Ghost in the Machine

Full CTF Challenge Writeup — Stack-Based VM Reverse Engineering

Solved by **ssstrickys** · March 31, 2026

Challenge Name	Ghost (Prescription_Pad_2.zip)
Category	Reverse Engineering
Architecture	Linux x86-64 ELF, PIE, dynamically linked, stripped
Compiler	GCC 13.3.0 (Ubuntu 24.04)
Binary Size	15,544 bytes
Bytecode	666 bytes at file offset 0x3040 (vaddr 0x4040 at runtime)
Passphrase Len	37 printable ASCII characters (null-terminated at index 37)
Flag	`SDG{ad3ab2968f77e8e6f3ebc626cca75fd4}`
Status	**SOLVED**

TL;DR: The ghost binary embeds a 666-byte custom stack-based VM that validates a 37-character passphrase through 45 sequential comparison checks. The key to solving it was identifying a shared **key-value store** (opcodes 0x31/0x30) that creates a rolling state register chaining all character checks together. Once the emulator correctly modelled this, a printable-ASCII sensitivity-based solver recovered the full flag in 36 iterations.

1. Challenge Overview

The challenge provides a ZIP archive containing a stripped Linux x86-64 PIE ELF binary named **ghost** and a README that reads:

```
Ghost in the Machine
====================
The binary is a Linux x86_64 ELF.

Run: ./ghost <passphrase>

The tool responds with "Correct!" or "Wrong."
Something unusual is running inside.
```

Running the binary with a test passphrase confirms the interface:

```
$ ./ghost test
Wrong.
$ ./ghost SDG{ad3ab2968f77e8e6f3ebc626cca75fd4}
Correct!
```

The hint *"Something unusual is running inside"* immediately suggests a custom virtual machine or obfuscated interpreter, which static analysis confirms.

2. Static Analysis

2.1 File Reconnaissance

```
$ file ghost
ghost: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
       dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
       BuildID[sha1]=876f4556b6f4a0c76094f9693f058405b40ab1c0,
       for GNU/Linux 3.2.0, stripped

$ readelf -S ghost | grep -E 'Name|data|text|bss'
  [16] .text     PROGBITS  0x10e0  0x10e0  0x619
  [18] .rodata   PROGBITS  0x2000  0x2000  0x1f4
  [25] .data     PROGBITS  0x4000  0x3000  0x440
  [26] .bss      NOBITS    0x4440  0x3440  0x260
```

Key observations: stripped binary (no symbols), PIE (base address 0 in file), .data section 0x440 bytes starting at file offset 0x3000. The bytecode lives at file offset **0x3040** (= .data + 0x40, vaddr 0x4040 at runtime).

2.2 Locating the Bytecode

Searching for the bytecode pointer in main() disassembly:

```
; From main() at 0x112c:
lea r9, [rip + 0x2f0d]    ; r9 = 0x1134 + 0x2f0d = vaddr 0x4041
                           ; file offset: 0x2d98 + (0x4041-0x3d98) = 0x3041
                           ; -> file[0x3040] = bytecode base

; Also at 0x1119:
mov r11d, [rip + 0x2f00]  ; r11d = bytecode length = 666 (0x29a)
```

The bytecode is **666 bytes** long. The VM step counter runs to a maximum of **100,000 steps** (0x186a0), after which it also branches to Correct — a quirk that never triggers in practice since the flag check terminates early.

2.3 Main() Structure

```
main(argc, argv):
  if argc != 2: print usage, exit(1)

  rbx = argv[1]               ; passphrase string pointer
  ebp = strlen(argv[1])       ; passphrase length
  r11d = bytecode_length      ; = 666
  r9  = bytecode_base         ; file offset 0x3040
  r10 = jump_table            ; opcode dispatch table at 0x2068
  r8d = 0                     ; VM program counter
  edx = 0                     ; step counter

  ; --- VM dispatch loop ---
loop:
  edx += 1
  ecx = bytecode[r8]          ; fetch opcode
  al  = ecx - 0x10            ; normalize (valid: 0x10..0x72)
  if al > 0x62: goto WRONG    ; out-of-range opcode
  jmp [r10 + al*4]            ; dispatch to handler
```

```
  ; --- Exit paths ---
WRONG:   puts("Wrong.");   exit(1)
CORRECT: puts("Correct!"); exit(0)
MAXSTEP: if edx == 100000: goto CORRECT
```

3. ISA Recovery — The Virtual Machine Instruction Set

The jump table at vaddr **0x2068** contains 99 signed 32-bit offsets, one per opcode index (opcodes 0x10–0x72). Disassembling each handler reveals the complete instruction set. The VM uses a single unified **byte-width stack** (all values truncated to 8 bits) plus a 256-slot **key-value store** in the BSS segment.

3.1 Complete Instruction Set

Opcode	Encoding	Operation	Notes
PUSH	10 XX	stack.push(XX)	1-byte immediate
POP	11	stack.pop() [discard]	
DUP	12	stack.push(stack.top())	
ADD	20	b=pop; a=pop; push (a+b)&0xFF	
SUB	21	b=pop; a=pop; push (a-b)&0xFF	
XOR	22	b=pop; a=pop; push a^b	
MUL	23	b=pop; a=pop; push (a*b)&0xFF	Uses x86 MUL, low byte only
ROL	24	n=pop; a=pop; push ROL8(a,n)	n masked to 3 bits (n&7)
AND	25	b=pop; a=pop; push a&b	Used for nibble isolation
KVLOAD	30 XX	push kv_store[XX]	Reads shared KV store
KVSTORE	31 XX	kv_store[XX] = pop()	Writes shared KV store
INPUT	40 XX	push passphrase[XX]	XX is a direct index into argv[1]
CMP	50	b=pop; a=pop; flag=(a==b)	Sets 1-bit flag register
JNZ	60 XX YY	if flag: pc = (YY<<8)|XX	16-bit little-endian target
JZ	61 XX YY	if !flag: pc = (YY<<8)|XX	16-bit little-endian target
JMP	62 XX YY	pc = (YY<<8)|XX	Unconditional
CORRECT	71	print 'Correct!'; exit(0)	Terminal opcode
WRONG	72	print 'Wrong.'; exit(1)	Terminal opcode
NOP	99 / 02	pc += 1	Padding bytes

3.2 The KV Store — The Critical Hidden Mechanism

The most important and subtle mechanism in this VM is the shared key-value store operated by opcodes **0x31 (KVSTORE)** and **0x30 (KVLOAD)**.

From the handler disassembly, both instructions access the same 256-byte BSS buffer at runtime vaddr **0x4480** (file section .bss). KVSTORE pops a value off the stack and writes it to **kv_store[operand]**. KVLOAD reads **kv_store[operand]** and pushes it onto the stack. The store is zero-initialized at process start.

Early in the bytecode (offset 16–18), the VM writes the seed value:

```
; Bytecode offset 16:
10 5a    PUSH 0x5a       ; push constant 0x5a ('Z')
31 00    KVSTORE kv[0]   ; kv_store[0] = 0x5a  (rolling state init)
```

After checking each character, the VM updates the rolling state:

```
; After each character check (e.g. for passphrase[3]):
30 00    KVLOAD  kv[0]   ; push current rolling state
40 03    INPUT[3]        ; push passphrase[3]
20       ADD             ; state = (state + char) & 0xFF
31 00    KVSTORE kv[0]   ; kv_store[0] = updated state
```

This means every character check depends on the result of all previous checks. The rolling state values observed during the correct solution were:

Characters	kv_store[0] entering block	Characters checked
Init	0x5a	—
[0]–[2]	0x5a	S, D, G (kv not used in these checks)
[3]	0x5a	{ (XOR with kv[0]: '{' ^ 0x5a = 0x21 ✓)
[4]–[7]	0xd5	a, d, 3, a
[8]–[13]	0x37	b, 2, 9, 6, 8, f
[14]–[19]	0x9d	7, 7, e, 8, e, 6
[20]–[25]	0x02	f, 3, e, b, c, 6
[26]–[31]	0x64	2, 6, c, c, a, 7
[32]–[35]	0xc7 / 0x2d	5, f, d, 4

Why this was missed initially: The first emulator discarded the value in KVSTORE (treating it as a dead write) and returned 0 from every KVLOAD. This made all character checks appear to compare against static bytecode constants, and falsely identified passphrase[3] as '!' (0x21) rather than '{' (0x7b = 0x21 XOR 0x5a).

4. Bytecode Layout

4.1 High-Level Structure

Region	Offsets	Size	Purpose
Length check A	0–7	8 B	INPUT[37]; CMP==0; JZ->665 (null term check)
Length check B	8–15	8 B	INPUT[36]; CMP==0; JNZ->665 (non-null closing char)
KV init	16–19	4 B	PUSH 0x5a; KVSTORE kv[0] (seed rolling state)
Char check [0]	20–33	14 B	Verify 'S': (INPUT[0] XOR 0x13) + 0x07 == 0x47
Char check [1]	34–47	14 B	Verify 'D': (INPUT[1] * 0x21) XOR 0x6c == 0xa8
Char check [2]	48–61	14 B	Verify 'G': ROL(INPUT[2], 3) XOR 0x27 == 0x1d
Char check [3]	62–79	18 B	Verify '{': INPUT[3] XOR kv[0] == 0x21; update kv
Char checks [4]–[35]	80–661	582 B	32 chars x ~18 bytes; each uses 2 CMPs + kv update
Terminal CORRECT	664	1 B	Opcode 0x71 -> Correct!
Terminal WRONG	665	1 B	Opcode 0x72 -> Wrong. (also target of all fail branches)

4.2 Per-Character Check Block Anatomy

Characters [3]–[35] follow a consistent 18-byte block structure. Each block applies two independent transformations to the character, compares each result to a hardcoded expected value, and then updates the rolling state in kv_store[0]:

```
; ■■ Block structure for character at passphrase[N] ■■

; Check 1: first transform
40 NN        INPUT[N]          ; push passphrase[N]
10 K1        PUSH k1           ; push transform operand
<op>         <arithmetic>      ; apply op (XOR / AND / ROL / ADD / MUL)
[10 K2 <op>] [optional chain]  ; some checks have 2 operations
10 E1        PUSH expected1    ; push expected result
50           CMP               ; compare
61 99 02     JZ -> 665         ; jump to WRONG if mismatch

; KV state update
30 00        KVLOAD  kv[0]     ; push current rolling state
40 NN        INPUT[N]          ; push passphrase[N] again
20           ADD               ; new_state = (state + char) & 0xFF
31 00        KVSTORE kv[0]     ; persist updated state

; Check 2: second transform (using updated kv state)
40 NN        INPUT[N]          ; push passphrase[N]
30 00        KVLOAD  kv[0]     ; push updated state
22           XOR               ; mix character with state
10 E2        PUSH expected2
50           CMP
61 99 02     JZ -> 665
```

4.3 Annotated Bytecode — First 80 Bytes

The following annotated dump illustrates the full init + first four character checks:

```
Offset  Bytes          Mnemonic        Comment
------  ------         --------        -------
00      40 25          INPUT[37]       ; check null terminator
02      10 00          PUSH 0x00
04      50             CMP             ; passphrase[37] == 0?
05      61 99 02       JZ  -> 665      ; WRONG if not null (too long)
08      40 24          INPUT[36]       ; check closing char not null
0a      10 00          PUSH 0x00
0c      50             CMP             ; passphrase[36] == 0?
0d      60 99 02       JNZ -> 665      ; WRONG if null (too short)
10      10 5a          PUSH 0x5a       ; seed value for rolling state
12      31 00          KVSTORE kv[0]   ; kv[0] = 0x5a
14      40 00          INPUT[0]        ; passphrase[0] = 'S' = 0x53
16      10 13          PUSH 0x13
18      22             XOR             ; 0x53 ^ 0x13 = 0x40
19      10 07          PUSH 0x07
1b      20             ADD             ; 0x40 + 0x07 = 0x47
1c      10 47          PUSH 0x47
1e      50             CMP             ; 0x47 == 0x47? PASS
1f      61 99 02       JZ  -> 665
22      40 01          INPUT[1]        ; passphrase[1] = 'D' = 0x44
24      10 21          PUSH 0x21
26      23             MUL             ; 0x44 * 0x21 = 0x24 (low byte)
                                       ;   actually 68*33=2244, &0xFF=0xc4
27      10 6c          PUSH 0x6c
29      22             XOR             ; 0xc4 ^ 0x6c = 0xa8
2a      10 a8          PUSH 0xa8
2c      50             CMP             ; 0xa8 == 0xa8? PASS
2d      61 99 02       JZ  -> 665
30      40 02          INPUT[2]        ; passphrase[2] = 'G' = 0x47
32      10 03          PUSH 0x03
34      24             ROL             ; ROL8(0x47, 3) = 0x3a... wait
                                       ; 0x47=0100_0111, ROL3=0011_1010=0x3a
35      10 27          PUSH 0x27
37      22             XOR             ; 0x3a ^ 0x27 = 0x1d
38      10 1d          PUSH 0x1d
3a      50             CMP             ; 0x1d == 0x1d? PASS
3b      61 99 02       JZ  -> 665
3e      40 03          INPUT[3]        ; passphrase[3] = '{' = 0x7b
40      30 00          KVLOAD kv[0]    ; push 0x5a (initial seed)
42      22             XOR             ; 0x7b ^ 0x5a = 0x21
43      10 21          PUSH 0x21
45      50             CMP             ; 0x21 == 0x21? PASS
46      61 99 02       JZ  -> 665
49      30 00          KVLOAD kv[0]    ; push 0x5a
4b      40 03          INPUT[3]        ; push '{' = 0x7b
4d      20             ADD             ; (0x5a + 0x7b) & 0xFF = 0xd5
4e      31 00          KVSTORE kv[0]   ; kv[0] = 0xd5  <-- state updated
```

5. Building the Emulator

With the ISA fully recovered, a Python emulator is straightforward to implement. The complete, final, correct emulator is shown below. Note the kv dict and the distinction between INPUT (0x40, reads passphrase directly) and KVLOAD (0x30, reads from the internal store):

```
#!/usr/bin/env python3
# ghost_vm.py — correct emulator for ghost CTF VM

def rol8(val, n):
    n = n & 7
    return ((val << n) | (val >> (8 - n))) & 0xFF

def emulate(bytecode, passphrase, collect_cmps=False):
    """
    Emulate the ghost VM.
    Returns (result: bool, cmps: list)
    cmps entries: {'pc': int, 'a': int, 'b': int, 'match': bool}
    """
    stack = []
    kv    = {}          # key-value store: written by 0x31, read by 0x30
    flag  = 0           # single-bit comparison flag
    pc    = 0
    cmps  = []

    def push(v): stack.append(v & 0xFF)
    def pop():   return stack.pop() if stack else 0

    while pc < len(bytecode):
        op = bytecode[pc]

        if   op == 0x10:                        # PUSH immediate
            push(bytecode[pc+1]); pc += 2
        elif op == 0x11: pop(); pc += 1          # POP (discard)
        elif op == 0x12:                         # DUP
            if stack: push(stack[-1])
            pc += 1
        elif op == 0x20:                         # ADD
            b=pop(); a=pop(); push((a+b)&0xFF); pc+=1
        elif op == 0x21:                         # SUB
            b=pop(); a=pop(); push((a-b)&0xFF); pc+=1
        elif op == 0x22:                         # XOR
            b=pop(); a=pop(); push(a^b);         pc+=1
        elif op == 0x23:                         # MUL
            b=pop(); a=pop(); push((a*b)&0xFF); pc+=1
        elif op == 0x24:                         # ROL8
            b=pop(); a=pop(); push(rol8(a,b));  pc+=1
        elif op == 0x25:                         # AND
            b=pop(); a=pop(); push(a&b);         pc+=1

        elif op == 0x30:                         # KVLOAD
            push(kv.get(bytecode[pc+1], 0)); pc += 2
        elif op == 0x31:                         # KVSTORE
            kv[bytecode[pc+1]] = pop(); pc += 2

        elif op == 0x40:                         # INPUT passphrase[XX]
            idx = bytecode[pc+1]
            push(passphrase[idx] if idx < len(passphrase) else 0)
            pc += 2
```

```
        elif op == 0x50:                        # CMP
            b=pop(); a=pop()
            flag = 1 if a == b else 0
            if collect_cmps:
                cmps.append({'pc':pc,'a':a,'b':b,'match':flag==1})
            pc += 1

        elif op == 0x60:                        # JNZ
            t = bytecode[pc+1] | (bytecode[pc+2] << 8)
            pc = t if flag else pc+3
        elif op == 0x61:                        # JZ
            t = bytecode[pc+1] | (bytecode[pc+2] << 8)
            pc = t if not flag else pc+3
        elif op == 0x62:                        # JMP
            pc = bytecode[pc+1] | (bytecode[pc+2] << 8)

        elif op == 0x71: return True,  cmps     # CORRECT
        elif op == 0x72: return False, cmps     # WRONG
        elif op in (0x99, 0x02, 0x00): pc += 1  # NOP
        else: pc += 1                           # unknown: skip

    return False, cmps


if __name__ == '__main__':
    import sys
    data     = open('ghost', 'rb').read()
    bytecode = data[0x3040:0x3040+666]
    flag_candidate = sys.argv[1].encode() if len(sys.argv) > 1 else b''
    result, _ = emulate(bytecode, flag_candidate)
    print('Correct!' if result else 'Wrong.')
```

6. The Solver

6.1 Strategy

Because each character check depends on the rolling kv_store state (which is updated by every preceding character), a pure inversion approach is not straightforward. Instead, a **sensitivity-based sequential brute force** is used:

1. Start with the known prefix **SDG{** (indices 0–3) and closing **}** (index 36), with all interior positions zeroed.
2. Run the emulator and find the first CMP that fails (excluding the intentional length sentinel at pc=12).
3. Scan each unsolved character position across the full printable ASCII range (0x20–0x7e), checking which value fixes the failing CMP without breaking any previously passing ones.
4. Commit the winning value, mark the position solved, and repeat from step 2.
5. Terminate when the emulator returns **True** (CORRECT opcode reached).

Because each check is sensitive to exactly one unsolved character at a time (due to the sequential kv_store update), this greedy approach converges to the unique solution without backtracking.

6.2 Full Solver Source

```
#!/usr/bin/env python3
# ghost_solver.py — recovers the ghost CTF flag

import subprocess
# (paste emulate() and rol8() from ghost_vm.py here)

data     = open('ghost', 'rb').read()
bc       = data[0x3040:0x3040+666]
PRINTABLE = list(range(0x20, 0x7F))   # printable ASCII

# Seed with known prefix and closing brace
known       = bytearray(37)
known[0]    = ord('S')
known[1]    = ord('D')
known[2]    = ord('G')
known[3]    = ord('{')
known[36]   = ord('}')
solved      = {0, 1, 2, 3, 36}

for iteration in range(50):
    result, cmps = emulate(bc, bytes(known), collect_cmps=True)
    if result:
        flag = bytes(known).decode()
        print(f'[+] FLAG: {flag}')
        # Verify against the actual binary
        r = subprocess.run(['./ghost', bytes(known)], capture_output=True)
        print(f'[+] Binary: {r.stdout.strip().decode()}')
        break

    # Find the first genuinely failing CMP
    # (pc=12 is the intentional 'not-null' sentinel — skip it)
    fail = next((c for c in cmps
                 if not c['match'] and c['pc'] != 12), None)
    if not fail:
        print('[-] No failing CMPs but not CORRECT — unexpected state')
```

```
    break

current_pass = sum(1 for c in cmps
                   if c['match'] and c['pc'] != 12)

found_idx = found_val = best_pass = None

for idx in range(37):
    if idx in solved:
        continue
    for v in PRINTABLE:
        test = bytearray(known)
        test[idx] = v
        r2, cmps2 = emulate(bc, bytes(test), collect_cmps=True)
        if r2:                              # full solution found!
            print(f'[+] FLAG: {bytes(test).decode()}')
            exit(0)
        cfail2 = next((c for c in cmps2
                       if c['pc'] == fail['pc']), None)
        pass2 = sum(1 for c in cmps2
                    if c['match'] and c['pc'] != 12)
        if (cfail2 and cfail2['match']
                and pass2 >= current_pass):
            if best_pass is None or pass2 > best_pass:
                found_idx = idx
                found_val = v
                best_pass = pass2
    if found_idx is not None:
        break

if found_idx is None:
    print(f'[-] Stuck at iter {iteration}, CMP pc={fail["pc"]}')
    break

known[found_idx] = found_val
solved.add(found_idx)
partial = bytes(known).decode('latin-1')
print(f'[iter {iteration:2d}] idx={found_idx:2d} -> '
      f"'{chr(found_val)}' ({best_pass} CMPs)  {partial}")
```

7. Solve Trace — Character by Character

The solver recovered all 37 flag characters in 36 iterations. The complete solve trace is shown below:

Iter	Idx	Char	Hex	CMPs Passing	Partial Flag
—	0	S	0x53	3	SDG{
—	1	D	0x44	4	SDG{ (seed)
—	2	G	0x47	5	SDG{
—	3	{	0x7b	6	SDG{
0	4	a	0x61	7	SDG{a
1	5	d	0x64	8	SDG{ad
2	6	3	0x33	9	SDG{ad3
3	7	a	0x61	10	SDG{ad3a
4	8	b	0x62	11	SDG{ad3ab
5	9	2	0x32	13	SDG{ad3ab2
6	10	9	0x39	14	SDG{ad3ab29
7	11	6	0x36	15	SDG{ad3ab296
8	12	8	0x38	16	SDG{ad3ab2968
9	13	f	0x66	17	SDG{ad3ab2968f
10	14	7	0x37	19	SDG{ad3ab2968f7
11	15	7	0x37	20	SDG{ad3ab2968f77
12	16	e	0x65	21	SDG{ad3ab2968f77e
13	17	8	0x38	22	SDG{ad3ab2968f77e8
14	18	e	0x65	23	SDG{ad3ab2968f77e8e
15	19	6	0x36	25	SDG{ad3ab2968f77e8e6
16	20	f	0x66	26	SDG{ad3ab2968f77e8e6f
17	21	3	0x33	27	SDG{ad3ab2968f77e8e6f3
18	22	e	0x65	28	SDG{ad3ab2968f77e8e6f3e
19	23	b	0x62	29	SDG{ad3ab2968f77e8e6f3eb
20	24	c	0x63	31	SDG{ad3ab2968f77e8e6f3ebc
21	25	6	0x36	32	SDG{ad3ab2968f77e8e6f3ebc6
22	26	2	0x32	33	SDG{ad3ab2968f77e8e6f3ebc62
23	27	6	0x36	34	SDG{ad3ab2968f77e8e6f3ebc626
24	28	c	0x63	35	SDG{ad3ab2968f77e8e6f3ebc626c
25	29	c	0x63	37	SDG{ad3ab2968f77e8e6f3ebc626cc
26	30	a	0x61	38	SDG{ad3ab2968f77e8e6f3ebc626cca
27	31	7	0x37	39	SDG{ad3ab2968f77e8e6f3ebc626cca7

28	32	5	0x35	40	SDG{ad3ab2968f77e8e6f3ebc626cca75
29	33	f	0x66	41	SDG{ad3ab2968f77e8e6f3ebc626cca75f
30	34	d	0x64	43	SDG{ad3ab2968f77e8e6f3ebc626cca75fd
35	**35**	**4**	**0x34**	**45**	**SDG{ad3ab2968f77e8e6f3ebc626cca75fd4}**

8. Flag

```
$ ./ghost 'SDG{ad3ab2968f77e8e6f3ebc626cca75fd4}'
Correct!
```

SDG{ad3ab2968f77e8e6f3ebc626cca75fd4}

9. Pitfalls, Dead Ends, and Lessons Learned

Pitfall 1: Misidentifying 0x40 and 0x30

The initial opcode map — derived from bytecode pattern matching rather than handler disassembly — labeled **0x40** as 'MSET' and **0x30** as 'INPUT'. This was backwards. Running a verbose trace revealed that 0x40 reads from argv[1] directly (via the rbx pointer set in main), while 0x30 accesses a static buffer. **Lesson:** always verify opcodes through handler disassembly, not bytecode inference.

Pitfall 2: Treating 0x31 as a dead write

Before identifying the kv_store pattern, the 0x31 handler was modelled as popping and discarding the stack top. This caused 0x30 to always return 0, making every check appear to compare against a static constant. The solver produced **SDG!ad3a...** instead of **SDG{ad3a...** — plausible-looking but wrong. **Lesson:** trace both sides of every store/load pair. If a BSS address is written by one opcode, scan all others for reads to the same address.

Pitfall 3: The length sentinel is not a check failure

The CMP at bytecode pc=12 checks **passphrase[36] == 0** and branches via JNZ. Since passphrase[36] is '}' (0x7d), the comparison fails (flag=0), and JNZ does not fire — execution continues. Early analysis classified this as a failed check and tried to satisfy it, which is wrong. **Lesson:** distinguish 'fail to match expected value' from 'unconditional wrong path'.

Pitfall 4: angr symbolic execution failed silently

Running angr with a find/avoid strategy (find=Correct output, avoid=Wrong output) exhausted all paths without finding a solution for lengths 6–13. The branching structure — multiple JZ/JNZ all targeting the same WRONG block — caused angr's explorer to prune the correct path early. **Lesson:** symbolic execution tools are powerful but not universal. A custom emulator with direct constraint logging is often faster for VM-in-binary challenges.

Pitfall 5: Greedy solver gave wrong chars without the kv fix

Running the greedy solver on the broken emulator produced **idx=3='!'** instead of **'{'**. Both '!' (0x21) and '{' (0x7b) produce 0x21 after transformation — but only when the kv_store is modelled correctly does the XOR-with-kv[0] operation reveal that '{' is required. The solver was self-consistent but wrong. **Lesson:** always verify solver output against the actual binary, not just the emulator.

10. Tools and Environment

Tool	Version / Notes	Used For
Python 3.12	Standard library only	Emulator, solver, bytecode analysis
capstone	pip install capstone	Disassembly of opcode handlers and main()
angr	pip install angr	Attempted symbolic execution (insufficient)
readelf / file	binutils	ELF section mapping, segment analysis
Custom VM emulator	~80 lines Python	Full VM emulation with CMP logging
Sensitivity brute forcer	~60 lines Python	Per-character constraint solving
reportlab	PDF generation	This writeup

11. Summary

The ghost challenge is a well-constructed VM reversing challenge with a clean pedagogical structure. The difficulty lies not in the complexity of any individual operation — the arithmetic is basic — but in correctly identifying all of the VM's mechanisms, particularly the non-obvious kv_store feedback loop that chains character checks together.

The solve path in order of discovery was: ELF recon → main() disassembly → jump table extraction → handler disassembly for all 19 opcodes → recognition of the 0x30/0x31 kv_store pair → correct emulator → sensitivity-based brute force solver → flag in 36 iterations.

Total time from first look to flag: approximately one extended session of iterative emulator refinement. The kv_store identification was the singular breakthrough that unlocked the solution.

Writeup by **ssstrickys** · March 31, 2026 · Flag: `SDG{ad3ab2968f77e8e6f3ebc626cca75fd4}`