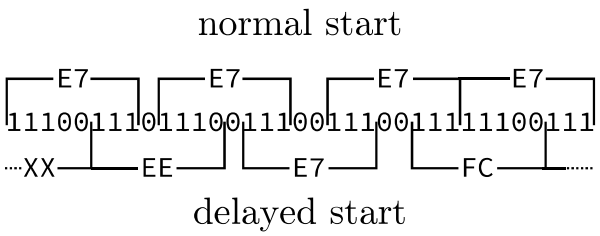
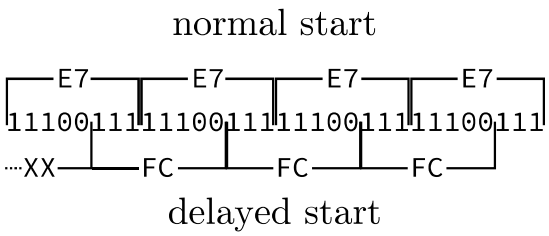
5 In Search of the Most Amazing Thing; or, Towards a Universal Method to Defeat E7 Protection on the Apple][Platform

by Peter Ferrie (qkumba, san inc)
with thanks to 4am





5.1 Introduction

In the early days, there was a protection technique known as the "generic bit-slip protection." In modern times, the cracker known as 4am has dubbed it the "E7 bitstream," because of the trigger values that are used to locate it. It was a very popular technique.

While many nibble-checks could be defeated simply by not allowing them to run at all, some protection routines required that the code be run to produce their side effects, such as to decrypt pages or to emit certain values that are checked later. At a high level, our goal is therefore to simulate the E7 bitstream entirely, allowing the protection routine to run as usual. That is, using a data-only solution to avoid making any changes to the code. Stated explicitly, our goal is to produce either disks that can be copied by COPYA (which, during a copy operation, converts nibble data to *sector data* and then back again) or ".dsk"-format disk images (which contain only sector data). Therefore, we need sector data that, when written to disk, produce *nibble data* that pass the protection check. For that to be possible, we must understand the protection itself and the code that uses it.

A primer on the hardware in general and this technique in particular was included in PoC‖GTFO 10:7. The theory is that after issuing an access of Q6H ($C08D+(slot*16)), the QA switch of the Data Register will receive a copy of the status bits, where it will remain accessible for four CPU cycles. After four CPU cycles, the QA switch of the Data Register will be zeroed. Meanwhile, assuming that the disk is spinning at the time, the Logic State Sequencer continues to shift in the new bits. When the QA switch of the Data Register is zeroed, it discards the bits that were already shifted in, and the hardware will shift in bits as though nothing has been read previously. The relevant code looks like this:

```
READNIB  EQU $C08C
RSTLATCH EQU $C08D

         LDY    #0
NIB1
         LDA    READNIB,X*
         BPL    NIB1
         DEY
         BEQ    FAIL
         CMP    #$D5
         BNE    NIB1
                                  try 256 times:
                                    read nibble, compare with D5
         LDY    #0
NIB2
         LDA    READNIB,X
         BPL    NIB2
         DEY
         BEQ    FAIL
         CMP    #$E7
         BNE    NIB2
                                  try 256 times: (*1)
                                    read nibble, compare with E7
NIB3
         LDA    READNIB,X
         BPL    NIB3
         CMP    #$E7
         BNE    FAIL
                                  read nibble, compare with E7
NIB4
         LDA    READNIB,X
         BPL    NIB4
         CMP    #$E7
         BNE    FAIL
                                  read nibble, compare with E7
         LDA    RSTLATCH,X       desynch
         LDY    #$10
         BIT    $06              ensure >4 cycles between reads
NIB5
         LDA    READNIB,X
         BPL    NIB5
         DEY
         BEQ    FAIL
         CMP    #$EE
         BNE    NIB5
                                  try 16 times: (*2)
                                    read nibble, compare with EE
                                  * X = BootSlot << 4
```

Interestingly, the `bit $06` instruction is a misdirection. It exists only for the purpose of consuming some cycles. Any other instruction of equal duration could have been used, and it might be considered a watermark. While it is the value that exists most commonly, some titles changed the value of the address to `80` or `FF`, and these versions were spread, too.

In the most common implementation of the E7 protection, the stream on disk appears as `D5 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7` with some harmless zero-bits in between. So from where do the other values come? The magic is in the timing of the reads, and timing is everything, so we must count the cycles!

```
LDA     READNIB,X
BPL     NIB4            2 cycles
CMP     #$E7            2 cycles
BNE     FAIL            2 cycles

LDA     RSTLATCH,X      4 cycles

LDY     #$10            2 cycles

BIT     $06             3 cycles
                        ---------
                        15 cycles
```

One bit is shifted in every four CPU cycles, so a delay of 15 CPU cycles is enough for three bits to be shifted in. Those bits are discarded. However, since the CPU and the Disk][system are not synchronized, then depending on exactly when the initial read began, there can be up to two additional cycles in the total count. That puts us in the 16 cycle range, which is sufficient for a fourth bit to be shifted in and then discarded. In any case, the hardware sees it like this, due to a slip of three (or four) bits:

`D5 E7 E7 E7 [slip] EE E7 FC EE E7 FC EE EE FC`

In binary, the stream looks like this, with the seemingly redundant zero-bits in bold.

```
11010101   11100111   11100111   11100111
  D5         E7         E7         E7
11100111 0 11100111 00 11100111   11100111 0 11100111 00
  E7         E7         E7         E7         E7
11100111   11100111 0 11100111 0 11100111   11100111
  E7         E7         E7         E7         E7
```

However, by skipping the first three or four bits, the stream looks quite different.

```
      skipped
     ⌒
11100 11101110 0 11100111 00 11111100   11101110
        EE         E7         FC         EE
    0 11100111 00 11111100   11101110 0 11101110 0 11111100 111...
        E7         FC         EE         EE         FC
```

The old zero-bits are still in bold, and the newly exposed zero-bits are in italics. We can see that the old zero-bits form part of the new stream. This decodes to `EE E7 FC EE E7 FC EE EE FC`, and we have our magic values. The fourth bit must be a zero-bit in the original stream in case only three bits are slipped. Having the fifth bit be a zero-bit in the original stream makes a nice pattern of repeating values, if for no other reason.

5.2 Well-Groomed Data

In order to defeat this at all, we need to produce a regular 6-and-2 encoded sector which can be read by real hardware and copied by regular DOS.

We start by exploiting the point marked by (*1). There's a search for E7 after the D5. This allows us to introduce a full data prologue without breaking the check. So now we have this:

`D5 AA AD E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 ...`

We can even conclude it with a regular epilogue so that there are no read errors. So now we have this:

`D5 AA AD E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 ... DE AA`

It looks like a regular sector. The next step is to fill the stream with the appropriate values, including simulating the presence of the timing bits.

5.3 The Hard Stuff

We will use Bank Street Writer III for our first attempt, since it is the simplest example. Bank Street Writer III requires only one nibble from the pattern to be valid as an 8-bit decryption key for one page of memory. That nibble appears at a position four nibbles after the `EE`, and its value must be `E7`, so our pattern looks like this:

`EE ?? ?? ?? E7 ...`

Since we can't rely on timing bits in our stream (because we need *sector data* that produces *nibble*

data that this code interprets as valid), we can't place the EE inside a pair of E7s because after the bit-slip the wrong value will be read. Instead, we have to encode the value EE directly after discarding the first three bits, and placing a zero-bit in the fourth bit for compatibility purposes. In binary, that looks like this:

```
???01110 1110???? ???????? ????????
???????? 11100111 ...
```

After the bit-slip (and our extra zero-bit), the hardware sees:

```
...11101110 ???????? ???????? ????????
???? [11100111] ...
```

We must make those last four bits "disappear," in order to align our E7 value correctly and allow it to be seen. If we turn those four bits into zeroes and distribute them within the stream, while adhering to the rule of not more than two consecutive zeroes, and replace the rest with ones, we get this:

```
...11101110 11111111 00 11111111 00
11111111 [11100111] ...
```

The hardware reads this as EE FF FF FF E7. Then we prepend one-bits and a zero-bit to the first (partial) nibble, like this:

```
[1110]11101110 11111111 00 11111111 00
11111111 [11100111] ...
```

After realigning the stream, we have this:

```
11101110 11101111 11110011 11111100
11111111 [11100111] ...
```

On disk, it appears as EE EF F3 FC FF E7.

The final step is to pad the data to a multiple of the sector size, so that we have a complete sector. We must also include the calculate the proper checksum. The remaining contents of the sector at this point are entirely arbitrary. We could place a text message or draw a picture, if we chose. Perhaps the most aesthetic version is to include a nibble which will zero the running value, and then fill the rest of the sector with 96s, since 96 is the nibble value for zero. This will yield a sector which is devoid of all content other than the needed values. If that version is chosen, then a quick lookup in the nibble translation table shows us that the nibble value which will zero the running value is F3, so our whole stream appears as:

```
D5 AA AD E7 E7 E7 EE EF F3 FC FF E7 F3
96 96 ... DE AA
```

Great, it runs on hardware.

5.4 Apple for the Win, or Not.

Then we try AppleWin (as at 1.25.0.4). It doesn't work. Why not? Because instead of shifting bits into the data latch one at a time until the top bit is set, AppleWin shifts in an entire nibble immediately. It means that AppleWin does not (and cannot!) support bit-slip at all. Hmm, can we support both at the same time? Let's see about that.

We need to encode the first nibble as an EE, while also allowing a bit-slipping hardware to decode it as an EE. Well, we have that already, so we're halfway there! That just leaves the value four nibbles after the EE, which is currently the arbitrary value of FF. We change that FF to E7, so our stream on disk appears as:

```
EE EF F3 FC E7 E7
```

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with 96s. A quick lookup in the nibble translation table shows us that the needed value is D6, so our whole stream appears as:

```
D5 AA AD E7 E7 E7 EE EF F3 FC E7 E7 D6
96 96 ... DE AA
```

We have a regular sector that works on hardware and AppleWin at the same time.

5.5 Totally Rad

Next up is Rad Warrior. It requires four nibbles from the pattern to be valid (as a 32-bit decryption key for four pages of memory), starting with the fourth nibble. It means that our Bank Street Writer III technique won't work because the pattern will be read differently between the bit-slip and the non-bitslip version, after the fourth nibble.

We have to come up with another technique. We do this by exploiting the point marked by (*2). There's a search for the EE. It means that we can insert nibbles after the point of the bit-slip, which will re-sync the stream to the non-slip form. At that point, we can insert any pattern that we need. We start with an arbitrary compatible sequence:

```
EF FF FF FF
```

In binary, it's:

`11101111 11111111 11111111 11111111`

After the bit-slip (and our extra zero-bit), the hardware sees:

`...11111111 11111111 11111111 1111`

As above, we must make those last four bits disappear, in order to align our pattern later. As above, we turn the four bits into zeroes and distribute them within the stream, while adhering to the rule of not more than two consecutive zeroes. Let's try this:

`...0 11111111 00 11111111 0 11111111`

The hardware reads this as `FF FF FF`. Then we prepend one-bits and a zero-bit to the first (partial) nibble again, like this:

`[1110]011111111 00 11111111 0 11111111`

After realigning the stream, we have this:

`11100111 11111001 11111110 11111111`

On disk, it appears as:

`E7 F9 FE FF`

That final FF is redundant, so we remove it. Then we append our complete pattern without any consideration for bit-slip. Our stream looks like this:

`E7 F9 FE EE E7 FC EE E7 FC EE EE FC`

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with `96`s. A quick lookup in the nibble translation table shows us that the needed value is `FB`, so our whole stream appears as:

`D5 AA AD E7 E7 E7 E7 F9 FE EE E7 FC EE E7 FC EE EE FC FB 96 96 ... DE AA`

We have a regular sector that works on hardware and AppleWin at the same time.

It also immediately supports Batman and Prince of Persia, both of which require the entire pattern (as a 64-bit decryption key for five pages of memory in Batman, and as a seed for several check-bytes during gameplay in Prince of Persia). Superb!

5.6 A Small Bump in the Road

Then we try it all in MAME (as of 0.169), because MAME is supposed to behave like the hardware... But. It. Does. Not. Work. Well, shit. And why not? Because while MAME does support bit-slip, it always consumes four bits for the code above, but most critically, it treats the bit in the fifth position as though it were always a one-bit.

It means that these four sequences are all decoded as `11111111 00 11111111 00` after the bit-slip. (Only one of which is correct.)

```
1 11111111 11110011 11111100
  11101111 11110011 11111100
3 11110111 11110011 11111100
  11100111 11110011 11111100
```

`11110011 11110011 11111100` is decoded as `10111111 00 11111111 00` after the bit-slip, which is not correct, either.

Despite the time that I've spent poring over the source code, I have not yet determined the cause, so we're left to work around it. Can we add support for MAME, while keeping the existing support? Without duplicating everything? Let's see about that.

We need to move a zero-bit beyond the slipped region so that the hardware will read the same bits that MAME does.

```
2 [1110]0 11111111 00 11111111 0x ...
        V-->-->-->-->-->-->-->-^
```

After moving the zero bit, we have `[1110]11111111 00 11111111 00 ....` Realigning that stream, we get `11101111 11110011 11111100 ...`, which looks good. On disk, it appears as `EF F3 FC`.

Then we append our complete pattern without any consideration for bit-slip. This stream is `EF F3 FC EE E7 FC EE E7 FC EE EE FC`.

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with `96`s. A quick lookup in the nibble translation table shows us that the needed value is `EA`, so our whole stream appears as `D5 AA AD E7 E7 E7 EF F3 FC EE E7 FC EE E7 FC EE EE FC EA 96 96 ... DE AA`.

5.7 Success!

We have a truly universal nib sequence, which works on hardware, which works on AppleWin, which works on MAME (and which will still work when the bug is fixed), and which defeats the E7 protection.

Here is our universal sequence in the form of a disk sector:

```
   03 00 03 02 02 02 00 03 03 01 02 02 00 02 02 00
 2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 6 00 00 00 00 00 00 01 00 01 01 03 00 00 01 02 02
   03 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00
 8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 01 00 01 02
12 01 02 01 00 03 00 01 02 01 02 01 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

This can be applied wherever the E7 sequence is the regular pattern. For other patterns, such as those used by Thunder Mountain's "Dig Dug" (E7 EE EE EE E7 E7 EE E7 EE EE EE E7 EE E7 EE EE), Sunburst's "1-2-3 Sequence Me" (BB F9 Fx), and MCE's "The 4th R - Reasoning" (EB B6 EF 9A DB B7 ED F9 D7 BF BD A7 B3 FF B3 BA), just place the proper pattern after the "EF F3 FC" sequence, pad the sector as you like, and then fix the sector checksum.

For the record, the E7 stream is used in many other titles (games or educational software), such as Commando, Deathsword, Ikari Warriors, Impossible Mission II, Karate Champ, Paperboy, Rambo First Blood Part II (a pure text adventure!), Summer/Winter/World Games, The Ancient Art of War [at Sea], Tetris, and Xevious.

As far as we know, this technique first appeared in 1983. It was used to protect the title Locksmith, ironically a product for defeating copy-protection.

None of the disk copiers of the day could copy E7 disks without a parameter unique to the target, so duplicating these disks always required a bit of expertise.

5.8 Final Words

Here is an interesting question: What if you don't have an entire sector available on the track that you need?

Fortunately, this would be a concern only for a protection which used the rest of the sector (and the rest of the track) for meaningful data, which I have not seen so far. In any case, the solution would be to insert only the nibble sequence "EF F3 FC ... EE EE FC" and to not pad the sector. This would yield a freely-copyable disk in its original form. However, we must discourage that idea with these words from 4am:

Never patch an original disk.
Don't reduce the number of original disks in the world.
They aren't making any more of them.
-4am