I treated this one like a “side-channel flavored” SPN cipher that *pretends* to be a noisy oracle at runtime, but still ships a fully deterministic transform inside the binary via lookup tables. The clean way to solve it is to extract those tables and invert the transform offline.

Target output:

`0x4C494D494E414C21`

Final answer:

`0xfun{0x4c8e40be1e97f544}`

Quick Files

- Binary: `Liminal`
- Offline solver: `solve.py`

Run:

```
python3 ./solve.py
```

1) Recon: What Input Format Does It Want?

If you run the binary without arguments, it prints usage text:

```
./Liminal
```

It expects a single hex integer like `0xDEADBEEFCAFEBABE`. So the “input” we’re searching for is one 64-bit value.

```
if ( n2 != 2 )
{
  __fprintf_chk(stderr, 2, "Usage: %s <hex_input>\n", *a2);
  __fprintf_chk(stderr, 2, "Example: %s 0xDEADBEEFCAFEBABE\n", *a2);
  return 1;
}
v4 = strtoull(a2[1], &endptr, 16);
if ( *endptr && *endptr != 10 )
{
  fwrite("Error: Invalid hex input\n", 1u, 0x19u, stderr);
  return 1;
}
n3 = 3;
__printf_chk(2, "Calibrating...");
fflush(stdout);
while ( !(unsigned int)sub_406298() )
{
  if ( !--n3 )
  {
    puts("FAILED");
    fwrite("Your CPU does not support the required features.\n", 1u, 0x31u, stderr);
    fwrite("This binary requires speculative execution side-channels.\n", 1u, 0x3Au, stderr);
    return 1;
  }
```

2) Why You Should Not Trust The Runtime Oracle

In the output, the program prints a “confidence” counter:

`compute(0x%016lx) = 0x%016lx (confidence: %lu/%d)`

When you look at the disassembly, you’ll see the classic timing/side-channel primitives:

- `clflush`
- `mfence`
- `rdtsc/rdtscp`

That is the “whispering through silicon shadows” part. On many setups (VMs, WSL, mitigations, scheduling noise), the timing path is unstable, so calling `compute(x)` repeatedly can produce different results and never converge beyond `confidence: 1/50`.

So I did not brute-force against the running program. I extracted the deterministic transform from the embedded tables and inverted it offline.

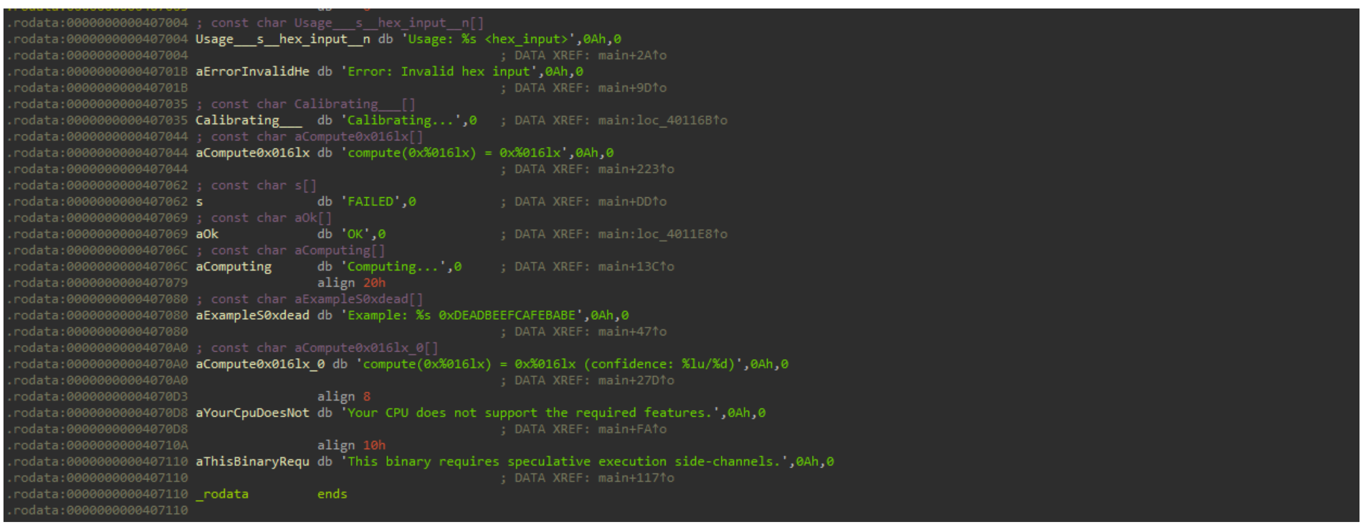
Place screenshot here (show rdtscp/clflush block):

“side_channel_primitives.png” could not be found.

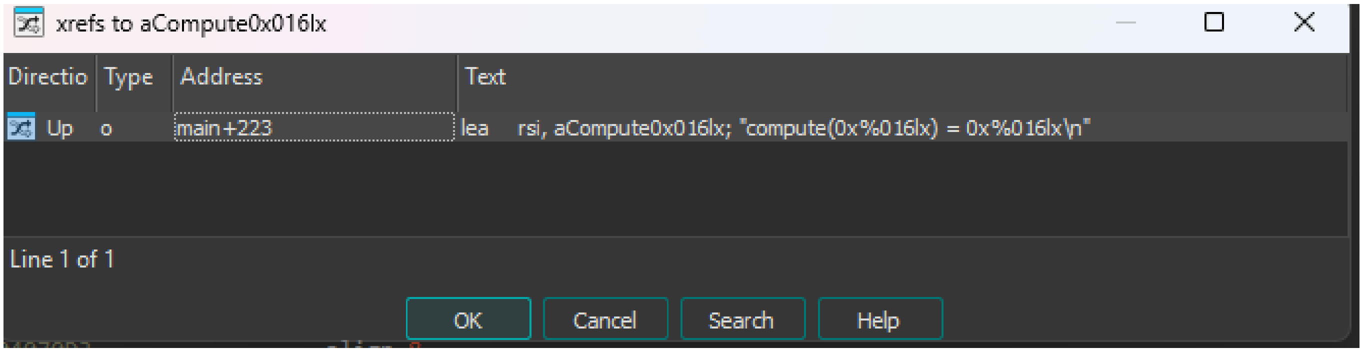
3) Find The Real Compute Routine In IDA

I started from Strings and Xrefs:

1. Open `Strings` and search for `compute(0x%016lx)`.
2. Jump to its Xrefs to find the print site (main logic).



```
.rodata:0000000000407004 ; const char Usage___s__hex_input__n[]
.rodata:0000000000407004 Usage___s__hex_input__n db 'Usage: %s <hex_input>',0Ah,0
.rodata:0000000000407004                                         ; DATA XREF: main+2A↑o
.rodata:000000000040701B aErrorInvalidHe db 'Error: Invalid hex input',0Ah,0
.rodata:000000000040701B                                         ; DATA XREF: main+9D↑o
.rodata:0000000000407035 ; const char Calibrating___[]
.rodata:0000000000407035 Calibrating___  db 'Calibrating...',0   ; DATA XREF: main:loc_40116B↑o
.rodata:0000000000407044 ; const char aCompute0x016lx[]
.rodata:0000000000407044 aCompute0x016lx db 'compute(0x%016lx) = 0x%016lx',0Ah,0
.rodata:0000000000407044                                         ; DATA XREF: main+223↑o
.rodata:0000000000407062 ; const char s[]
.rodata:0000000000407062 s               db 'FAILED',0           ; DATA XREF: main+DD↑o
.rodata:0000000000407069 ; const char aOk[]
.rodata:0000000000407069 aOk             db 'OK',0               ; DATA XREF: main:loc_4011E8↑o
.rodata:000000000040706C ; const char aComputing[]
.rodata:000000000040706C aComputing      db 'Computing...',0     ; DATA XREF: main+13C↑o
.rodata:0000000000407079                 align 20h
.rodata:0000000000407080 ; const char aExampleS0xdead[]
.rodata:0000000000407080 aExampleS0xdead db 'Example: %s 0xDEADBEEFCAFEBABE',0Ah,0
.rodata:0000000000407080                                         ; DATA XREF: main+47↑o
.rodata:00000000004070A0 ; const char aCompute0x016lx_0[]
.rodata:00000000004070A0 aCompute0x016lx_0 db 'compute(0x%016lx) = 0x%016lx (confidence: %lu/%d)',0Ah,0
.rodata:00000000004070A0                                         ; DATA XREF: main+27D↑o
.rodata:00000000004070D3                 align 8
.rodata:00000000004070D8 aYourCpuDoesNot db 'Your CPU does not support the required features.',0Ah,0
.rodata:00000000004070D8                                         ; DATA XREF: main+FA↑o
.rodata:000000000040710A                 align 10h
.rodata:0000000000407110 aThisBinaryRequ db 'This binary requires speculative execution side-channels.',0Ah,0
.rodata:0000000000407110                                         ; DATA XREF: main+117↑o
.rodata:0000000000407110 _rodata         ends
.rodata:0000000000407110
```




```
    }
    while ( v10 != v8 );
    if ( n2_1 > 2 )
    {
      __printf_chk(2, "compute(0x%016lx) = 0x%016lx\n", v4, v9);
      return 0;
    }
    ++n3_1;
    v8 = v10 + 8;
  }
```

From there, the interesting function ends up around this address range:

- core routine: `0x405b37`

In IDA you can `G` to `0x405B37` and inspect it in graph view.

```
; Attributes: bp-based frame

; __int64 __fastcall sub_405B37(__int64)
sub_405B37 proc near
push    rbp
mov     rbp, rsp
push    rbx
push    r12
push    r13
push    r14
push    r15
sub     rsp, 20h
mov     r14, rdi
lea     r15, qword_42F2C0
mov     rax, [r15]
xor     r14, rax
xor     r13, r13
mov     rdi, r14
shr     rdi, 0
and     rdi, 0FFh
call    sub_4056A1
shl     rax, 0
or      r13, rax
mov     rdi, r14
shr     rdi, 8
and     rdi, 0FFh
call    sub_40572C
shl     rax, 8
or      r13, rax
mov     rdi, r14
shr     rdi, 10h
and     rdi, 0FFh
call    sub_4057B7
shl     rax, 10h
or      r13, rax
mov     rdi, r14
shr     rdi, 18h
and     rdi, 0FFh
call    sub_405842
shl     rax, 18h
or      r13, rax
mov     rdi, r14
shr     rdi, 20h
and     rdi, 0FFh
call    sub_4058CD
shl     rax, 20h
```

What matters is the structure, not the exact decompiler output. The function implements an SPN-like 64-bit transform:

1. XOR a 64-bit round key.
2. Apply a byte-wise substitution layer (8 independent 8-bit S-boxes, one per byte position).
3. Apply a fixed bit-permutation across the full 64-bit state (between rounds).

It does 8 rounds total and skips the bit permutation on the final round.

4) The Key Insight: All Secrets Are In .data

Even though the binary uses a side-channel to *evaluate* bits at runtime, it still needs constant tables to describe what it wants. Those are all in `.data`.

You can confirm sections with:

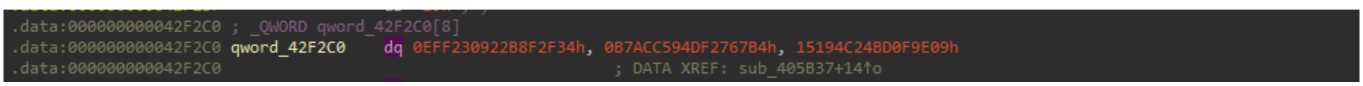
```
readelf -S ./Liminal
```

4.1 Round Keys (8 qwords)

There is an 8-qword key schedule at:

- `0x42F2C0`

In IDA, jump to it (`G` -> `0x42F2C0`) and define it as `qword[8]` if needed.



4.2 Bit Permutation Table (64 bytes)

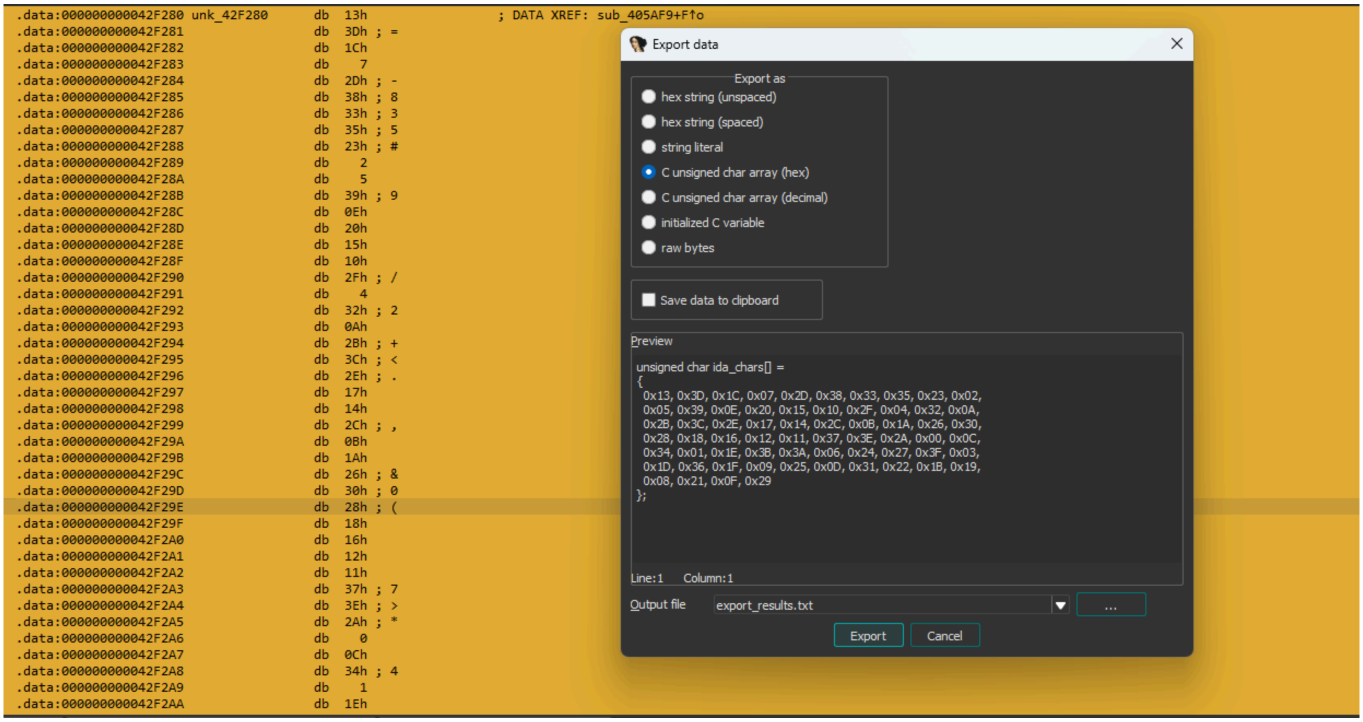
There is a 64-byte permutation table at:

- `0x42F280`

In IDA, jump to it (`G` -> `0x42F280`) and define it as `byte[64]` .

Interpretation:

- `perm[out_bit] = in_bit`
- forward: `out[out_bit] = in[in_bit]`
- inverse: build `inv_perm[in_bit] = out_bit`



4.3 S-boxes Are Stored As “Bit Tables”

This is the part that looks weird until you write down the layout.

There are 8 different S-boxes, one per byte position in the 64-bit word. Each S-box is not stored as a clean `sbox[256]`. Instead it’s stored as 8 separate “bit tables” (one per output bit).

Start address for the whole structure:

- `0x40F280`

The layout is:

- byte position `i` in 0..7 has a block of size `0x4000`
- inside that block, output bit `b` in 0..7 has a table of size `0x800`
- that table contains 256 qwords

Addressing:

- `base_i = 0x40F280 + i*0x4000`
- `table(i, b)[x]` is the qword at `base_i + b*0x800 + x*8`

Each qword is basically a boolean, but encoded as one of two values:

- `0x100`
- `0x340`

To reconstruct the actual 8-bit S-box output for input `x`:

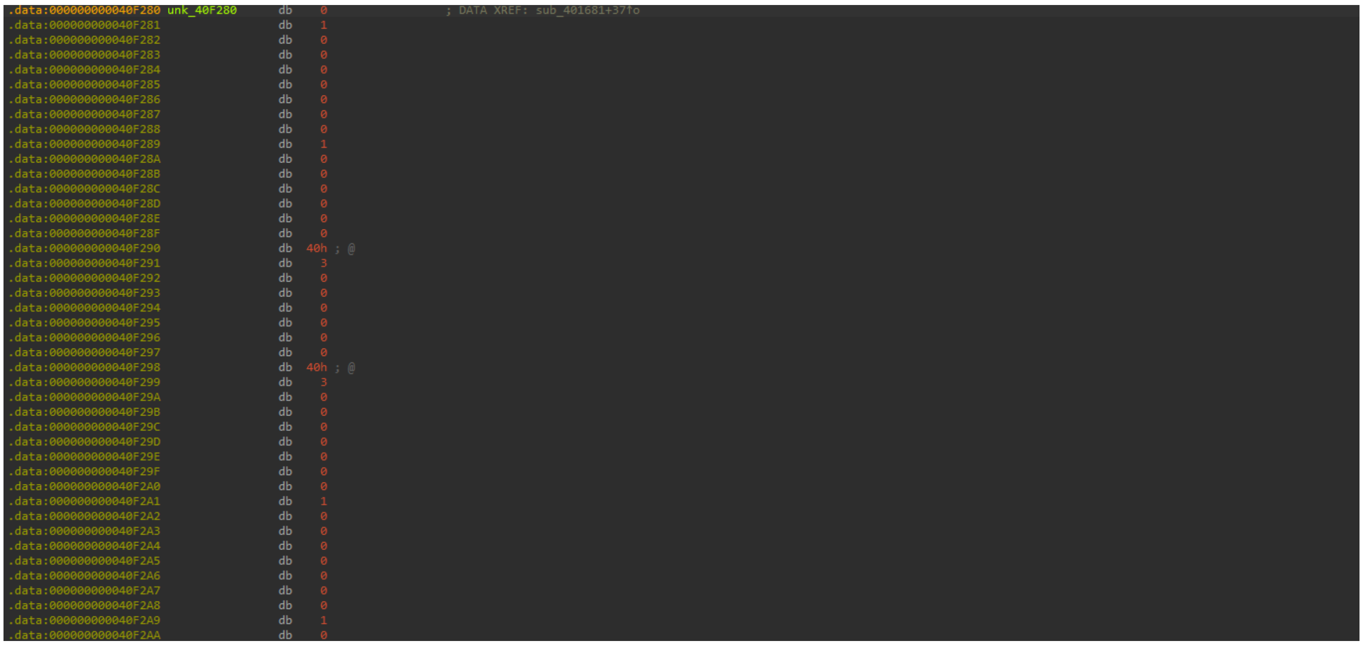
- output bit `b` is 1 if `table(i, b)[x] == 0x340`
- output bit `b` is 0 if `table(i, b)[x] == 0x100`

So:

- `sbox_i[x] = sum(((table(i, b)[x] == 0x340) << b) for b in 0..7)`

Do this for all `x` in 0..255 and you have the full S-box for that byte position. Then invert it by swapping the mapping.

Place screenshot here (show the repeating 0x100/0x340 pattern):



5) Reconstruct The Deterministic Transform

Once you have:

- `keys[0..7]`
- `sbox_i` for i = 0..7 and their inverses
- `perm[64]` and `inv_perm[64]`

You can model the intended `compute(x)` deterministically as:

For rounds `r = 0..7`:

- `state ^= keys[r]`
- `state = SubBytes(state)` where each byte uses its own S-box
- if `r != 7`: `state = PermuteBits(state)`

SubBytes is byte-position dependent:

- byte 0 uses `sbox_0`, byte 1 uses `sbox_1`, ... byte 7 uses `sbox_7`

Permutation is bit-level using `perm[out_bit] = in_bit`.

6) Invert It To Solve The Challenge

We want:

`compute(input) = TARGET`

So invert the rounds in reverse order:

1. Undo final round (no permutation in that round):
   - `state = InvSubBytes(TARGET)`
   - `state ^= keys[7]`
2. For rounds `r = 6 down to 0`:
   - `state = InvPermuteBits(state)`
   - `state = InvSubBytes(state)`
   - `state ^= keys[r]`

That final `state` is the required input.

For this target (`0x4C494D494E414C21`), the result is:

`0x4c8e40be1e97f544`

7) The Offline Solver Script

I wrote `solve.py` to do the full extraction + inversion directly from the ELF file.

It:

1. Reads `.data` by virtual address (the script uses the `.data` base vaddr and file offset).
2. Loads `perm` at `0x42F280` and builds `inv_perm`.
3. Loads `keys` at `0x42F2C0`.
4. Reconstructs 8 S-boxes from the bit tables starting at `0x40F280`.
5. Inverts the SPN to recover the preimage for the target.

```
import struct

BIN_PATH = "./Liminal"

DATA_VADDR_BASE = 0x40A000
DATA_FILE_OFF_BASE = 0x9000
```

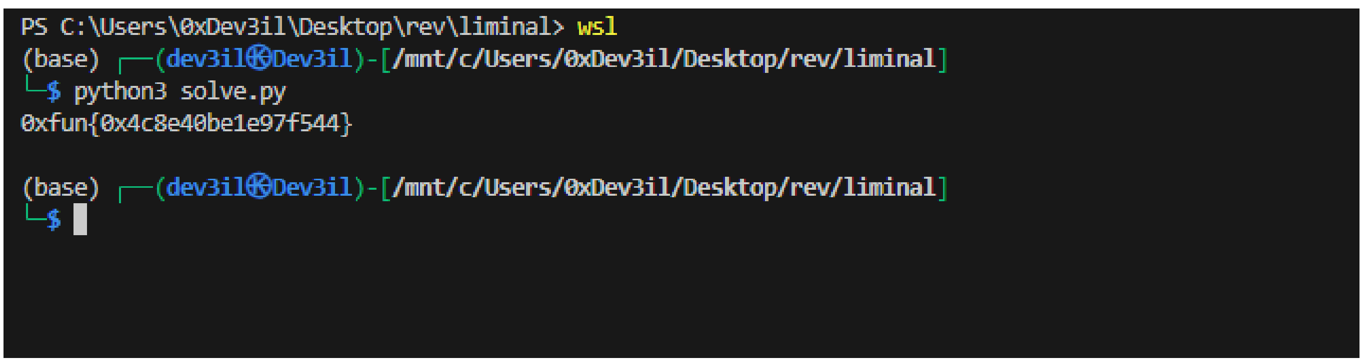
```python
TABLE0_VADDR = 0x40F280
PERM_VADDR = 0x42F280
KEYS_VADDR = 0x42F2C0


TARGET = 0x4C494D494E414C21


def _read_at_vaddr(vaddr, nbytes):
    with open(BIN_PATH, "rb") as f:
        f.seek(DATA_FILE_OFF_BASE + (vaddr - DATA_VADDR_BASE))
        return f.read(nbytes)


def _u8(vaddr, n):
    return list(_read_at_vaddr(vaddr, n))



def _u64(vaddr, n):
    return list(struct.unpack("<%dQ" % n, _read_at_vaddr(vaddr, 8 * n)))

def _build():
    perm = _u8(PERM_VADDR, 64)
    inv_perm = [0] * 64
    for out_bit, in_bit in enumerate(perm):
        inv_perm[in_bit] = out_bit

    keys = _u64(KEYS_VADDR, 8)

    sboxes = []
    inv_sboxes = []
    for byte_pos in range(8):
        base = TABLE0_VADDR + byte_pos * 0x4000
        bit_tables = [_u64(base + b * 0x800, 256) for b in range(8)]
        sb = [0] * 256
        for x in range(256):
            y = 0
            for b in range(8):
                if bit_tables[b][x] == 0x340:
                    y |= 1 << b
            sb[x] = y
        inv = [0] * 256
        for x, y in enumerate(sb):
            inv[y] = x
        sboxes.append(sb)
```

```python
        inv_sboxes.append(inv)

    return perm, inv_perm, keys, sboxes, inv_sboxes




def _sub_bytes(state, sboxes):
    out = 0
    for i in range(8):
        b = (state >> (8 * i)) & 0xFF
        out |= sboxes[i][b] << (8 * i)
    return out



def _inv_sub_bytes(state, inv_sboxes):
    out = 0
    for i in range(8):
        b = (state >> (8 * i)) & 0xFF
        out |= inv_sboxes[i][b] << (8 * i)
    return out




def _permute_bits(state, perm):
    out = 0
    for out_bit, in_bit in enumerate(perm):
        out |= ((state >> in_bit) & 1) << out_bit
    return out



def _inv_permute_bits(state, inv_perm):
    out = 0
    for in_bit, out_bit in enumerate(inv_perm):
        out |= ((state >> out_bit) & 1) << in_bit
    return out


def invert_target(target):
    perm, inv_perm, keys, sboxes, inv_sboxes = _build()
    s = target
    s = _inv_sub_bytes(s, inv_sboxes)
    s ^= keys[7]
```

```python
    for r in range(6, -1, -1):
        s = _inv_permute_bits(s, inv_perm)
        s = _inv_sub_bytes(s, inv_sboxes)
        s ^= keys[r]
    return s


def main():
    x = invert_target(TARGET)
    print(f"0xfun{{0x{x:016x}}}")



if __name__ == "__main__":
    main()
```

```
python3 solve.py
```

Place screenshot here:



Expected output:

`0xfun{0x4c8e40be1e97f544}`