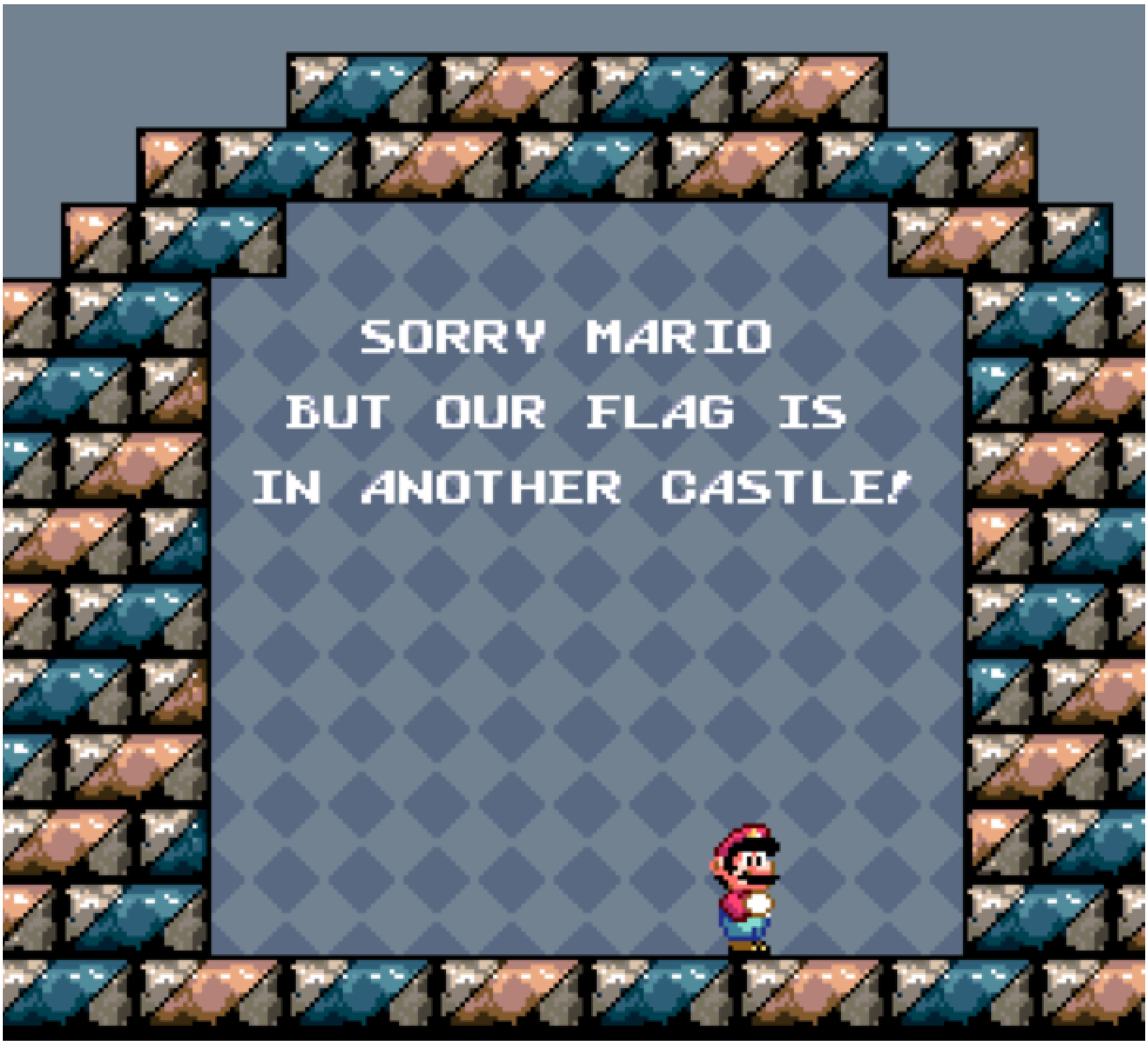
Snake Part 2

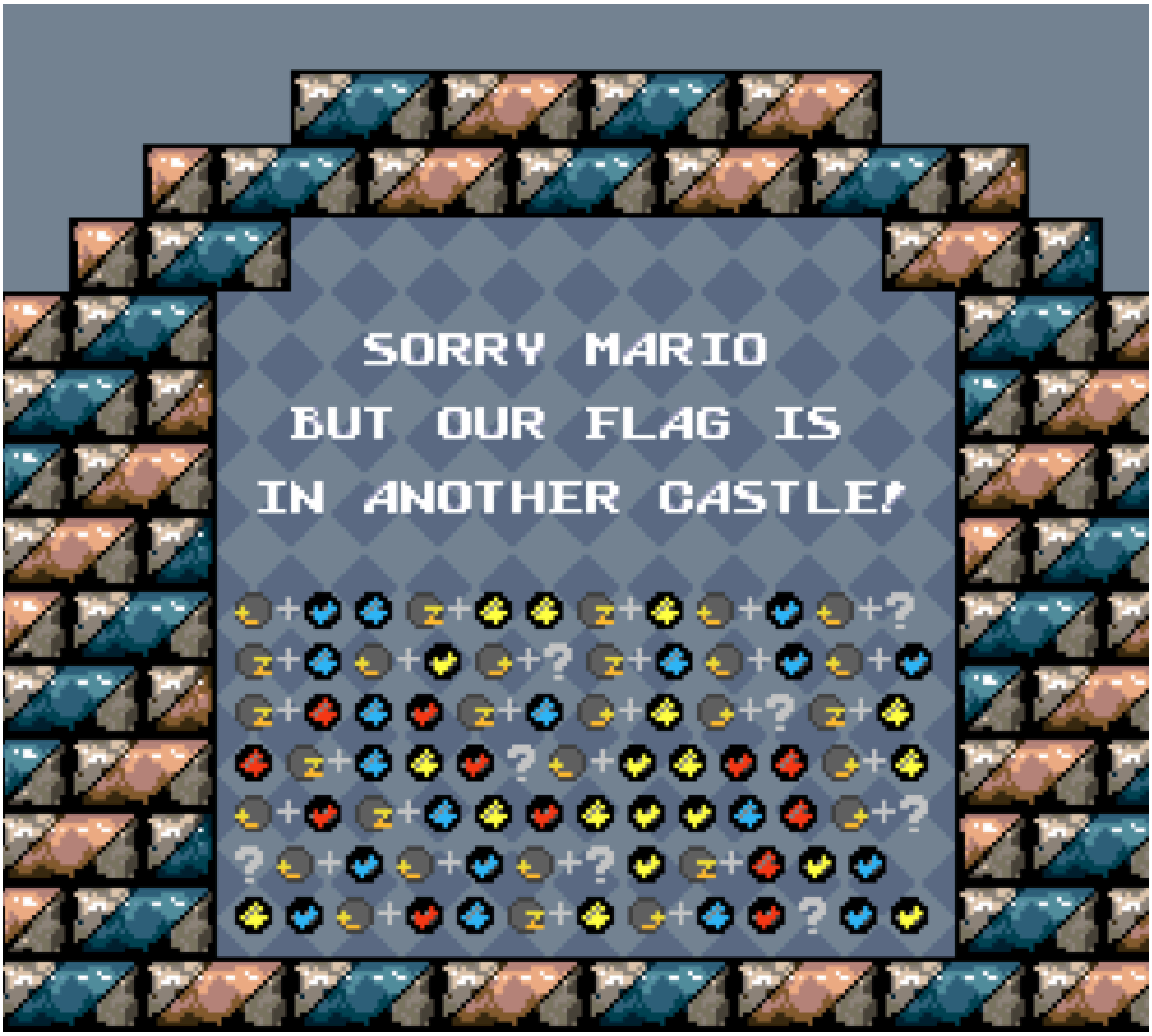
This level is not about snakes and apples.

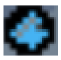
Pick up some apples then enter the door that shows up.



Press the Konami code, i.e. **Up Up Down Down Left Right Left Right B A Start**

This plays a sound effect and it now becomes this:



These are street fighter moves. The  means a single move. You can also make combos such as Shoriyoken and Hadouken . The  indicates unknown and you will need to try each of the three alternative (preferably in an automated fashion). Also you don't know which color corresponds to each of the strength types (light, medium, hard) so you need to check for all those 6 permutations.

By looking in the disasm you can see that each sound effect (corresponding to each move or combo) that gets sent to the APU (audio chip) is stored in array.

After receiving 58 combos, CRC64_reversed is executed on this array, and the two lowest bytes are compared to 0x4d9c. If this matches, then it uses RC4 to decrypt a block of machine code and jumps to it. This plays a sound effect (18) then displays:

Then you can see that the code now looks for another 6 moves/combos and stores those in the array and sends them to the APU for playback, similarly comparing the CRC64_rev against 0x4d9c. Bruteforce the 15**6 = 11M CRC values which result in 0x4d9c. Then try to decrypt the code blob and just do some statistical analysis (such as index of coincidence) to see which RC4 key resulted in non randomness. This is then the valid code.

Interestingly, it's exact the same code blob that got decrypted as before. But it's polyglot in the sense that two different keys will make it decrypt into two different valid machine code sequences.

The game now displays this

And this is a clear hint that you no longer need to worry about 65816. Instead, focus should be on the sound effects that have been sent to the APU. The code now seems to compare against an impossibly long sequence of joypad inputs (and all bytes in the encrypted code have been decrypted) indicating that we reached a dead end here.

Focus now goes to disassembling the APU code. From the first look, it seems like it just plays each sound effect it receives from the main SNES code.

```
ROM:0865 LOOP:                                ; CODE XREF: ROM:0869↓j
ROM:0865                                      ; ROM:086F↓j ...
ROM:0865                 mov     a, DSP_PORT0
ROM:0867                 cmp     a, last_cmd
ROM:0869                 beq     LOOP
ROM:086B                 mov     last_cmd, a
ROM:086D                 cmp     a, #0
ROM:086F                 beq     LOOP
ROM:0871                 dec     a
ROM:0872                 mov     dsp_addr, #4
ROM:0875                 mov     dsp_data, a
ROM:0878                 mov     dsp_addr, #3
ROM:087B                 mov     dsp_data, #4
ROM:087E                 mov     dsp_addr, #2
ROM:0881                 mov     dsp_data, #0
ROM:0884                 mov     dsp_addr, #$4C ; 'L'
ROM:0887                 mov     dsp_data, #1
ROM:088A                 bra     LOOP
```

However, if you notice the earlier write:

```
ROM:080C                 mov     a, #$5D ; ']'
ROM:080E                 mov     $7AB+x, a
```

You will notice that the jump at 0x88A got overwritten and it instead jumps to a secret location which looks silly but was disguised in that way to make it less obvious that it’s code

```
ROM:08F0                 adc     a, cksum+$E
ROM:08F2                 nop
ROM:08F3                 dec     x
ROM:08F4                 dec     y
ROM:08F5                 dec     x
ROM:08F6                 dec     y
ROM:08F7                 mov     cksum+$E, a
ROM:08F9                 adc     a, cksum+$D
ROM:08FB                 nop
ROM:08FC                 inc     y
ROM:08FD                 notc
ROM:08FE                 inc     y
ROM:08FF                 notc
ROM:0900                 mov     cksum+$D, a
ROM:0902                 adc     a, cksum+$C
ROM:0904                 nop
ROM:0905                 inc     x
ROM:0906                 clrp
ROM:0907                 inc     x
ROM:0908                 clrp
ROM:0909                 mov     cksum+$C, a
ROM:090B                 adc     a, cksum+$B
ROM:090D                 nop
ROM:090E                 inc     y
ROM:090F                 notc
ROM:0910                 inc     y
ROM:0911                 notc
ROM:0912                 mov     cksum+$B, a
ROM:0914                 adc     a, cksum+$A
ROM:0916                 nop
ROM:0917                 notc
ROM:0918                 dec     x
ROM:0919                 notc
ROM:091A                 dec     x
ROM:091B                 mov     cksum+$A, a
ROM:091D                 adc     a, cksum+9
ROM:091F                 nop
ROM:0920                 dec     x
ROM:0921                 dec     y
ROM:0922                 dec     x
ROM:0923                 dec     y
ROM:0924                 mov     cksum+9, a
ROM:0926                 adc     a, cksum+8
```

```
ROM:096C                 mov     cksum+1, a
ROM:096E                 adc     a, cksum
ROM:0970                 nop
ROM:0971                 ei
ROM:0972                 clrp
ROM:0973                 ei
ROM:0974                 clrp
ROM:0975                 mov     cksum, a
ROM:0977                 cmp     a, #$43 ; 'C'
ROM:0979                 nop
ROM:097A                 clrp
ROM:097B                 clrc
ROM:097C                 clrp
ROM:097D                 clrc
ROM:097E                 bne     loc_9A2
ROM:0980                 mov     a, cksum+1
ROM:0982                 nop
ROM:0983                 clrc
ROM:0984                 dec     y
ROM:0985                 clrc
ROM:0986                 dec     y
ROM:0987                 cmp     a, #$A2
ROM:0989                 bne     loc_9A2
ROM:098B                 nop
ROM:098C                 dec     y
ROM:098D                 push    a
ROM:098E                 dec     y
ROM:098F                 push    a
ROM:0990                 mov     a, cksum+2
ROM:0992                 eor     a, #$65 ; 'e'
ROM:0994                 nop
ROM:0995                 push    a
ROM:0996                 nop
ROM:0997                 push    a
ROM:0998                 nop
ROM:0999                 mov     a, cksum+3
ROM:099B                 eor     a, #2
ROM:099D                 nop
ROM:099E                 push    a
ROM:099F                 nop
ROM:09A0                 ret
```

This code computes a checksum of each sound effect that was played (with the carry from the first ADC carried over to the next)

```python
def sounds_to_key(sounds):
  sums = bytearray([0]*16)
  for value in sounds:
    for i in range(14, -1, -1):
      value = sums[i] + (value & 0xff) + (value >> 8)
      sums[i] = value & 0xff
  return list(sums)
```

The code then compares two bytes of the checksum against a static value, then jumps to the address in the next two bytes. This could be anywhere in the memory. But realistically there are not too many options. You can exclude all code that appear to be valid BRR blocks (the file format used for audio samples).

You will notice that slightly after the last BRR block (about 57 bytes after) you'll see cleartext code that can be cleaned up in this (note all the opaque predicates, i.e. always taken branches, to annoy the reverser) :

```
ROM:A00A always_taken0:                                ; CODE XREF: ROM:A018↓j
ROM:A00A                 inc     a
ROM:A00B                 asl     a
ROM:A00C                 bne     always_taken1
ROM:A00C ; ---------------------------------------------------------------
ROM:A00E                 .db $5F ; _
ROM:A00F ; ---------------------------------------------------------------
ROM:A00F                 or      a, byte_1733
ROM:A012                 mov     a, #$10
ROM:A014                 cmp     x, byte_25
ROM:A016                 xcn
ROM:A017                 push    psw
ROM:A018                 bne     always_taken0
ROM:A018 ; ---------------------------------------------------------------
ROM:A01A                 .db $9A
ROM:A01B ; ---------------------------------------------------------------
ROM:A01B
ROM:A01B always_taken1:                                ; CODE XREF: ROM:A00C↑j
ROM:A01B                 mov     y, #$5C ; '\'
ROM:A01D                 pop     x
ROM:A01E                 mov     x, a
ROM:A01F                 bpl     always_taken2
ROM:A01F ; ---------------------------------------------------------------
ROM:A021                 .db $1E
ROM:A022 ; ---------------------------------------------------------------
ROM:A022
ROM:A022 loop3:                                        ; CODE XREF: ROM:A05A↓j
ROM:A022                 inc     ADDRH
ROM:A025                 dec     x
ROM:A026                 bpl     always_taken
ROM:A026 ; ---------------------------------------------------------------
ROM:A028                 .db $3F ; ?
ROM:A029 ; ---------------------------------------------------------------
ROM:A029
ROM:A029 always_taken:                                 ; CODE XREF: ROM:A026↑j
ROM:A029                 dec     x
ROM:A02A                 bne     loop2_cont
ROM:A02C                 mov     a, x
ROM:A02D                 asl     a
ROM:A02E                 beq     encr1_start
ROM:A02E ; ---------------------------------------------------------------
ROM:A030                 .db $DB
ROM:A031 ; ---------------------------------------------------------------
```

```
ROM:A031 always_taken2:                                ; CODE XREF: ROM:A01F↑j
ROM:A031                 adc     a, #$51 ; 'Q'
ROM:A033                 bcc     always_taken3
ROM:A033 ; ---------------------------------------------------------------
ROM:A035                 .db $CA
ROM:A036 ; ---------------------------------------------------------------
ROM:A036
ROM:A036 loop2:                                        ; CODE XREF: ROM:A057↓j
ROM:A036                 bra     loop2_cont
ROM:A036 ; ---------------------------------------------------------------
ROM:A038                 .db $CC
ROM:A039 ; ---------------------------------------------------------------
ROM:A039
ROM:A039 partial:                                      ; CODE XREF: ROM:A051↓p
ROM:A039                 adc     a, #172
ROM:A03B                 notc
ROM:A03C                 dec     byte_34
ROM:A03E                 mov     (ADDRL)+y, a
ROM:A040                 inc     byte_35
ROM:A042                 xcn
ROM:A043                 ret
ROM:A043 ; ---------------------------------------------------------------
ROM:A044                 .db $EA
ROM:A045 ; ---------------------------------------------------------------
ROM:A045
ROM:A045 always_taken3:                                ; CODE XREF: ROM:A033↑j
ROM:A045                 mov     ADDRH, #$A0
ROM:A048                 push    y
ROM:A049                 eor     a, cksum+4
ROM:A04C                 xcn
ROM:A04D
ROM:A04D loop2_cont:                                   ; CODE XREF: ROM:A02A↑j
ROM:A04D                                               ; ROM:loop2↑j
ROM:A04D                 pop     y
ROM:A04E                 xcn
ROM:A04F                 eor     a, (ADDRL)+y
ROM:A051                 call    partial
ROM:A054                 inc     y
ROM:A055                 push    y
ROM:A056                 notc
ROM:A057                 bne     loop2
ROM:A059                 inc     x
ROM:A05A                 bne     loop3
```

The appropriate starting position seems to be either 0xa00f (which is just a junk instruction that we can ignore) or 0xa012.

Deobfuscating the code gives us:

```
dec0_start:
  mov a, #16
  xcn            ; i.e. swap nibbles, a=1
  inc a          ; a = 2
  asl a          ; a = 4
  mov y, #$5c    ; low byte
  mov x, a       ; x = 4
  adc a, #$51    ; a = 0x55, carry = 0
  mov ADDRH, $a0 ; ADDR = 0xa000
  eor a, CKSUM+4 ; a ^= CKSUM[4]
enc0:
  eor a, (ADDRL)+y  ; decrypts code at 0xa05c
  adc a, #172
  mov (ADDRL)+y, a
  inc y
  bne enc0          ; loop until y=0
  inc (ADDRH)
  dec x
  bne enc0          ; loop another 1024 bytes
```

Now we don't know the correct value for CKSUM[4], but it's only a byte so there's only 256 combinations to try, in order to get code that looks valid. Once you find the right byte value, you'll get another decryption loop that looks like:

```
enc1:
  mov.b a, CKSUM+5
  mov ADDRH,#enc2>>8
  mov y, #enc2&$ff
  mov x, #4
-
  adc a, (ADDRL)+y
  push a : asl a : pop a : rol a
  mov (ADDRL)+y, a
  inc y
  bne -
  inc (ADDRH)
  dec x
  bne -
  mov ADDRH,#enc3>>8
  mov y, #enc3&$ff
  mov x, #4
  mov.b a, KEY+2
notc2: clrc
```

Again, you need to try all the combinations for CKSUM[5]. This continues in the same way with various decryption loops all the way up to CKSUM[10].

So in summary, we know know these values for CKSUM:

0..1	0x43,0xa2 - statically compared
2	0xc5 - high byte of jump target xored by 0x65 (0x65 ^ 0xa0 = 0xc5)
3	0x0d - low byte of jump target xored with 2 **or** 0x14 - low byte of jump target xored with 2
4..10	0x7c, 0x54, 0x77, 0x1b, 0x0d, 0x0d, 0dd5 - from reversing encryption loop 0..6
11..14	4 unknown bytes
15	0x00 (Always zero, never written to)

The cleaned up final decrypted SPC code looks like this.

```
   %SpcWrite(!V0SRCN, $00)
  mov a,#sinewave&$ff
  mov $0600,a :  mov $0602,a
  mov a,#sinewave>>8
  mov $0601,a :  mov $0603,a            ; Loads a sine wave into the sound table

  ; Initialize rc4
  mov x,#0
- mov a,x
  mov Rc4+x,a
  inc x : bne -
  mov y,#0
- mov.b R0,y
  mov a,x : and a,#15 : mov y,a
  mov a,cksum+y                  ; RC4 key is cksum
  clrc : adc a,Rc4+x
  clrc : adc.b a,R0
  mov y,a
  mov a,Rc4+x
  push a :
  mov a,Rc4+y
  mov Rc4+x,a
  pop a
  mov Rc4+y,a
  inc x
  bne -
  mov.b Rc4_i,x
```

```
  mov.b Rc4_j,x
  mov.b BITDATA,x

loop:
- mov y, !DSP_COUNTER0 : beq -
- mov y, !DSP_COUNTER0 : beq -
- mov y, !DSP_COUNTER0 : beq -
  clrv

;; Get next bit
  lsr.b BITDATA
  bne +
  ; Get next RC4 byte of keystream
  ; i = (i + 1) mod 256
  inc.b Rc4_i
  mov.b x, Rc4_i
  ; j = (j + S[i]) mod 256
  mov a, Rc4+x
  push a
  clrc : adc.b a, Rc4_j
  mov.b Rc4_j, a
  mov y, a
  ; swap S[i], S[j]
  mov a, Rc4+y
  mov Rc4+x, a
  pop a
  mov Rc4+y,a
  clrc : adc a, Rc4+x
  mov x, a
  mov a, Rc4+x

  ; Decrypt it with the next byte of morse code
  mov.b x, BITPTR : inc BITPTR
  eor a, morse_code+x

  ; Play silence after morse code
  cmp x, #morse_code_end-morse_code
  bne morse_code_not_done
  dec.b BITPTR
  mov a, #0
morse_code_not_done:
  setc
  ror a
  mov.b BITDATA, a
+
```

```
;; Convert bit to 1 or 0, when bit changes, either play or stop sinewave.
  mov a, #0 : adc a, #0
  cmp.b a, LASTBIT
  beq loop
  mov.b LASTBIT,a
  cmp a,#0
  beq +
  %SpcWrite(!KOF, $0)
  %SpcWrite(!KON, $1)
  bra loop
+ %SpcWrite(!KOF, $1)
  bra loop
morse_code:
  db .. ; encrypted morse blob
```

So now we need to decrypt the morse code. There's two approaches. (CKSUM[11..14] are unknown and CKSUM[2] have more than one candidate)

- The intended solution was to brute force the RC4, so about 33 bits need to be bruteforced, with the caveat that you need to realize that CKSUM[2] has more than one possible value. You know you found the value when the morse code shows clear signs of non randomness (use for example a high value of index of coincidence as a statistical measure of non-randomness). This approach doesn't require you to find actual joypad inputs.
- An alternative solution is to find further joypad inputs that result in the same values for CKSUM using the simple additive checksum. You don't know the number of moves up-front, but you need 11 further inputs, i.e. around 44 bits. Could be solved with brute force or possibly with Z3. Then decrypt the morse with the found candidates.

The whole list of required sound effects to be played that's passed to the checksummer (I did minus 1 already cause the code does that before checksumming):

```
16                                ; Round 1 sound effect

10, 7, 2, 8, 2, 10, 11, 1, 11, 6, 1, 10, 10, 3, 7, 15, 1, 5, 5, 2, 9, 1, 8, 15, 15, 11,
8, 15, 9, 5, 12, 1, 8, 15, 8, 14, 14, 7, 9, 5, 13, 10, 10, 12, 14, 3, 14, 13, 8, 13,
12, 7, 2, 4, 15, 8, 13, 14       ; Initial moves from the image

17                                ; Round 2 sound effect

5, 6, 14, 10, 9, 4                ; 6 unknown moves (can be bruted in 65816 code)

18                                ; Round 3 sound effect

9, 11, 13, 4, 1, 2, 14, 2, 3, 12, 3   ; Can be bruted or found with z3
```

The final morse code is:

```
db $d7, $d1, $d1, $15, $77, $74, $74, $01, $d0, $15, $51, $17, $07, $c0,
$55, $74, $d1, $71, $5d, $04, $40, $55, $5c, $54, $77, $54, $05, $c0,
$1d, $d5, $1d, $d5, $1d, $47, $55, $00, $dc, $71, $77, $77, $74, $51,
$15, $d5, $1d, $00, $dd, $d1, $dd, $1d, $47, $15, $00, $55, $71, $77,
$77, $dc, $51, $dd, $01, $50, $dd, $71, $71, $5d, $74, $71, $dd, $d1,
$5d, $1c, $dd, $dd, $71, $77, $77, $5c, $00, $dc, $dd, $1d, $17, $00,
$c7, $dd, $dd, $d1, $5d, $00, $74, $51, $dc, $45, $15, $07, $c0, $55,
$74, $d1, $71, $5d, $c4, $5d, $dd, $01, $d0, $45, $74, $17, $5d, $d1,
$71, $5d, $04, $40, $45, $77, $d1, $71, $5d, $04, $40, $77, $14, $47,
$15, $00, $75, $5c, $5c, $11, $5d, $54, $5c, $17, $77, $47, $17, $01
```

Or written with symbols:

```
-.- .- .-.. -- .- .-. _ .-.. . ..-. - _ -... .-. .- -.-. . _ ..... -.
...-- ..... _ -- ...-- ...-- - ..... _ -- ----- .-. ..... ...-- _ .--
.---- - .... _ ..... ----- -- ...-- _ ...-- -. -.-. .-. -.-- .--. -
.---- ----- -. _ ----- -. _ - ----- .--. _ .-. .. --. .... - _ -... .-.
.- -.-. . --..-- _ .-. . .--. .-.. .- -.-. . _ ... .--. .- -.-. . _ .--
.. - .... _ ..- -. -.. . .-. ... -.-. --- .-. .
```

```
kalmar left brace 5n35 m33t5 m0r53 w1th 50m3 3ncrypt10n 0n t0p right
brace, replace space with underscore
```

```
kalmar{5n35_m33t5_m0r53_w1th_50m3_3ncrypt10n_0n_t0p_right}
```