02x  %s, %s, %s" % (cn, f3, f7, R[rd], R[rs1], R[rs2])
```

That single change is what makes the challenge tractable: the four unknown words are all R-type, so `rd` / `rs1` / `rs2` / `funct3` / `funct7` decode normally even though the mnemonic is unknown. Their *operands* are visible, and the operands are the evidence.

3.1 `_start` — `0x20000000`

```
20000000: auipc   sp, 0x60010        # sp = 0x80010000
20000004: mv      sp, sp
20000008: auipc   a0, 0x60000        # a0 = 0x80000000   (bss start)
2000000c: addi    a0, a0, -8
20000010: auipc   a1, 0x60000        # a1 = 0x80000010   (bss end)
20000014: mv      a1, a1
20000018: bgeu    a0, a1, 0x20000028
2000001c: sw      zero, 0(a0)        # zero .bss
20000020: addi    a0, a0, 4
20000024: bltu    a0, a1, 0x2000001c
20000028: auipc   a0, 0x0            # src = 0x20000234  (.data — empty)
2000002c: addi    a0, a0, 524
20000030: auipc   a1, 0x60000
20000034: addi    a1, a1, -48        # dst = 0x80000000
20000038: auipc   a2, 0x60000
2000003c: addi    a2, a2, -56        # end = 0x80000000  → dst == end, loop never runs
20000040: bgeu    a1, a2, 0x20000058
...
20000058: jal     ra, 0x20000094     # main()
2000005c: jal     zero, 0x2000005c   # halt: spin forever
```

Standard crt0: set stack, zero BSS, copy `.data` (which is empty — `dst == end`), call `main`, then spin. So the 16 BSS bytes at `0x80000000` are simply **zeroed**, and the firmware never reads a character from anywhere. The candidate is expected to be *placed* at `0x80000000` by whatever harness runs the stone.

3.2 `puts` — `0x20000060`

```
20000060: lui     a5, 0x10000        # a5 = 0x10000000  (16550 UART, QEMU virt)
20000064: lbu     a4, 0(a0)
20000068: bne     a4, zero, 0x20000080
2000006c: lw      a4, 0(a5)          # poll TX status
20000070: blt     a4, zero, 0x2000006c
20000074: li      a4, 10             # '\n'
20000078: sw      a4, 0(a5)
2000007c: jalr    zero, 0(ra)
20000080: addi    a0, a0, 1
20000084: lw      a3, 0(a5)
20000088: blt     a3, zero, 0x20000084
2000008c: sw      a4, 0(a5)
20000090: jal     zero, 0x20000064
```

A NUL-terminated `puts` over the memory-mapped UART at `0x10000000`, busy-waiting while the status word is negative (bit 31 = busy) and appending a newline. Confirms the QEMU `virt` target.

3.3 `main` — `0x20000094`

```
20000094: addi    sp, sp, -16
20000098: auipc   a0, 0x0
2000009c: addi    a0, a0, 224        # a0 = 0x20000178  "the stone witnesses..."
200000a0: sw      ra, 12(sp)
200000a4: jal     ra, 0x20000060     # puts(banner)
200000a8: auipc   a0, 0x60000
200000ac: addi    a0, a0, -168       # a0 = 0x80000000  (the 16-byte candidate)
200000b0: jal     ra, 0x200000d0     # check(a0)
200000b4: auipc   a0, 0x0
200000b8: addi    a0, a0, 276        # a0 = 0x200001c8  "ACCEPTED: ..."
200000bc: jal     ra, 0x20000060     # puts(accepted)
200000c0: lw      ra, 12(sp)
200000c4: li      a0, 0
200000cc: jalr    zero, 0(ra)
```

Note there is **no branch** on `check`'s return value, and `check` unconditionally returns 0. `main` prints the banner, calls `check`, and prints `ACCEPTED` — with nothing in between that could skip the success message. The only way the program can fail to print `ACCEPTED` is if `check` never returns, i.e. if something inside it **traps**. That is the first strong clue about rune 4: it must be the thing that aborts execution. "answer with only two words: true, or nothing at all."

3.4 `check` — `0x200000d0` (the heart)

```
200000d0: addi    sp, sp, -48
200000d4: sw      ra, 44(sp)          ; ... callee-saved spills ...
200000ec: mv      s0, a0              # s0 = input pointer
200000f0: li      s1, 0               # s1 = i = 0
200000f4: li      a2, 165             # a2 = 0xA5   ⟵ rune 3's initial state
200000f8: auipc   s2, 0x0
200000fc: addi    s2, s2, 252         # s2 = 0x200001f4  rot table
20000100: auipc   s3, 0x0
20000104: addi    s3, s3, 260         # s3 = 0x20000204  mul table
20000108: auipc   s4, 0x0
2000010c: addi    s4, s4, 268         # s4 = 0x20000214  target digest

.loop:
20000110: add     t0, s0, s1
20000114: lbu     a0, 0(t0)           # a0 = input[i]
20000118: add     t0, s2, s1
2000011c: lbu     a1, 0(t0)
20000120: andi    a1, a1, 7           # a1 = rot[i] & 7          ⟵ !!
20000124: >>> CUSTOM0 f3=0 f7=0x0  a0, a0, a1      ; RUNE 1
20000128: add     t0, s3, s1
2000012c: lbu     a1, 0(t0)           # a1 = mul[i]
20000130: >>> CUSTOM0 f3=1 f7=0x0  a0, a0, a1      ; RUNE 2
20000134: >>> CUSTOM1 f3=0 f7=0x0  a0, a0, a2      ; RUNE 3  (a2 = state)
20000138: mv      a2, a0              # state = result
2000013c: add     t0, s4, s1
20000140: lbu     a3, 0(t0)           # a3 = target[i]
20000144: >>> CUSTOM1 f3=0 f7=0x1  zero, a0, a3    ; RUNE 4  (rd = zero!)
20000148: addi    s1, s1, 1
2000014c: li      t0, 16
20000150: blt     s1, t0, 0x20000110
20000154: li      a0, 0               # always returns 0
```

In C:

```c
uint8_t state = 0xA5;
for (int i = 0; i < 16; i++) {
    uint8_t v = input[i];
    v = rune1(v, rot[i] & 7);   // CUSTOM0 funct3=0
    v = rune2(v, mul[i]);       // CUSTOM0 funct3=1
    v = rune3(v, state);        // CUSTOM1 funct3=0 funct7=0
    state = v;
    rune4(v, target[i]);        // CUSTOM1 funct3=0 funct7=1, rd=x0
}
return 0;
```

---

4. Recovering the four runes

Rune 4 — `CUSTOM1 / funct7=1` — *attest*

`rd` is hard-wired to `x0`, so the instruction produces **no architectural result**. An instruction that consumes two values and writes nothing is only useful for its side effects. Combined with §3.3 — `main` has no conditional branch, yet the challenge promises "true, or nothing at all" — rune 4 must be a *compare-and-trap*: continue if `a0 == a3`, otherwise abort.

This matters for the writeup but not for the maths: rune 4 is a pure assertion, so **inverting it is unnecessary**. It just tells us the constraint is `rune3_output[i] == target[i]` for all 16 rounds.

Rune 3 — `CUSTOM1` / `funct7=0` — given

From the final dry stone in the vault:

```
out   = a0 ^ state ^ carry
carry = old_a0 & state          (carry starts at 0)
state = out                     (state starts at 0xA5)
```

The code confirms both constants: `li a2, 165` (`0xA5`) before the loop, and `mv a2, a0` right after rune 3.

This is the lever that cracks the whole challenge. Rune 4 pins `out_i == target[i]`, and `state_i` is just `target[i-1]` (with `state_0 = 0xA5`). So the value entering rune 3 — equivalently, the **output of rune 2** — can be computed for every round with zero knowledge of runes 1 and 2:

```python
state, carry, A = 0xA5, 0, []
for i in range(16):
    out = target[i]
    a   = out ^ state ^ carry     # == rune2 output for round i
    A.append(a)
    carry = a & state             # carry uses the *old* a0 and *old* state
    state = out
```

```
post-rune2 values:
b4 cf 4e ef cb 76 4e e1 80 06 29 15 44 6a d3 fc
```

The search space is now only two byte-wise operations deep, and each round is independent.

Rune 1 — `CUSTOM0` / `funct3=0` — `ROTR8`

The evidence is the `andi a1, a1, 7` at `0x20000120`. The compiler masks rune 1's second operand to **0–7** before handing it over. A 3-bit operand on an 8-bit lane is a shift or rotate distance. And `rot[7] = rot[15] = 0`: a rotate by 0 is the identity (harmless), whereas a *shift* by 0 would also be identity but a shift by 7 (`rot[1]`) would destroy 7 bits of input, making the digest non-unique and the challenge unsolvable. So rune 1 is a **rotate**; direction is the only remaining bit of freedom.

Rune 2 — `CUSTOM0` / `funct3=1` — GF(2^8) multiply, poly `0x11B`

Rune 2's operand is *not* masked, and the mul table only ever contains `{2, 3, 5, 7}` — all odd-or-2, all small primes. Note rune 1 could not have been the multiply: its operand is masked to 0–7 and the table contains `0`, and multiplying by 0 annihilates the byte. So the multiply, if there is one, has to be rune 2. In GF(2^8) every non-zero element is invertible, and `{2,3,5,7}` are exactly the kind of small constants used in AES's MixColumns.

Rather than argue further, I brute-forced it. For each candidate pair (rune1, rune2) drawn from rotates/shifts/xor/add/sub and GF-multiplies over 28 different irreducible polynomials, I inverted `A[i]` back to `input[i]` and scored the 16 recovered bytes for printability:

```
score   45  rune1=rotr  rune2=gf11b   → 'cut_f0r_th3_P1NT'
score   16  rune1=rotr  rune2=gf1f3   → 'cOt.asr_z..TPr.T'
score   15  rune1=rotr  rune2=gf1bd   → 'c.t59-r_.R..P,{T'
score   15  rune1=rotr  rune2=gf1f9   → 'c.tqiVr_j...PWYT'
score   12  rune1=rotr  rune2=gf177   → 'cnt..:r_.|_.P;-T'
```

One combination separates from the field by a mile: **rune 1 = `ROTR8`, rune 2 = GF(2^8) multiply modulo the AES polynomial `0x11B`**, giving fully coherent leetspeak. Every other polynomial produces noise.

---

5. Solution

Full forward verification of the recovered input, printing every round:

```
 i= 0 in=63(c) rotr3→6c gf*3→b4  out=11  tgt=11 ok
 i= 1 in=75(u) rotr7→ea gf*2→cf  out=7a  tgt=7a ok
 i= 2 in=74(t) rotr1→3a gf*3→4e  out=35  tgt=35 ok
 i= 3 in=5f(_) rotr5→fa gf*2→ef  out=90  tgt=90 ok
 i= 4 in=66(f) rotr2→99 gf*5→cb  out=7e  tgt=7e ok
 i= 5 in=30(0) rotr6→c0 gf*7→76  out=88  tgt=88 ok
 i= 6 in=72(r) rotr4→27 gf*2→4e  out=b0  tgt=b0 ok
 i= 7 in=5f(_) rotr0→5f gf*3→e1  out=59  tgt=59 ok
 i= 8 in=74(t) rotr3→8e gf*5→80  out=79  tgt=79 ok
 i= 9 in=68(h) rotr7→d0 gf*7→06  out=7f  tgt=7f ok
 i=10 in=33(3) rotr1→99 gf*2→29  out=56  tgt=56 ok
 i=11 in=5f(_) rotr5→fa gf*3→15  out=6a  tgt=6a ok
 i=12 in=50(P) rotr2→14 gf*5→44  out=3a  tgt=3a ok
 i=13 in=31(1) rotr6→c4 gf*7→6a  out=10  tgt=10 ok
 i=14 in=4e(N) rotr4→e4 gf*2→d3  out=e9  tgt=e9 ok
 i=15 in=54(T) rotr0→54 gf*3→fc  out=05  tgt=05 ok
```

All 16 bytes reproduce the digest exactly:

```
cut_f0r_th3_P1NT → 117a35907e88b059797f566a3a10e905  == target
```

5.1 Uniqueness

`ROTR8` is a bijection; GF(2^8) multiplication by a non-zero constant is a bijection; and rune 3 is fully determined once the outputs are fixed (they must equal the digest) because its state and carry chain is driven by those outputs. Every round is therefore forced, and **exactly one 16-byte input satisfies the stone**. There is no second preimage to worry about.

A near-miss worth recording: the story leans hard on kings, so `cut_f0r_th3_K1NG` is the obvious guess. It is wrong —

```
cut_f0r_th3_K1NG → 117a35907e88b059797f566ac9811a52  ≠ target
```

It agrees on the first 12 bytes and diverges from round 12 on. This is a good demonstration of why the maths matters more than the narrative: rounds 12–15 reuse the same table parameters as rounds 4–7 (§2.1), so a wrong guess there is not detectable by pattern-matching the tables — only by running the pipeline.

5.2 End-to-end confirmation

To leave nothing resting on my algebra, I wrote a small RV32IM interpreter that loads the `PT_LOAD` segments, emulates the UART at `0x10000000`, implements the four runes as recovered (with rune 4 as compare-and-trap), and injects the candidate into `0x80000000` just before `main` calls `check`:

```
input=b'cut_f0r_th3_P1NT'  trap=None
  output='the stone witnesses. it does not bargain. read the four marks or be cast out.
          ACCEPTED: The First Mark was cut in steel.'

input=b'wrong_input_1234'  trap=REJECTED at pc=0x20000144
  output='the stone witnesses. it does not bargain. read the four marks or be cast out.'
```

The real binary's own control flow, driven by the recovered instruction semantics, prints `ACCEPTED`. A wrong input traps precisely at the rune-4 word (`0x20000144`) and the stone stays silent — "true, or nothing at all", exactly as promised.

---

6. Solver

```python
#!/usr/bin/env python3
"""First Mark — recover the 16-byte input accepted by the RISC-V 'stone'."""

M = 0xFF

# --- .rodata @ 0x200001f4 / 0x20000204 / 0x20000214 -------------------------
ROT = [3, 7, 1, 5, 2, 6, 4, 0] * 2
MUL = [3, 2, 3, 2, 5, 7, 2, 3, 5, 7, 2, 3, 5, 7, 2, 3]
TGT = [0x11, 0x7a, 0x35, 0x90, 0x7e, 0x88, 0xb0, 0x59,
       0x79, 0x7f, 0x56, 0x6a, 0x3a, 0x10, 0xe9, 0x05]

# --- the four undocumented runes --------------------------------------------
def rune1(x, n):                        # CUSTOM0 f3=0 : rotate right, 8-bit
    n &= 7
    return ((x >> n) | (x << (8 - n))) & M

def rune2(x, k, poly=0x11B):            # CUSTOM0 f3=1 : GF(2^8) multiply
    r = 0
    for _ in range(8):
        if k & 1:
            r ^= x
        k >>= 1
        x <<= 1
        if x & 0x100:
            x ^= poly
    return r & M

# rune3 = CUSTOM1 f7=0 : out = a ^ state ^ carry ; carry = a & state
# rune4 = CUSTOM1 f7=1 : attest (trap unless a == target[i]); rd = x0

def gf_inv(k, poly=0x11B):
    return next(c for c in range(1, 256) if rune2(k, c, poly) == 1)

def solve():
    state, carry, out = 0xA5, 0, bytearray()
    for i in range(16):
        # rune 4 forces rune 3's output to equal the digest byte
        o = TGT[i]
        # invert rune 3: a = out ^ state ^ carry
        a = o ^ state ^ carry
        carry, state = a & state, o
        # invert rune 2, then rune 1
        v = rune2(a, gf_inv(MUL[i]))
        out.append(rune1(v, -ROT[i] & 7))   # rotl == rotr by the complement
    return bytes(out)

def check(inp):
    state, carry = 0xA5, 0
    for i in range(16):
        v = rune2(rune1(inp[i], ROT[i] & 7), MUL[i])
        o = v ^ state ^ carry
        carry, state = v & state, o
        if o != TGT[i]:
            return False
    return True

if __name__ == "__main__":
    ans = solve()
    assert check(ans), "pipeline mismatch"
    print("input :", ans.decode())
    print("flag  : HTB{%s}" % ans.decode())
```

```
$ python3 solve.py
input : cut_f0r_th3_P1NT
flag  : HTB{cut_f0r_th3_P1NT}
```

7. Takeaways

- **Custom opcode spaces are a legitimate hiding place.** RISC-V reserves `custom-0`…`custom-3` (`0x0b`, `0x2b`, `0x5b`, `0x7b`) for vendor extensions. Standard disassemblers emit `.insn` / `unknown` and stop, which is the "wall every common chisel hits". Because the encoding is still R-type, though, the register fields decode for free — patching a disassembler to *report* unknown opcodes with their operands, instead of aborting, converts a dead end into a data-flow problem.
- **The compiler leaks the semantics.** `andi a1, a1, 7` on one operand is a 3-bit shift distance. `rd = x0` means "executed for side effects only". A `0xA5` seed materialised right before the loop and copied back from the result is a state register. None of that requires the missing manual.
- **Solve the given stage first to collapse the search space.** Rune 3's disclosed rule, plus rune 4 pinning each round's output to the digest, determines the rune-2 output for all 16 rounds outright. What could have been a 4-unknown search became a 2-operation brute force scored on printable ASCII.
- **Don't let the narrative pick your bytes.** `" keep your steel"` is a perfectly shaped 16-byte decoy that no code path touches, and `K1NG` is the thematically obvious ending that the algebra rejects. Cross-referencing every `.rodata` address against the code, and verifying forward in an emulator, is what separates the answer from the story.

> **Flag → HTB{cut_f0r_th3_P1NT}**