7 A Brief Description of Some Popular Copy-Protection Techniques on the Apple][Platform

by Peter Ferrie (qkumba, san inc)

7.1 Ancient history

I've been... let's call it "preserving" software since about 1983, albeit under a different name. However, the most interesting efforts have been recent, requiring skills that I definitely didn't have until now: I am the author of the only two-side 16-sector conversion of Prince of Persia[31], the six-side 16-sector conversion of The Toy Shop[32], the single file conversion of Joust, Moon Patrol, and Mr. Do!, as well as the DOS and ProDOS file-based conversions of Aquatron, Conan[33], The Goonies, Jungle Hunt, Karateka, Lady Tut (including the long-lost ending from side B), Mr. Do!, Plasmania, and Swashbuckler, to name a few. I am also the only one to crack Rastan cleanly on the IIGS, just 25 years late.[34] Yes, I do 16-bit, too.

I've spent 13 years writing articles for the Virus Bulletin[35] journal. My faithful readers will recognise the style.

7.2 Isn't it ironic

4am[36] declined to write this document himself, but his work and approval inspired me to do it instead. Since his collection is so varied, and his write-ups so detailed, they served as a rich source of information, which I coupled with my own analyses, to fill in the gaps for titles that I don't have. Everyone knows already that he's funny, but he's also quite friendly and very generous. Together, we corrected a few mistakes in the write-ups, so I gave something back. I even consider us friends now, so I think that I got the better deal.

While I don't *regret* writing this paper, I do have to say that, considering the time and effort that it required, he probably made a wise decision... ;-)

I have tried to associate at least one example of a real program for each technique, but in Section 7.20 you'll find some nifty new protection techniques that I've developed just for this paper.

7.3 Why why why?

Why the Apple][? It's because I grew up with the Apple][, I learned to code on the Apple][, I *know* the Apple][.

Why now? Because the disks that were fresh when the Apple][was current are failing, and if we do not work to preserve them now, some of the titles will be lost forever.

This paper is dedicated to anyone who has an interest in helping to preserve what's left, I sincerely hope it may help to recognise and defeat the copy-protection that they have come across.

7.4 Okay, let's split

We can separate copy protection into two categories; they are either *What You Have* or *What You Know*. What You Have protections are generally protected disks, while What You Know protections are gener-

---

[31] http://pferrie.host22.com/misc/lowlevel14.htm
[32] http://pferrie.host22.com/misc/lowlevel15.htm
[33] http://pferrie.host22.com/misc/lowlevel16.htm
[34] http://www.hackzapple.com/phpBB2/viewtopic.php?t=952
[35] http://www.virusbtn.com
[36] https://archive.org/details/apple_ii_library_4am

ally off-disk, such as requests to type in a word from the manual.

What You Know protections come in several forms. One is an explicit challenge with immediate effect; you must answer now to continue. Another is an explicit challenge with delayed effect; if you answer incorrectly now, the game becomes unplayable later. Yet another is an implicit challenge; in order to proceed, you should perform an action as described in the manual, but the game will *appear* to be playable without it.

Infocom were infamous for their use of all three:

Starcross issued a direct challenge with immediate effect, and you could not even leave the second room without typing the correct co-ordinates from the star chart.[37]

Spellbreaker[38] issued a direct challenge with delayed effect, along the lines of "name the wizard who..." Any name from their word list is accepted, but an incorrect answer results in the player receiving the wrong key. This key cannot unlock a critical door much later in the game, causing the character to be killed instead.

Border Zone made use of an implicit challenge. It required reading the manual in order to know the correct words to excuse yourself — Oopzi Dazi![39]— after bumping into someone, in order to establish contact with the friendly spy. Failure to make contact within the allotted time ended the game.

Brøderbund's Prince of Persia had a variety of delayed effects, depending on which of the several copy protection checks failed. One of them included crashing immediately before showing the closing scene upon winning the game. That is, after completing *fourteen levels*!
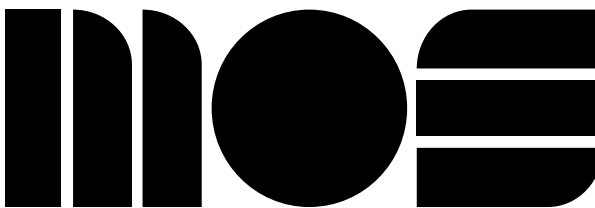
However, the What You Have is perhaps the more interesting, given the vast number of possibilities.

7.5 Accept your limitations

The first important component that we will consider in the Apple][is the MOS 6502 or 65C02 CPU. These CPUs have no separation of code and data. That is, they are a Von Neumann, not Harvard architecture. All memory and I/O addresses are executable, and everything that is not in ROM is writable, including the stack.

Since the stack is writable directly, it introduces the possibility of tricks relating to transfer of control. (§7.14.) Since the stack is executable, it introduces the possibility of hosting code. (§7.18.5.)

The CPU has no prefetch queue, only a single prefetched byte of the next instruction (which is why the minimum instruction execution time is two cycles—one for the instruction, and one for the prefetch), as the last stage in the execution of the current instruction. This introduces the possibility of self-modifying code, including the next instruction to execute, because any memory write will have completed before the prefetch occurs. (§7.15.2.)

7.6 Lay it out for me

The second important component that we will consider in the Apple][is the Disk][controller. The Disk][controller is a peripheral which is placed in a slot. It exposes an interface through memory-mapped I/O, so the various soft-switches can be read and written, just like regular RAM. The interface looks like accesses to `$C0sX`, where `s` is `#$80` plus the slot times 16, and `X` is the switch to access.

The Disk][controller runs independently of the CPU. Once the drive is turned on and spinning the disk, the drive will continue to spin the disk until the drive is turned off again. The drive rotates the disk at a fixed speed—approximately 300 RPM, and five rotations per second, which works out to be 200ms per rotation. However, the speed varies somewhat from drive to drive. For 5.25" disks, the data density is equal across all tracks. At 300 RPM, each

[37] http://infocom.elsewhere.org/gallery/starcross/starcross-map.gif
[38] http://gallery.guetech.org/spellbreaker/spellbreaker.html
[39] http://infodoc.plover.net/manuals/temp/borderzo.pdf p19

track holds 50000 bits, which is equal to 6250 8-bit nibbles.

The data on a disk is simply a stream of bits to be read. For a 5.25" disk, those bits are usually gathered into 16 sectors of 256 bytes each, spread across 35 tracks—$256 \times 16 \times 35 = 143,360$ bytes, or 140kb. When reading from a disk, the Disk][controller shifts in bits at a rate equivalent to one bit every four CPU cycles, once the first one-bit is seen. Thus, a full nibble takes the equivalent of 32 CPU cycles to shift in. After the full nibble is shifted in, the controller holds it in the QA switch of the Data Register for the equivalent of another four CPU cycles, to allow it to be fetched reliably. After those four CPU cycles elapse, and once a one-bit is seen, the QA switch of the Data Register will be zeroed, and then the controller will begin to shift in more bits. As a result, programmers must count CPU cycles carefully to avoid missing nibbles fetched by the controller.

The Disk][controller cannot tell you on which track the head resides. It also cannot tell you on which sector the head resides. (The Shugart SA400 on which the Disk][controller is based does have this capability via index detector circuits, but that feature was removed from the Disk][controller to reduce the cost to manufacture it.) As a result, sectors are usually prepended with a structure known as the "address field", which holds the sector's track and sector number. The controller does not need or use this information. Only the boot PROM makes use of it when requested to read a sector. Beyond that, the information exists solely for the purpose of the program which interprets it.

disk data
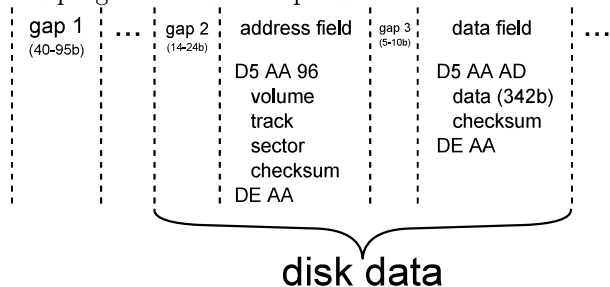
Following the address field that defines a sector's location on the disk, there is another structure known as the "data field", which holds the sector body. One reason for the separate address and data fields is to allow the sector body to be skipped, as opposed to stored and then decoded, in the event that the sector address is not the desired one. Another reason is that it allows a sector to be updated in-place, by overwriting the data field only, instead of rewriting the entire track to update all of the sectors.

(If the sector were a single structure, the CPU time required to verify that the desired sector has been found is so long that the write would begin after the start of the sector body and extend beyond the original end of the sector, overwriting part of the following sector.)

Between the sectors are dead space, which can be filled with a sequence of self-synchronizing values, timing bits, and protection-specific bytes.

The two structures that define a sector are each bounded by a prologue and an epilogue. The prologues for the address and data fields are composed of three values. Two of those values are never used in the sector body, to distinguish the structures from the sector body, and the third value is different between the two structures, to distinguish them from each other. The epilogues for the address and data fields are composed of two values. One of those values is common to both epilogues but never used in the sector body, to distinguish it from the sector data.

The Disk][controller cannot even tell you where it is within the bitstream. The problem is that the stream does not have an explicit start and end. Instead, a specific sequence must be laid on the track, to form an implicit start. That way, the hardware can find the start of the stream reliably. These values are the "self-synchronizing values." For DOS 3.3, and systems with a compatible sector format, the self-synchronising values are composed of a minimum of five ten-bit "FF"s. A ten-bit "FF" is eight bits of one followed by two bits of zero. Self-synchronising values are usually placed before both structures that define a sector, to allow synchronisation to occur at any point on the disk. However, this is not a requirement if read-performance is not a consideration.[40] That is, the fewer the number of self-synchronizing values that are present, the more data that can be placed on a track. However, the fewer the number of self-synchronizing values that are present, the more the controller must read before it can enter a synchronized state, and then start

---

[40]It is a requirement if the data field can be written independently of its address field. Since the write is not guaranteed to begin on a byte boundary, the self-synchronizing values are required for the controller to synchronize itself when reading the data again.

to return meaningful data.

Finally, the Disk][controller can write—but not read reliably—arbitrary eight-bit values. Instead, for reading each eight-bit value, only seven of the bits can be used—the top bit must always be set, in order for the hardware to know when all eight bits have been read, without the overhead of having to count them. (See §7.10.15 for a deeper discussion about an effect made possible by the lack of a counter.) In addition to requiring the top bit to be set, there should not be more than two consecutive zero-bits in a row for the modern drive. (The original disk system did not allow even that. See §7.10.13 for a deeper discussion about the effect of excessive zeroes.)
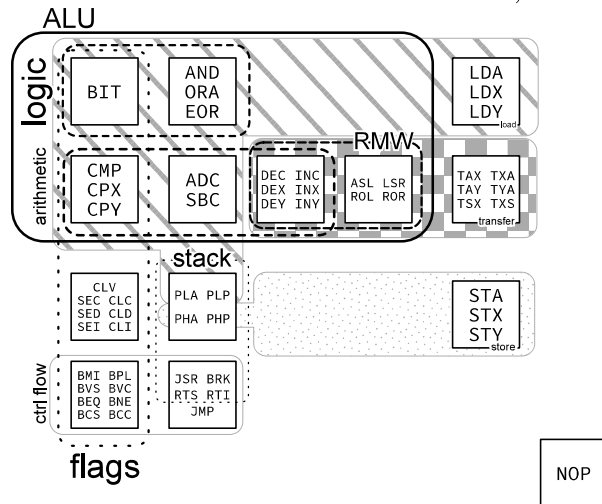
7.7 Copy me, I want to travel

Now that we understand the format of data on the disk, we consider the ways in which that data can be copied.

First is the sector-copier. It relies on sectors being well-defined, and requires knowing only the values for the prologues and epilogues. The sectors are copied one at a time in sequential order, for each of the tracks on the disk, discarding the data between the sectors, and writing new self-synchronizing values instead. Some sector-copiers rely on DOS to perform the writing. In order for that to work, the disk must be formatted first, because that kind of

sector-copier will not write new address fields to the disk. Instead, it will reuse the existing ones, since only the data field needs to be updated to place a sector on a track. In any case, the sector-copier cannot deal easily with deviations from the standard format, and requires a lot of interaction to copy sectors for which the prologue and/or epilogue values are not constant. Some sector-copiers can be directed to ignore the sectors that they cannot read, but obviously this can lead to important data being missed.

Second is the track-copier. It also relies on sectors being well-defined, with known the values for the prologues and epilogues. However, it reads the sectors in the order in which they arrive, and then writes the entire track in one pass[41], by itself. It shares the same limitations as the sector-copier regarding reading sectors and discarding the data between them, but it keeps the sectors in the same order as they were originally, which can be important. (§7.10.9.)

Third is the bit-copier. Unlike the previous two, it makes as few assumptions as possible about the data on the disk. Instead, it treats tracks as the bitstream that they are, and attempts to measure the length of the track while reading.[42] It intends to write the track exactly as it appears on the disk, including the data between the sectors, in one pass. Some bit-copiers can be directed to copy the additional zero-bits in the stream, but there is a limit to how reliably these bits can be detected, and the method to detect them can be exploited. Some bit-copiers can be directed to attempt to reproduce the layout of the disk across track boundaries. See sections 7.10.12 and 7.11.3.

The most important point about copiers in general is that there is simply no way to read data off of a disk with 100% accuracy, unless you can capture the complete bitstream on the disk itself, which can be done only with specialised hardware. There is no way for software alone to read all of the bits explicitly and understand how the controller will behave while parsing them.

---

[41] As opposed to reading the sectors in sequential order, and then writing the entire track—that would only make it a sector-copier with a faster write routine.

[42] A sector-copier can use the collection of sectors as a basic track length; the bit-copier has no such luxury. Instead, it is left to "guess", and might be forced to discard or insert additional data to reconstruct a track of the same length. The difference occurs when the rotation speed of the drive that is being used to make the copy is not the same as that of the drive that was used to make the original.

7.8 Super-super decoder ring

Despite the quite strict requirements regarding the format of data on the disk, DOS introduced two additional requirements regarding the format of data within a sector. The first requirement is that there must not be more than one pair of zero-bits in the value. The second requirement is that there be at least one pair of consecutive one-bits, excluding the sign bit.

If we ignore the DOS requirements for the moment, and consider instead all possible values which comply with the hardware requirement to have no more than two consecutive zero-bits, then there are 81 legal values.

```
10010010 (92)   10101101 (AD)   11001110 (CE)   11101011 (EB)
10010011 (93)   10101110 (AE)   11001111 (CF)   11101100 (EC)
10010100 (94)   10101111 (AF)   11010010 (D2)   11101101 (ED)
10010101 (95)   10110010 (B2)   11010011 (D3)   11101110 (EE)
10010110 (96)   10110011 (B3)   11010100 (D4)   11101111 (EF)
10010111 (97)   10110100 (B4)   11010101 (D5)   11110010 (F2)
10011001 (99)   10110101 (B5)   11010110 (D6)   11110011 (F3)
10011010 (9A)   10110110 (B6)   11010111 (D7)   11110100 (F4)
10011011 (9B)   10110111 (B7)   11011001 (D9)   11110101 (F5)
10011100 (9C)   10111001 (B9)   11011010 (DA)   11110110 (F6)
10011101 (9D)   10111010 (BA)   11011011 (DB)   11110111 (F7)
10011110 (9E)   10111011 (BB)   11011100 (DC)   11111001 (F9)
10011111 (9F)   10111100 (BC)   11011101 (DD)   11111010 (FA)
10100100 (A4)   10111101 (BD)   11011110 (DE)   11111011 (FB)
10100101 (A5)   10111110 (BE)   11011111 (DF)   11111100 (FC)
10100110 (A6)   10111111 (BF)   11100100 (E4)   11111101 (FD)
10100111 (A7)   11001001 (C9)   11100101 (E5)   11111110 (FE)
10101001 (A9)   11001010 (CA)   11100110 (E6)   11111111 (FF)
10101010 (AA)   11001011 (CB)   11100111 (E7)
10101011 (AB)   11001100 (CC)   11101001 (E9)
10101100 (AC)   11001101 (CD)   11101010 (EA)
```

If we introduce the first of the DOS requirements that there not be more than one pair of zero-bits, then there are only 72 compliant values, as we see here:

```
10010101 (95)   10110010 (B2)   11010010 (D2)   11101011 (EB)
10010110 (96)   10110011 (B3)   11010011 (D3)   11101100 (EC)
10010111 (97)   10110100 (B4)   11010100 (D4)   11101101 (ED)
10011010 (9A)   10110101 (B5)   11010101 (D5)   11101110 (EE)
10011011 (9B)   10110110 (B6)   11010110 (D6)   11101111 (EF)
10011101 (9D)   10110111 (B7)   11010111 (D7)   11110010 (F2)
10011110 (9E)   10111001 (B9)   11011001 (D9)   11110011 (F3)
10011111 (9F)   10111010 (BA)   11011010 (DA)   11110100 (F4)
10100101 (A5)   10111011 (BB)   11011011 (DB)   11110101 (F5)
10100110 (A6)   10111100 (BC)   11011100 (DC)   11110110 (F6)
10100111 (A7)   10111101 (BD)   11011101 (DD)   11110111 (F7)
10101001 (A9)   10111110 (BE)   11011110 (DE)   11111001 (F9)
10101010 (AA)   10111111 (BF)   11011111 (DF)   11111010 (FA)
10101011 (AB)   11001010 (CA)   11100101 (E5)   11111011 (FB)
10101100 (AC)   11001011 (CB)   11100110 (E6)   11111100 (FC)
10101101 (AD)   11001101 (CD)   11100111 (E7)   11111101 (FD)
10101110 (AE)   11001110 (CE)   11101001 (E9)   11111110 (FE)
10101111 (AF)   11001111 (CF)   11101010 (EA)   11111111 (FF)
```

If we introduce the second of the DOS requirements that there be at least one pair of consecutive one-bits, excluding the sign bit, then there are only 64 compliant values:

That leaves us with eight values for which there is not more than one pair of zero-bits, but also not one pair of consecutive one-bits, excluding the sign bit. DOS reserves some of these value for a separate purpose.

```
10010101 (95)
11010010 (D2)
11010100 (D4)
11010101 (D5)
10100101 (A5)
10101001 (A9)
10101010 (AA)
11001010 (CA)
```

That leaves us with 17 values for which there are not more than two consecutive zero-bits, which seems like a missed opportunity for a better encoding:

```
10010010 (92)   10101001 (A9)   11100100 (E4)
10010011 (93)   10101010 (AA)
10010100 (94)   11001001 (C9)
10010101 (95)   11001010 (CA)
10011001 (99)   11001100 (CC)
10011100 (9C)   11010010 (D2)
10100100 (A4)   11010100 (D4)
10100101 (A5)   11010101 (D5)
```

Having exactly 64 entries in the table allows us to represent all of the values using six bits. That leads us to an encoding method known as "6-and-2 Group Code Recording (GCR)" or more commonly "6-and-2" encoding.

In "6-and-2" encoding, an eight-bit value is split into two parts, where the high six bits are separated from the low two bits. (The disk system for which DOS 3.2 was first written had an additional restriction that did not allow consecutive zero-bits, and so used "5-and-3" encoding for the same purpose.) To encode an entire sector, each of the two-bit values are gathered together, such that three of them form another six-bit value in reverse order, and are stored first, followed by each of the regular six-bit values. Prior to storing any of the values, they must be transformed into the values in our table of 64 nibbles. This is done by using the original value as an index into the nibble table, and writing the value from the table instead.

When we place the original value beside the nibble value, the table looks like this:

00 = 96	10 = B4	20 = D6	30 = ED
01 = 97	11 = B5	21 = D7	31 = EE
02 = 9A	12 = B6	22 = D9	32 = EF
03 = 9B	13 = B7	23 = DA	33 = F2
04 = 9D	14 = B9	24 = DB	34 = F3
05 = 9E	15 = BA	25 = DC	35 = F4
06 = 9F	16 = BB	26 = DD	36 = F5
07 = A6	17 = BC	27 = DE	37 = F6
08 = A7	18 = BD	28 = DF	38 = F7
09 = AB	19 = BE	29 = E5	39 = F9
0A = AC	1A = BF	2A = E6	3A = FA
0B = AD	1B = CB	2B = E7	3B = FB
0C = AE	1C = CD	2C = E9	3C = FC
0D = AF	1D = CE	2D = EA	3D = FD
0E = B2	1E = CF	2E = EB	3E = FE
0F = B3	1F = D3	2F = EC	3F = FF

DOS reserved two values from our fourth table—#$AA and #$D5—for the prologue signatures. These values are good candidates for the purpose of identifying the headers, because they do not conform to the "at least one pair of consecutive one-bits" criterion, and thus do not conflict with the entries in the "nibbilisation" table. It is not a coincidence that they have alternating bit values; #$D5 is #$55 without the sign bit. By reserving these values, it ensures that the bitstream generated by arbitrary sector data cannot contain a long string of ones (prevented by reserving #$FF), or alternating zeroes and ones (prevented by reserving #$AA and #$D5), regardless of the user's data.

The third value of the prologue signature (#$96 or #$AD) need be unique only between the headers, in order to distinguish between the two. The combination of unique values and non-unique values still produces a unique sequence.

DOS reserved one value from our fourth table—#$AA—for the second byte of the epilogue signatures, for the same reason as for the prologue. The first byte of the epilogue signature need not be unique with respect to sector data (because the combination of unique values and non-unique values still produces a unique sequence), but obviously it must not match the first byte of the prologue, because the third byte of the epilogue (intended to be #$EB) is written sometimes with only limited success (and it is never verified for this reason), and so could potentially be read as the third byte of a prologue instead, with unpredictable results.

The decoding process requires a reverse transformation, via a table which is typically filled with all of the values in a six-bit number. (See the sections on Race Conditions and SpiraDisc for two counter-examples.) The layout of the table is the special thing, though—the nibbles that are read from disk are used as an index into the table, in order to recover the original six-bit value. So the table has gaps between some of the values, because the legal values of the nibbles are not consecutive.

Note that convention is a powerful force. There is no reason for the table to have the nibbilisation entries in that order, or to exclude #$AA or #$D5 (or any of the other 15 entries from the last table) from the set. Further, according to John Brooks, it is possible to use all 81 values from our first table, combined with a special encoding method, which would increase the data density by 105.5%, and potentially even more.[43]

7.9 Write-protection

The absolute simplest possible protection against a copy is to check if the disk is write-protected. The vast majority of owners of duplicated software won't bother to write-protect the disk. If the disk is not write-protected, then the image is considered to be a copy, rather than the original.

Alien Addition uses this technique.

```
1 ;assumes slot 6
  7975    LDA    $C0ED    ;request status
3 7978    LDA    $C0EE    ;read status
  797B    BPL    $7985    ;taken if write-
         enabled
```

A more generic version of the technique is slightly longer:

```
  0000    LDX    $2B        ;fetch slot (x16)
2 0002    LDA    $C08D,X    ;request status
  0005    LDA    $C08E,X    ;read status
4 0008    BPL    $0008      ;hang if write-
         enabled
```

7.10 Sector-level protections

7.10.1 Altered prologue/epilogue

This is one of the simpler techniques available, and was used by many titles. Standard DOS 3.3 uses

---

[43]http://www.bigmessowires.com/2015/08/27/apple-ii-copy-protection/#comment-227325

the sequence `#$D5` `#$AA` `#$96` to identify the address field prologue, `#$D5` `#$AA` `#$AD` to identify the data field prologue, and `#$DE` `#$AA` to identify both of the epilogues. Of course, it is possible to choose from the 17 values from our fifth table, for either the first two bytes of the prologue values, or the second byte of the epilogue. It is also possible to choose from among the 81 values from our first table, for either the third byte of the prologue, or the first byte of the epilogue.

Most commonly, only one value is changed in the prologue or epilogue, and that same value is used for every sector on every track of the disk.

Lucifer's Realm uses this technique; the epilogue was changed from `#$DE` `#$AA` to `#$DF` `#$AA`.

The Tracer Sanction extended the technique by carrying a table of values, and using a different value for each track.

Masquerade extended the technique to the sector level, by requiring that each even sector has one value, and each odd sector has another value. The routine extracts bit zero of the sector number, and then inverts it, to create the key which is applied to the identification byte. Thus, even sectors use `#$D5` (the standard value), and odd sectors use `#$D4`. This is necessary because sector zero of track zero must have the regular value in order to be readable by the boot PROM.

The Coveted Mirror used exactly the same technique—and almost the exact same code—at only the track level.

Due to size limitations, the boot PROM does not verify the epilogue bytes[44] allowing all sectors on all tracks—including the boot sector itself—to be protected. The most common technique involved altering the epilogue values to something other than the default value. This protection cannot be reproduced by a sector-copier or track-copier, which requires the default values to be seen, because they will fail to copy the sector. Operation Apocalypse uses this technique.

Given that the boot PROM does not verify the epilogue bytes, a very light protection technique is to change the epilogue values to something other than the default values for sector zero of track zero only, leaving all other sectors readable. This protection cannot be reproduced by a sector-copier or track-copier which requires the default values to be seen, because they will fail to copy the boot-sector, leaving the disk unusable. Alien Addition makes use

of this technique.

A common technique to defeat this protection is to ignore read errors for all sectors, in the hope that it is caused by the non-default epilogue values alone. However, given the degrading state of floppy disks these days, ignoring read errors can hide the fact that the disk is truly failing.

The address field contains more than just the track and sector numbers. It also contains a volume number. This value can be used as a quick method to determine which disk from a set is currently inserted into the drive. However, support for it—even in DOS—is poor. So many programs, including DOS itself, assume that the volume number is the default value. When it is changed, the read fails. By hard-coding the new value in DOS, the disk will be readable only by itself. Algebra Arcade uses this technique.

This technique can also be used in a slightly different way. Since each sector can have its own volume number, any value can be put there, as long as the program is aware of that fact.

Randamn sets the volume number to a checksum calculated from the current track and sector, and hangs if the values do not match.

Both the address field and data field contain a checksum of the data that precede it, prior to the epilogue. The checksum algorithm is usually a rolling exclusive-OR of each of the bytes, with a zero seed. However, there is no requirement that either of these things is used, for sectors other than sector zero of track zero. For other sectors, the seed can be set to any value, and the algorithm can be a cumulative ADD or anything else at all. This protection cannot be reproduced by a sector-copier or track-copier which relies on the regular algorithm, because the disk will appear to be corrupted.

Hellfire Warrior uses a slight variation on this technique. It maintains a counter at address `$40`, which coincides with the track number which is stored by the boot PROM. In order to break out of the loop that reads sectors into memory, the program requests the boot PROM to read a sector with an intentionally bad checksum. This causes the boot PROM to rewrite the value at address `$40`. The new value is exactly what the program requires as the exit condition. This protection cannot be reproduced by a sector-copier or track-copier, because they will fail to copy this sector, resulting in a disk that has only sectors with good checksums. The disk

---

[44]It also ignores the address field checksum and volume number.

will not boot because it will never exit the loop.

The volume number is normally an eight-bit value. For efficiency of encoding it, DOS uses a "4-and-4" encoding, where the four odd bits are separated from the low even bits, and converted to nibbles. To recombine them, it is a simple matter to shift the nibble holding the odd bits ("abcd") one to the left, resulting in an encoding that looks like "a1b1c1d1", and then to `AND` the result with the nibble holding the even bits ("efgh"), whose encoding that looks like "1e1f1g1h". This method requires 16 bytes to describe the address field. Since the track, sector, and checksum, are known to fit into six bits each, it is easy to see that if the volume number is disregarded, a "6-and-0" encoding can be used instead. This method requires only four nibbles to describe the address field. Algernon uses this technique.

The entries in the address field have a defined order because the boot PROM needs to read them to identify sector zero of track zero, and any other sector which the PROM is asked to read. However, it is possible to change the order of the entries for other sectors on the disk, and then to read the sectors manually.

7.10.2 Fewer sectors

The major reason for using 16 sectors per track is because that is the maximum number that can fit within the standard format created by DOS 3.3. DOS 3.2 supported only 13 sectors per track, because of the limitation of the hardware regarding consecutive zeroes. Copy protection techniques are free to use fewer sectors than either of those values.

Wavy Navy uses ten sectors per track, while Olympic Decathlon uses eleven and Karateka uses a dozen. The sectors in these examples are all the regular size, but encoded in a wasteful manner. (Primarily the "4-and-4" encoding was used because the decoder is very small, but sometimes "5-and-3" because the decoder looks weird when compared with the more familiar "6-and-2" encoding.) The wasteful encoding is the reason for the reduced sector count; there really isn't more room for more sectors.

7.10.3 More sectors

The standard DOS 3.3 format disk uses 16 individual sectors per track, with relatively large gaps between the sectors. Consider how much space would be available if those sectors were combined into a single large sector, with a single field that combines both address (specifically, only the track number) and data fields. Yes, it would require reading the entire track in order to find the field again once the track had been verified, but for some applications, performance is not that critical. This is what Infocom did, on programs such as A Mind Forever Voyaging. Once the track had been found, and the data field found again, then the program read (and discarded) sectors sequentially until the required one was found. Again, if the performance is not that critical, the fact that the routine can fetch only one sector at a time is not an issue. In fact, the implementation works well enough for the text-adventure scenario in which it was used. Since the user will be reading the text while additional text is loading, the time required for that loading goes mostly unnoticed.

Consider how much space would be available if those gaps were reduced to the minimum of five self-synchronizing values before the address field prologue, with just a few bytes of gap between the address and data headers. Then reducing the prologue byte count from three to two, and the epilogue byte count from two to one. Consider how much space would be available by merging groups of sectors. If you converted the track into six sectors of three times the size, you would have RWTS18. This is a good compromise between speed and density. On one side, having fewer sectors means less processing; and on the other side, having more sectors means less latency to find a sector. The RWTS18 routine also supports "read scattering" by assigning a dummy write address to the pages that aren't needed.

This second technique was used very heavily by Brøderbund, on programs such as Airheart (and even three years later, on Prince of Persia), but other companies made use of it, too, such as Infogrames in Hold-Up. Interestingly, in the case of Airheart, after compressing the title screen to reduce its size

on the disk, the rest of the game fit on a regular 16-sector disk.

7.10.4   Big sectors

There is no requirement to define multiple sectors per track. It is possible to define a single sector that spans the entire track.[45] However, there can be a significant time penalty while reading such a track, because it requires up to one complete rotation in order to find the start of the sector.

Lady Tut uses a single sector per track, at a size equivalent to eleven 256-bytes sectors.

7.10.5   Encoded sectors

As noted previously, there is no reason for a disk to use our sixth table—there is no reason to have the nibbilisation entries in that order, nor even to use those values at all. Any alteration to the table results in a disk that can be copied freely, but whose contents cannot be read from the outside. Further, the DOS on such a disk cannot write files from the inside to the outside. The reason why the read would fail is because the standard table would be applied to data that requires the alternative table to decode, resulting in the wrong decoding. The reason why the write would fail is because the alternative table would be applied to data that requires the standard table to encode, resulting in the wrong encoding.

Maze Craze Construction Set uses an alternative nibble table—all of the values from `#$A9-FF` from our first table. These values might have been chosen because they provide the least sparse array when used as indexes.

Bop'N Wrestle uses the regular nibble table (and a standard DOS 3.3), but in reverse order.

7.10.6   Duplicated sectors

The address field carries the sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible to have multiple sectors with the same number.[46] There are numerous ways in which they could be distinguished, such as by the volume number. A protection technique could set every sector number to the same value in the address field. It could set them all to zero, provided that the checksum algorithm is changed, so that the boot PROM will read successfully only the true sector zero, in order to boot the disk. It could also use the volume number from the address field as the page number in which to write the sector data. This would be a very compact way to load data without the need to pass the address as a parameter to the loader.

Math Blaster has two sectors numbered zero on track zero. The program distinguishes between them by examining the first nibble after the address field epilogue, but the checksum of the second sector zero also fails verification, which is why the boot PROM does not see it. This protection cannot be reproduced by a sector-copier or track-copier, because those copiers will write only a single sector zero to a track. It is unpredictable which of the two sector zeroes would be written, but even if the true one is chosen, the copy is revealed by the program missing the duplicated sector.

7.10.7   Sector numbering

The address field carries the sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible to have sectors whose number is not in the range of zero to 15.[47] Any eight-bit value can be used, as long as the program is expecting it. This protection cannot be reproduced by a sector-copier, because the copier will not copy those sectors at all.

7.10.8   Sector location

The address field carries the track and sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible for a sector to "lie" about its location on the disk. For example, the address field of sector three on track zero could label itself as sector zero on track three. This protection cannot be reproduced by a sector-copier which relies on DOS to perform the write, because they will

---

[45]This would be the equivalent of about 18.5 256-bytes sectors in "6-and-2" encoding. Using 19 sectors is possible, if the full range of values from the first figure is used, but it introduces a problem to identify the start of the sector, since there are no single values that can be reserved exclusively. One possible solution is to find a sequence which cannot appear in user-data due to particular characteristics of the decoding process. Just because it is possible, it doesn't mean that it's easy.

[46]The same is true for the track number, and Jumble Jet has multiple tracks which claim to be track zero.

[47]The same is true for the track number. That is, a number which is not in the range of zero to 34.
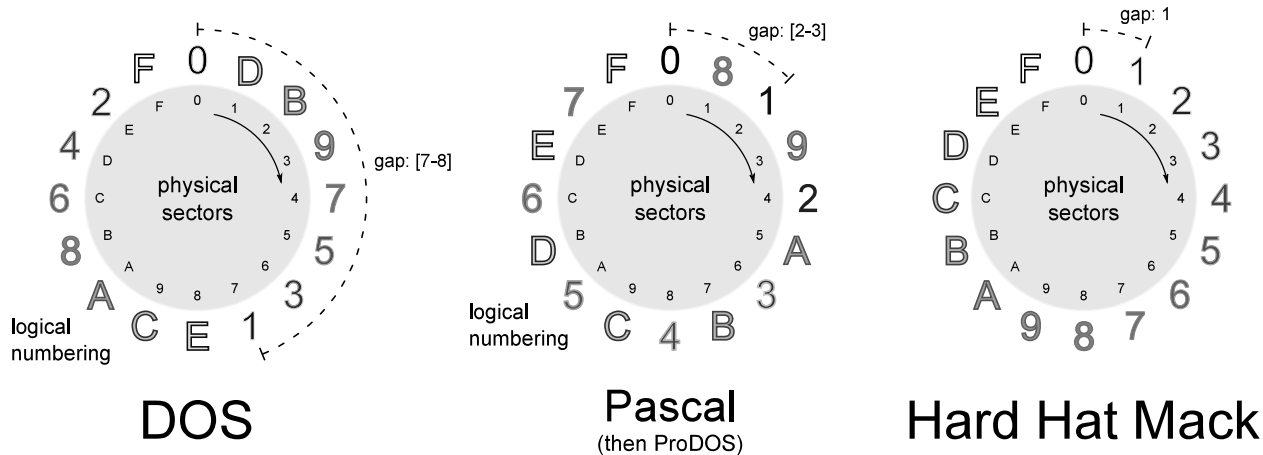
Figure 16 – Floppy sectors interleaving.

not duplicate this information, because DOS will fill in the address field by itself when placing the sector on the disk. Thus, a program that seeks to a track that contains "misplaced" sectors will not find any misplaced sectors, or will receive the wrong content instead.

Discover uses this technique; it changes the identity of one particular sector in the sector interleave table, on one particular track.

7.10.9 Synchronised sectors

Since the approximate rotation speed of the drive is known (~300 RPM), it becomes possible to place sectors at specific locations on a track, such that they have a special position relative to other sectors on the same track. This is difficult to reproduce because of the delay that is introduced while a sector-copier is writing the data.

Hard Hat Mack takes this to the extreme, by requiring that one track has all 16 sectors in incremental order. This protection is highly unlikely to be reproduced by using a sector-copier, because after factoring in the rotation speed of the drive, the next sector is more likely to be placed halfway around the disk.

7.10.10 Bad sectors

Some protections rely on the fact that intentionally bad sectors (for example, checksum mismatch in the simplest case, but potentially physical damage could be used, too) should return a read error.

Drelbs uses this technique. This protection cannot be reproduced even with a bit-copier, because

the copy will have no sectors that cannot be read.

7.10.11 Dead-space bytes

The data for a sector is well defined, but apart from the optional presence of the self-synchronizing values, the data between sectors is not defined at all. As a result, it is not often copied, either. It is possible to place specific counts of specific values in this location, which can be checked later. A program can detect a copy by the absence or wrong count of the special values.

Randamn checks the value of the byte immediately before the prologue of a particular sector, and reboots if the value looks like a self-synchronizing value. (A bit-copier might insert this values when asked to match the track length, and a sector-copier would always insert the value.)

Binomial Multiplication counts the number of values that appear between the address field epilogue and the data field prologue, and between the data field epilogue and the next sector address field prologue, for all of the sectors on a particular track. This protection cannot be reproduced by a sector-copier or a track-copier, because those copiers will discard the original data between the sectors.

7.10.12 Timing bits

The Disk][controller shifts in bits at a rate equivalent to one bit every four CPU cycles, once the

48

first one-bit is seen. Thus, a full nibble takes the equivalent of 32 CPU cycles to shift in. After the full nibble is shifted in, the controller holds it in the QA switch of the Data Register for the equivalent of another four CPU cycles, to allow it to be fetched reliably. After those four CPU cycles elapse, and once a one-bit is seen, the QA switch of the Data Register will be zeroed, and then the controller will begin to shift in more bits. The significant part of that statement is "once a one-bit is seen." It is possible to intentionally introduce "timing" (zero) bits into the stream in order to delay the reset. For each zero-bit that is present, the previous value will be held for another eight CPU cycles. For code that is not expecting these zero-bits to be present, a nibble that is being held back will be indistinguishable from a nibble that has newly arrived.

Creation uses this technique. It looks like this:

```
 ;wait  for  nibble  to  arrive
2 B94F      LDA    $C08C,X
  B952      BPL    $B94F
4 ;watch  for  #$D5
  B954      CMP    #$D5
6 B956      BNE    $B948
  ;delay  to  ensure  >  4  cycles  will  elapse
8 ;before  the  next  read  occurs
  B958      NOP
10 ;read  data  latch
  B959      LDA    $C08C,X
12 ;check  if  nibble  has  changed
  ;if  zero-bit  is  present,
14 ;then  read  value  lasts  longer
  B95C      CMP    #$D5
16 B95E      BEQ    $B972
```

Hacker II requires a pattern of zero-bits to be present in the stream. The effect of the delayed shift becomes clear when we count cycles.

```
  ;initialise  mask
2 403A      LDA    #$08
  ...
4 ;wait  for  nibble  to  arrive
  4044      LDY    $C08C,X
6 4047      BPL    $4044       ;2  cycles
  ;watch  for  #$FB
8 4049      CPY    #$FB        ;2  cycles
  404B      BNE    $403A       ;2  cycles
10 ;not  a  do-nothing  instruction!
  ;exists  to  be  timing-identical
12 ;to  the  BEQ  at  $4062
  404D      BEQ    $404F       ;3  cycles
14 404F      NOP                ;(2  cycles)
  4050      NOP                ;(2  cycles)
16 ;read  data  latch
  4051      LDY    $C08C,X    ;(4  cycles)
18 ;check  how  many  bits  have  shifted  in
  4054      CPY    #$08
```

```
20 ;shift  carry  into  A
  4056      ROL
22 ;until  a  set  bit  is  shifted  out
  ;(takes  five  rounds)
24 4057      BCS    $4064
  ;wait  for  nibble  to  arrive
26 4059      LDY    $C08C,X
  405C      BPL    $4059       ;2  cycles
28 ;watch  for  #$FF
  405E      CPY    #$FF        ;2  cycles
30 4060      BNE    $403A       ;2  cycles
  4062      BEQ    $404F       ;3  cycles
32 ;wait  for  nibble  to  arrive
  4064      LDY    $C08C,X
34 4067      BPL    $4064
  ;remember  its  value
36 4069      STY    $07
  ;check  if  proper  pattern  was  seen
38 ;(alternating  zero-bit  yes  and  no)
  406B      CMP    #$0A
40 406D      BNE    $403A
  ;wait  for  nibble  to  arrive
42 406F      LDA    $C08C,X
  4072      BPL    $406F
44 ;checksum  against  previous  value
  ;both  must  be  #$FF  to  pass
46 4074      SEC
  4075      ROL
48 4076      AND    $07
  4078      EOR    #$FF
50 407A      BEQ    $4080
```

The timing loop is long enough for four nibbles to be shifted in if no zero-bit is present, resulting in a value of at least #$08. (Specifically the right-hand "F" from the value "FF".) If a zero-bit is present, then fewer than four nibbles will be shifted in, resulting in a value of less than #$08. This explains the "CPY #$08" instruction at $4054. It is checking if a one-bit has been shifted in four times or three times.

The "CMP #$0A" instruction at $406B is checking the final results of the multiple CPYs that were made. In binary, the results look like 01010 but prior to that, the results progress like this:

00010000
00100001
01000010
10000101
00001010

That means it is expecting the first pass to have a value of less than eight (carry clear), then a value of at least eight (carry set), then a value of less than eight (carry clear), then a value of at least eight (carry set), and finally a value of less than eight (carry clear), followed by two "FF"s. That requires the stream to look like FB 0 FF FF 0 FF FF 0 Fx

49

FF FF.

7.10.13 Floating bits

What happens if more than two consecutive zero-bits are present in a stream? Something random. The Automatic Gain Control circuit will eventually insert a one-bit because of amplified noise. It might happen immediately after the second zero-bit, or it might happen after several more zero-bits. The point is that reading that part of the stream repeatedly will yield different responses.

Mr. Do! uses this technique.

```
     ;set counter to be used later
2    0710    LDY     #$06
     ...
4    ;set state
     0713    LDA     #$FF
6    0715    STA     $07C2
     ;wait for nibble to arrive
8    0718    LDA     $C088,X
     071B    BPL     $0718
10   ;watch for #$D5
     071D    CMP     #$D5
12   071F    BNE     $0718
     ;wait for nibble to arrive
14   0721    LDA     $C088,X
     0724    BPL     $0721
16   ;watch for #$9B
     0726    CMP     #$9B
18   0728    BNE     $071D
     ;wait for nibble to arrive
20   072A    LDA     $C088,X
     072D    BPL     $072A
22   ;watch for #$AB
     072F    CMP     #$AB
24   0731    BNE     $071D
     ;wait for nibble to arrive
26   0733    LDA     $C088,X
     7036    BPL     $0733
28   ;watch for #$B2
     0738    CMP     #$B2
30   073A    BNE     $071D
     ;wait for nibble to arrive
32   073C    LDA     $C088,X
     073F    BPL     $073C
34   ;watch for #$9E
     0741    CMP     #$9E
36   0743    BNE     $071D
     ;wait for nibble to arrive
38   0745    LDA     $C088,X
     0748    BPL     $0745
40   ;watch for #$BE
     074A    CMP     #$BE
42   074C    BNE     $071D
     ;wait for nibble to arrive
44   074E    LDA     $C088,X
     0751    BPL     $074E
46   ;loop six times
     0753    DEY
48   0754    BNE     $074E
```

```
     ;change state
50   0756    INC     $07C2
     0759    BNE     $2761
52   ;store last read value on first pass
     075B    STA     $07C3
54   ;allow complete revolution and read again
     075E    JMP     $071D
56   ;check last read value on subsequent pass
     ;must be different from the first pass
58   0761    CMP     $07C3
     0764    BNE     $0771
60   ;retry up to four times
     0766    INC     $07C2
62   0769    LDA     $07C2
     076C    CMP     #$08
64   076E    BNE     $271D
```

On the first pass, the program watches for the sequence $#D5 #$9B #$AB #$B2 #$9E #$BE, skips the next five nibbles, and then reads and saves the sixth nibble. On subsequent passes, the program watches again for the sequence $#D5 #$9B #$AB #$B2 #$9E #$BE, skips the next five nibbles, and then reads and compares the sixth nibble against the sixth nibble that was read initially. The value that is read will always be a legal value, but on the original disk, with multiple zero-bits in the stream, the value that was read in one of the subsequent passes will not match the value that was read in the first pass. No matter how many extra zero-bits existed in the stream, the bit-copier will not write them out. Instead, it will "freeze" the appearance of the stream, and normalise it so that there are no more than two zero-bits emitted. As a result, the sixth nibble that was read will have the same value for all passes, and therefore fail the protection check.

7.10.14 Nibble count

Since a track is simply a stream of bits, it is possible to control the layout of the values in that stream, as long as it follows the rules of the hardware. The number of self-sychronizing values can be reduced to a single set of the minimum number, if performance is not a consideration. That means there are no other zero-bits present on the track. However, a bit-copier cannot detect the zero-bits reliably (neither their presence, nor their number), so it is left to guess if the value #$FF must be stored using eight or ten bits. (That is, if it is a data nibble or a self-synchronizing value.) If there are enough #$FF bytes on a track, and if the bit-copier assumes that every one of them must be ten bits wide, then it is possible that the bit-copier will write more data

than can fit on the track, resulting in part of the track being overwritten when the revolution completes before the write completes.

As a separate technique, it is also possible to reduce the speed of the drive while writing the data to the original disk, resulting in a track that is so dense, that the data cannot fit on a disk when written at regular speed. This is known as a "fat" track.

The more common technique is to simply use a sequence of nibbles with enough zero-bits between them, that the "delayed fetch" effect is triggered. (§7.10.12.) When the zero-bits are present, and if the fetch is fast enough (that is, it polls the QA switch of the Data Register while the top bit is clear, stores the fetched value, and then resumes polling), then there will appear to be more nibbles of a particular value than really exist, because the next bit will not be ready to shift in. A program that counts the number of nibbles will see more nibbles in the copy than in the original.

If the fetch is slow enough... now, this is an interesting case. Bit-copiers try to read the data as quickly as it comes in. This is done not by polling the QA switch of the Data Register, but by checking if the top bit is already set, in an unrolled loop, like this:

```
     ;2 cycle delay so
 2   ;shift might finish
     TDL1    NOP
 4   ;try to detect timing bit
     LDA $C0EC, X
 6   BMI TDS2
     TDL2    LDA $C0EC, X
 8   BMI TDS2
     ;timing bit probably present
10   LDA $C0EC, X
     BMI TDS3
12   LDA $C0EC, X
     BMI TDS3
14   LDA $C0EC, X
     BMI TDS3
16   LDA $C0EC, X
     BMI TDS3
18   ;3 cycle penalty if taken!
     BPL TDL2
20   TDS2    STA ($0), Y
     ...
22   RTS
     ;store value with timing bit
24   ;loses one bit as a result
     TDS3    AND #$7F
26   STA ($0), Y
     ...
28   RTS
```

This code is a disassembly from Essential Data Duplicator (E.D.D.), but apart from the BPL instruction, it is shared by Copy][+. (Someone copied!) Normally, a nibble will be shifted in before TDL2 completes, so that TDS2 is reached, and the nibble is stored intact. However, by using only six fetches, the code is vulnerable to a well-placed timing bit, such that the BPL will be reached just before the last bit of the nibble is shifted in. That three-cycle time penalty when the branch is taken is just enough that, when combined with the two-cycle instruction before it, the shift will complete, and the four CPU cycles will elapse, before the next read occurs. The result is that the nibble is missed, and the next few nibbles that arrive will reach TDS3 instead, losing one bit each. When those data are written to disk by the bit-copier, the values will be entirely wrong.

Create With Garfield: Deluxe Edition uses this technique. (The original Create With Garfield uses an entirely different protection.) It has one track that is full of repeated sequences. Each of the sequences has a prologue of five bytes in length. Every second one of the prologues has a timing bit after each of the five bytes in the prologue. In the middle of the track is a collection of bytes which do not match the sequence, so the track is essentially split into two groups of these repeated sequences. The size of the two groups is the same. When the bit-copier attempts to read the data, the timing bits cause about half of the sequences to be lost. What remain are far fewer sequences than exist on the original disk. (Enough of them that the bit-copier mistakenly believes that it has copied the track successfully.) A program can detect a copy by the small count of these sequences. This technique is likely to have been created to defeat E.D.D. specifically, but Copy][+ is also affected. However, the protection can be reproduced with the use of a peripheral that connects to the drive controller (and thus see the zero-bits for exactly what they are), or by inserting an additional fetch in the software.

7.10.15 Bit-flip, or defeat bit-copiers with this one weird trick

Deeply technical content follows. Prepare yourself!

Let's take this simple sentence (sorry, but it's the best example that I could create at the time):
ITHASGOTTOBETHISLANDAHEAD

And split it according to some potential word boundaries:
IT HAS GOT TO BE THIS LAND AHEAD

Now we skip a bit:

OTTO BETH ISLAND AHEAD

A bit more:

TO BETH ISLAND AHEAD

A bit more still:

BET HIS L AND A HEAD

Okay, that last one doesn't make much sense, but I wanted a sentence which could be read differently, depending on where you started reading, as opposed to a series of arbitrary overlapping words. In any case, it's clear that depending on where you start reading, you can get vastly different results. Something similar is possible while reading the bit-stream from the disk. After a nibble is shifted in (determined by the top bit being set), and the four CPU cycles have elapsed, and once the one-bit is seen, then the QA switch of the Data Register is set to zero. The absence of a counter allows the hardware to be fooled about how many bits have been read. Specifically, the controller can be convinced to discard some of the bits that it has read from the disk while forming a nibble, and then the starting position within the stream will be shifted accordingly. This is possible with a single instruction, in conjunction with an appropriate delay.

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 (LSS) 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. Let's see that in action.

Tinka's Mazes does it this way, beginning with some preamble code which is common to many programs that used this technique.

```
      BB6A      LDY       #0
 2  ;wait  for  nibble  to  arrive
      BB6C      LDA       $C08C,X
 4    BB6F      BPL       $BB6C
      BB71      DEY
 6  ;retry  up  to  256  times
      BB72      BEQ       $BBBB
 8  ;watch  for  #$D5
      BB74      CMP       #$D5
10    BB76      BNE       $BB6C
      BB78      LDY       #0
12  ;wait  for  nibble  to  arrive
      BB7A      LDA       $C08C,X
14    BB7D      BPL       $BB7A
      BB7F      DEY
16  ;retry  up  to  256  times
      BB80      BEQ       $BBBB
18  ;watch  for  #$E7
      BB82      CMP       #$E7
20    BB84      BNE       $BB7A
      ;wait  for  nibble  to  arrive
22    BB86      LDA       $C08C,X
      BB89      BPL       $BB86
24  ;watch  for  #$E7
      BB8B      CMP       #$E7
26    BB8D      BNE       $BBBB
      ;wait  for  nibble  to  arrive
28    BB8F      LDA       $C08C,X
      BB92      BPL       $BB8F
30  ;watch  for  #$E7
      BB94      CMP       #$E7
32    BB96      BNE       $BBBB
```

Here is the switch:

```
     ;trigger desync
 2   BB98    LDA     $C08D,X
     BB9B    LDY     #$10
 4   ;delay to ensure > 4 cycles will elapse
     ;before the next read occurs
 6   BB9D    BIT     $6
     ;wait for nibble to arrive
 8   BB9F    LDA     $C08C,X
     BBA2    BPL     $BB9F
10   BBA4    DEY
     ;retry up to 16 times
12   BBA5    BEQ     $BBBB
     ;watch for #$EE
14   BBA7    CMP     #$EE
     BBA9    BNE     $BB9F
16   BBAB    LDY     #7
     ;wait for nibble to arrive
18   BBAD    LDA     $C08C,X
     BBB0    BPL     $BBAD
20   ;compare backwards against the list at $BBC1
     ;E7 FC EE E7 FC EE EE FC
22   BBB2    CMP     ($48),Y
     BBB4    BNE     $BBBB
24   BBB6    DEY
     BBB7    BPL     $BBAD
26   ;pass
     BBB9    CLC
28   BBBA    RTS
     BBBB    DEC     $50
30   ;retry if count remains
     BBBD    BNE     $BB57
32   ;fail
     BBBF    SEC
34   BBC0    RTS
     BBC1    .BYTE   $FC,$EE,$EE,$FC,$E7,$EE,$FC,
             $E7
```

But wait, there's more! To see the bitstream on disk, it looks like D5 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? Since the magic is in the timing of the reads, we must count cycles:

```
 1   BB8F    LDA     $C08C,X
     BB92    BPL     $BB8F    ;2 cycles
 3   BB94    CMP     #$E7     ;2 cycles
     BB96    BNE     $BBBB    ;2 cycles
 5   BB98    LDA     $C08D,X  ;4 cycles
     BB9B    LDY     #$10     ;2 cycles
 7   BB9D    BIT     $6       ;3 cycles
     ;total: 15 cycles
```

Time passes...

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. Back to our stream. In binary, it looks like the following, with the seemingly redundant zero-bits in bold.
11100111 **0** 11100111 **00** 11100111 11100111 **0** 11100111 **00** 11100111 11100111 **0** 11100111 **0** 11100111 11100111
However, by skipping the first three bits, the stream looks like this:
00 111**0**1110 *0* 111**00**111 *00* 11111100 111**0**1110 *0* 111**00**111 *00* 11111100 111**0**1110 *0* 111**0**1110 *0* 11111100 111...

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 E7 FC EE E7 FC EE EE FC, and we have our magic values●

Programs from Epyx that use this protection do not compare the values in the pattern. Instead, the values are used as a key to decode the rest of the data that are loaded. This hides the expected values, and causes the program to crash if they are altered.

The Thunder Mountain version of Dig Dug uses a slight variation on the technique, including a different preamble and switch. The company seems to have kept the variation to themselves. (Bop'N Wrestle from 1986 uses the same altered version, and comes from Mindscape, but Mindscape owned the Thunder Mountain label, so the connection is clear.)[48] That version looks like this:

```
     0224 LDY #$00
 2   ;wait for nibble to arrive
     0226 LDA $C08C,X
 4   0229 BPL $2226
     022B DEY
 6   ;retry up to 256 times
     022C BEQ $2275
 8   022E CMP #$AD
     0230 BNE $2226
```

A different prologue value is checked, allowing the bitstream to begin like a regular sector: D5 AA AD...

Here is the switch:

```
 1   ;trigger desync
     0252 LDA $C08D,X
```

---

[48] Interestingly, one title from Thunder Mountain and released in the same year is known to use the regular version. It is entirely possible that the alternative version was developed in-house to avoid paying royalties to protect other products.

```
 3| 0255 LDY #$10
  | ;no delay instruction in this version
 5| ;wait for nibble to arrive
  | 0257 LDA $C08C,X
 7| 025A BPL $2257
  | 025C DEY
 9| ;retry up to 16 times
  | 025D BEQ $2275
11| ;watch for #$E7 instead, but it's not a ''
  |     true'' E7
  | 025F CMP #$E7
13| 0261 BNE $2257
  | ;and double the size of the pattern to match
15| 0263 LDY #$0F
```

The bitstream on disk looks like `D5 AA AD` [many 96s] `E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7` with some harmless zero-bits in between. The desync timing is only 12 cycles, but the required pattern is not found right away, so the delay is not as interesting. In binary, the stream looks like 11100111 11100111 11100111 **00** 11100111 **0** 11100111 **0** 11100111 **0** 11100111 **00** 11100111 **00** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **0** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **0** 11100111 with the seemingly redundant zero-bits in bold. However, by skipping the first three bits, the stream looks like this:
00 11111100 11111100 111**00**111 (← E7, but not aligned) *00* 111**0**1110 *0* 111**0**1110 *0* 111**0**1110 *0* 111**00**111 *00* 111**00**111 *00* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**0**1110 *0* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**0**1110 *0* 111...

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 `FC` (ignored) `FC` (ignored) `E7 EE EE EE E7 E7 EE E7 EE EE EE E7 EE E7 EE EE`, a very smooth sequence indeed. Put simply, each single bold zero-bit sequence results `EE` being seen, and every double bold zero-bit sequence results in `E7` being seen, allowing easy control over exactly how smooth the sequence is.

1-2-3 Sequence Me uses the same technique but with different values:

```
 1| ;wait for nibble to arrive
  | BA5B    LDA   $C08C,X
 3| BA5E    BPL   $BA5B
  | ;watch for #$AA
 5| BA60    CMP   #$AA
  | BA62    BEQ   $BA7A
 7| ...
  | BA7A    LDY   #$02
```

```
 9| ;trigger desync
  | BA7C    LDA   $C08D,X
11| ;delay while status is loaded
  | BA7F    PHA
13| ;balance stack
  | BA80    PLA
15| ;wait for nibble to arrive
  | BA81    LDA   $C08C,X
17| BA84    BPL   $BA81
  | ;watch for #$BB
19| BA86    CMP   #$BB
  | BA88    BEQ   $BA8F
21| BA8A    DEY
  | ;retry if count remains
23| BA8B    BPL   $BA81
  | ;fail
25| BA8D    BMI   $BA77
  | ;wait for nibble to arrive
27| BA8F    LDA   $C08C,X
  | BA92    BPL   $BA8F
29| ;watch for #$F9
  | BA94    CMP   #$F9
31| BA96    BNE   $BA77
```

That stream looks like `AA EB 97 DF FF` with some harmless zero-bits in between. Now let's count the cycles:

```
 1| BA5B    LDA   $C08C,X
  | BA5E    BPL   $BA5B    ;2 cycles
 3| BA60    CMP   #$AA     ;2 cycles
  | BA62    BEQ   $BA7A    ;3 cycles
 5| ...
  | BA7A    LDY   #$02     ;2 cycles
 7| BA7C    LDA   $C08D,X  ;4 cycles
  | BA7F    PHA            ;3 cycles
 9| ;total: 16 cycles
```

One bit is shifted in every four CPU cycles, so a delay of 16 CPU cycles is enough for four bits to be shifted in. Those bits are discarded. Back to our stream. In binary, it would look like this:
11101011 **0** 10010111 **0** 11011111 **00** 11111111
with the seemingly redundant zero-bits in bold. However, by skipping the first four bits, the stream looks like this:
1011**0**100 10111**0**11 *0* 11111**00**1 111111...

The old zero-bits are still in bold, and the newly exposed zero-bit is in italics. We can see that the old zero-bits form part of the new stream. This decodes to `B4` (**ignored**) `BB F9 Fx`, and we have our magic values.

The 4th R: Reasoning uses another variation of this technique. Instead of matching the values explicitly, it watches for the data field on a particular sector, waits for three nibbles and three bits to pass,

and then reads and stores the next 16 nibbles in an array. Then it calculates a checksum of those 16 nibbles, and uses the checksum as an index into the table of those 16 nibbles, to fetch two 8-bit keys in a row. The table is treated as a circular list, so if the index were 15, then the two keys would be formed by fetching the last entry in the array and the first entry in the array. The keys are used to decipher the other nibbles that are read from all of the other sectors on the disk. It looks like this:

```
 1 ;wait for nibble to arrive
   BB63      LDA      $C08C,X
 3 BB66      BPL      $BB63
   ;wait for nibble to leave
 5 ;if zero-bit is present,
   ;then read value lasts longer
 7 BB68      LDA      $C08C,X
   BB6B      BMI      $BB68
 9 ;wait for nibble to arrive
   BB6D      LDA      $C08C,X
11 BB70      BPL      $BB6D
   ;trigger desync
13 BB72      STA      $C08D,X
   ;delay to reduce number of times
15 ;that branch will be taken
   BB75      NOP
17 ;wait for status value to leave
   ;if zero-bit is present,
19 ;then read value lasts longer
   BB76      LDA      $C08C,X
21 BB79      BMI      $BB76
   ;wait for next nibble to arrive
23 BB7B      LDA      $C08C,X
   BB7E      BPL      $BB7B
```

That stream looks like CF CF 9E FD ED BB E6 B6 ED FB FC EB DF DE D3 D9 FF D9 DD D7 with some harmless zero-bits in between. Now let's count those cycles:

```
   BB63      LDA      $C08C,X
 2 BB66      BPL      $BB63
   BB68      LDA      $C08C,X
 4 BB6B      BMI      $BB68
   BB6D      LDA      $C08C,X
 6 BB70      BPL      $BB6D      ;2 cycles
   BB72      STA      $C08D,X    ;5 cycles
 8 BB75      NOP                 ;2 cycles
   BB76      LDA      $C08C,X    ;4 cycles
10 ;but +4 cycles for each time reached
   ;because of zero-bit
12 BB79      BMI      $BB76      ;2 cycles
   ;but +3 cycles for each time
14 ;BMI is taken because of zero-bit
   ;total 15 (or 22 or even 29) 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. A delay of 22 CPU cycles would normally be enough for five bits to be shifted in. However, if the delay is caused by the presence of a zero-bit, then it behaves as though the delay were only 18 CPU cycles, which is enough for four bits to be shifted in. A delay of 29 CPU cycles is enough for seven bits to be shifted in. However, if the delay is caused by the presence of a second zero-bit, then it behaves as though the delay were only 21 CPU cycles, which is enough for five bits to be shifted in. In any case, the routine is written to discard a fixed number of regular bits, along with any zero-bits that are also present. Back to our stream, in binary, it would look like this:
11001111 11001111 **0** 10011110 11111101 **0** 11101101 10111011 11100110 10110110 11101101 11111011 **0** 11111100 11101011 11011111 11011110 11010011 11011001 11111111 11011001 11011101 **0** 11010111
with the seemingly redundant zero-bits in bold. However, by skipping the first three bits, the stream looks like this:
0 11110100 11110111 11101**0**11 10110110 11101111 10011010 11011011 10110111 1110110**1** 11111001 11010111 10111111 10111101 10100111 10110011 11111111 10110011 1011101**0** 11010111

The old zero-bits are still in bold, and the newly exposed zero-bit is in italics. We can see that the old zero-bits form part of the new stream. This decodes to F4 F7 (both ignored) EB B6 EF 9A DB B7 ED F9 D7 BF BD A7 B3 FF B3 BA. The trailing values are stored backwards, and the checksum is #$67. The low four bits (7) are the index into the table, and the values at offset 7 and 8 are #$D7 and #$F9.

A bit-copier that misses any of these zero-bits will write a track whose length and contents do not match the original.

7.10.16 Race conditions

Page 4 of the Software Control of the Disk][or IWM Controller document states that "The Disk][controller hardware will keep the ENABLE/ signal to its active low state for approximately one second after the execution of the motor off instruction, therefore read/write can be performed reliably within this period." So, a program can issue the motor off instruction, and then read sector data successfully for up to one second afterwards.

This behavior functions as a very nice anti-debugging mechanism, since single-stepping through the disk access code, after the motor-off instruction

has been issued, will cause the time period to be exceeded. Thus, the disk won't be readable at that time. Sherwood Forest uses this technique.

Page 4 of the Software Control of the Disk][or IWM Controller document also states that "...the program should verify that the motor is spinning by monitoring the change in data pattern read from the drive." That is to say, while the drive is spinning, the value will change. Once the drive stops spinning, the value will not change anymore.

Lady Tut uses this technique. It issues the motor-off instruction, and then reads continually from the drive until it sees two consecutive bytes of the same value. The program assumes at that point that the drive is no longer spinning. Periodically thereafter, the program reads from the QA switch of the Data Register, and compares the newly read value with the initially read value. If a different value is seen, then the program triggers a reboot.

In section 9-14 of Understanding the Apple][, Jim Sather says, "any even address could be used to load data from the data register to the MPU, although $C088 ... would be inappropriate." It might be considered inappropriate because of the one-second window noted previously, but that's exactly how the program Mr. Do! uses it. By reading from $C088, the program is able to issue the motor off instruction, and fetch the data at the same time. It is compact and useful for anti-debugging.

Faster pussycat

Another kind of race condition revolves around how quickly the data can be read from the disk. Borrowed Time, for example, reads an entire track in one revolution. In an interview for the Open Apple podcast, Rebecca Heineman says that she performs the decoding while the seek is in progress. While this is certainly possible, it would incur the significant overhead of having to store all 16 of the two-bit arrays—a total of 1.3kB! — before any decoding could occur. Of course, this is not what was done. Instead, each sector is read individually, but the denibbilisation is interleaved with the read. It means that the sector is decoded directly into memory, with only 86 bytes of overhead for a single two-bit array, and the use of two tables of 106 bytes and 256 bytes respectively. It is obviously fast enough to catch the next sector that arrives.

The code looks like this, after validating the data field prologue:

```
1  0946     LDY      #$AA
   ;zero  rolling  checksum
3  0948     LDA      #0
   094A     STA      $26
5  ;wait  for  nibble  to  arrive
   094C     LDX      $C0EC
7  094F     BPL      $94C
   ;index  into  table  of  offsets  of  structures
9  0951     LDA      $A00,X
   ;store  offset
11 0954     STA      $200,Y
   ;update  rolling  checksum
13 0957     EOR      $26
   ;fetch  86  times
15 0959     INY
   095A     BNE      $94A
17 095C     LDY      #$AA
   095E     BNE      $963
19 ;store  decoded  value
   0960     STA      $9F55,Y
21 ;wait  for  nibble  to  arrive
   0963     LDX      $C0EC
23 0966     BPL      $963
   ;update  rolling  checksum
25 0968     EOR      $A00,X
   ;fetch  structure  offset,  bits  0-1
27 096B     LDX      $200,Y
   ;merge  first  member  of  two-bit  structure
29 ;with  six-bit  value  to  recover  eight-bit
        value
   096E     EOR      $B00,X
31 ;loop  86  times
   0971     INY
33 0972     BNE      $960
   ;save  85th  decoded  value  for  last
35 0974     PHA
   ;clear  low  two  bits
37 0975     AND      #$FC
   0977     LDY      #$AA
39 ;wait  for  nibble  to  arrive
   0979     LDX      $C0EC
41 097C     BPL      $979
   ;update  rolling  checksum
43 097E     EOR      $A00,X
   ;fetch  structure  offset,  bits  2-3
45 0981     LDX      $200,Y
   ;merge  second  member  of  two-bit  structure
47 ;with  six-bit  value  to  recover  eight-bit
        value
   0984     EOR      $B01,X
49 ;store  decoded  value
   0987     STA      $9FAC,Y
51 ;loop  86  times
   098A     INY
53 098B     BNE      $979
   ;wait  for  nibble  to  arrive
55 098D     LDX      $C0EC
   0990     BPL      $98D
57 ;clear  low  two  bits
   0992     AND      #$FC
59 0994     LDY      #$AC
   ;update  rolling  checksum
61 0996     EOR      $A00,X
   ;fetch  structure  offset,  bits  4-5
```

```
63| ;offset −2 to account for Y+2
   | 0999    LDX     $1FE,Y
65| ;merge third member of two−bit structure
   | ;with six−bit value to recover eight−bit
   |     value
67| 099C    EOR     $B02,X
   | ;store decoded value
69| 099F    STA     $A000,Y
   | ;wait for nibble to arrive
71| 09A2    LDX     $C0EC
   | 09A5    BPL     $9A2
73| ;loop 84 times
   | 09A7    INY
75| 09A8    BNE     $996
   | ;clear low two bits
77| 09AA    AND     #$FC
   | ;update rolling checksum
79| 09AC    EOR     $A00,X
   | ;restore slot to X
81| 09AF    LDX     $2B
   | ;retry if checksum mismatch
83| 09B1    TAY
   | 09B2    BNE     $9BD
85| ;wait for nibble to arrive
   | 09B4    LDA     $C0EC
87| 09B7    BPL     $9B4
   | ;check only first epilogue byte
89| 09B9    CMP     #$DE
   | 09BB    BEQ     $9BF
91| 09BD    SEC
   | 09BE    .BYTE $24
93| 09BF    CLC
   | ;store 85th decoded value
95| 09C0    PLA
   | 09C1    LDY     #$55
97| 09C3    STA     ($44),Y
   | 09C5    RTS
```

The exact way in which the technique works is as follows. First, each of the two-bit values is read into memory, but instead of storing them directly, the values are used as an index into the 106-bytes table. The 106-bytes table serves two purposes. The first, in the context of the two-bit values, is as an array of offsets within the 256-bytes table. The second, in the context of the six-bit values, is as an array of pre-shifted values for the six-bit nibbles. The 256-bytes table is composed of groups of two-bit values in all possible combinations for each of the three positions in a nibble. To produce the eight-bit value, each of the pre-shifted six-bit values is ORed with the corresponding two-bit value. It is unknown why the 85th value is treated separately from the rest in that code; it could certainly be decoded at the same

time, saving five lines.

With the benefit of determination to improve it, and the ability to do so, I rewrote this loader to decode all of the bytes directly, reduced the size of the code, and made it even faster. I call it "0boot."[49] Then I reduced the overhead to just two bytes, if page $BF is not the destination. I call that one "q-boot."[50] The two tables are still 106 bytes and 256 bytes respectively. It might appear that the second table can be reduced to 192 bytes, since the other 64 bytes are unused. However, it is not possible for this algorithm, because the alignment is required to supply the pre-shifted values. If the table were reduced in size, then additional operations would be required to reproduce the effect of the shift, and which would take longer to execute than the time available before the next nibble arrived.

Interestingly, Heineman claims to have created and released the technique in 1980,[51] but it was apparently not until 1984 that she used it in a release herself. It certainly existed in 1980, though. Automated Simulations (which later became Epyx) included the technique with the programs Hellfire Warrior and Rescue At Rigel. In 1983, Free Fall Associates (founded by the co-founder of Automated Simulations, whose last name begins with "Free", and a programmer whose last name ends with "Fall") included the technique with the programs Murder on the Zinderneuf and Archon. (Apparently they took it with them, as Epyx did not use it again.) Also in 1983, Apple included the technique in ProDOS. In 1985, Brøderbund included the technique with the program Captain Goodnight. According to Roland Gustafsson, Apple supplied that code.[52]

[49]http://pferrie.host22.com/misc/0boot.zip

[50]http://pferrie.host22.com/misc/qboot.zip

[51]Personal communication

[52]Personal communication

Also interestingly, whoever included it in the Free Fall Associates programs either did not understand it, or just did not want to touch it—there, the loader has been patched to require page-aligned reads, but the code still performs the initialisation for arbitrary addressing. Twelve lines of code could have been removed from that version. The Interplay programs that use the technique also require page-aligned reads, but do not have the unnecessary initialisation code.

Quote of the day by Olivier Guinart, "It's ironic that the race condition would be used by a program called Borrowed Time."

7.11   Track-level protections

7.11.1   Track length

The length of a track might not be constant across all of the tracks on a disk. The speed of the drive is the primary reason: the faster the drive, the shorter the track (that is, fewer nibbles can be written) because of the larger gaps between the nibbles.

Wizardry determines the length of the track, by measuring the time between succeeding arrivals of sector zero, and then calculates the deviation from the expected value. This deviation value is applied to the length of several other tracks, and the result is compared against the expected lengths. If the length of the track is not within the range that is expected, then the program hangs. This protection cannot be reproduced by a sector-copier or track-copier, because they will discard the original data between the sectors, thus altering the length of the track. A bit-copier can usually reproduce this protection because it writes the entire track mostly as it appeared originally, so the track length is at least similar to the original.

7.11.2   Track positioning

The stepper motor in the Disk][system is composed of four magnets. To advance a whole track requires activating and deactivating two phases in the proper order, and with a sufficient delay, for each track to step. To step to a later track, the next phase must be activated while the other phases are deactivated. To step to an earlier track, the previous phase must be activated while the other phases are deactivated. As might be expected, activating and then deactivating only one of the phases will cause the stepper to stop half-way between two tracks. This is a half-track position. It is even possible to produce quarter-track stepping reliably, by performing the half-track stepping method, but with a smaller delay. Depending on the hardware, it can also be done by activating two of the phases, and then deactivating only one of them. This last technique is used by Spiradisc. (§7.11.9.)

The issue with half-track and quarter-track positioning is that data written to these partial track positions will cause signal interference with data written to the neighbouring half-track or quarter-track at the same relative position. To avoid unintentional cross-talk, data can be written to only part of the track such that there is no overlap, or placed at least three-quarters of a track apart. (The reliability of three-quarter tracks is questionable.)

The maximum amount of data that can be placed at partial-track intervals is proportional to the stepping—a quarter of a track for each of four consecutive quarter-tracks, half of a track for each of two consecutive half-tracks, or a full track for consecutive three-quarter-tracks. There can be a significant performance hit to access the data, too—it requires an almost complete rotation to reach the start of the data on subsequent tracks if the maximum density is used, because the seek time is long enough that the start will be missed on the first time around. As a result, the most common amount that is used is only a quarter of the track, and placed far enough around the track that the read can be performed almost continuously. Programs that make use of partial tracks usually include a standard format of individual sectors, so the only trick to the protection is the location of the data on the disk.

Agent USA uses the half-track technique with five sectors per track.

Championship Lode Runner uses an alternating quarter-track technique with just two sectors per track but of twice the size. While loading, the access alternates between the neighbouring quarter-tracks, resulting in the drive "chattering", but allowing the sectors to be spaced only half of a rotation apart. In both cases of the programs here, it results in an extremely fast load time because of the reduced head movement.

In this case, the protection is the use of partial tracks. Copy programs which do not copy the partial tracks (and copying partial tracks is not the default behavior) will fail to reproduce the protection.

7.11.3 Synchronised tracks

If the approximate rotation speed of the drive is known, then it becomes possible to place sectors at specific locations on tracks, such that they have a special position relative to sectors on other tracks. This technique is identical to synchronized sectors, except that it spans tracks, making it even more difficult to reproduce, because it is difficult to determine the relative position of sectors across tracks. Unlike "spiral tracking" (§7.11.4), this technique limits itself to checking for the existence of particular sectors, rather than actually reading them.

Blazing Paddles uses this technique. Once it finds sector zero on track zero, as a known starting point, it seeks to track one, reads the address field of the next sector to arrive, and then compares it to an expected value. If the proper sector is found, then the program seeks to track two, reads the address field of the next sector to arrive, and compares it to an expected value. If the proper sector is found, then the program seeks to track three. This is repeated over eight tracks in total. It means that the original disk has one sector placed at a specific location on each of eight consecutive tracks, relative to sector zero of track zero, such that it factors in how much the disk rotates during the time that the controller takes to move the head from track zero. It also supports slight variations in rotation speed, such that the read can begin anywhere after the address field for the previous sector, without failing the protection.

7.11.4 Track spiralling

"Track spiralling" or "spiral tracking" is a technique whereby the data is placed in partial-track intervals, but treated as a complete track. By measuring the time to move the head to a partial-track, the position on the track can be known, such that the next sector to be read will have a predictable number, and therefore can be read without validation, once the start of the sector is found. A copy of the disk will not place the data at the same relative position, causing the protection to fail. The stepping in spiral tracking goes in only one direction. A visualisation of the data access would look like a broken spiral, hence the name.

One major problem with spiral tracking is that variations in rotation speed can result in the read missing its queue and not finding the expected sector. For 30 years, I believed a claim[53] that the program Captain Goodnight uses this technique. It doesn't. The Observatory uses a spiral pattern for faster loading, but still verifies the sector number first. However, the program LifeSaver uses true spiral tracking.

7.11.5 Track arcing

"Track arcing" uses the same principle as spiral tracking, but instead of stepping in only one direction, it reaches a threshold and then reverses direction.

7.11.6 Track mirroring

Track mirroring should be placed conceptually between synchronized tracks and spiral tracking. As

---

[53] From a cracker whose crack-screens were displayed only by pressing a particular key-sequence during the boot. They were known as "Hidden Pages" (Imagine that—a cracker who didn't want to brag openly!) Both of the programs Captain Goodnight and Where In The World Is Carmen Sandiego (first release) use alternating quarter-tracks—the same technique as in the program Championship Lode Runner. (The former two were released within a year of the latter one.) The sectors are placed in a N/S/E/W orientation on the first two tracks, a NW/SE/NE/SW orientation on the next two tracks, and then back to the N/S/E/W orientation on the next two tracks, and so on. The loader will allow an entire revolution to pass, if necessary, in order to find the requested sector. The tracks are synchronized, however, because they must be to avoid cross-talk. (§7.11.7.)

with synchronized tracks, it expects a particular sector to be found after stepping across multiple tracks. As with spiral tracking, it reads the sector data. However, unlike spiral tracking, it verifies that the contents of that sector match exactly the contents of all of the other sectors that are synchronized similarly across the tracks.

The Toy Shop uses this technique. It reads three consecutive quarter-tracks in RWTS18 format, and verifies that they all fully readable and have a valid checksum. This is possible only because they are identical in their content and position. The contents of the last quarter-track are used to boot the program. A funny thing occurs when the program is converted to a NIB image: the protection is defeated transparently, because NIB images do not support partial tracks, so the attempt to read consecutive quarter-tracks will always return identical data, exactly as the protection requires.

Pinball Construction Set uses this technique. It reads a sector then activates a phase to advance the head, and then proceeds to read a sector *while the head is moving*. The head continues to drift over the track while the sector is being read. After reading the sector, the program deactivates the phase, reads another sector, and then completes the move to the next track. Once there, it reads a sector. It activates a phase to retreat the head, and then performs the same trick in reverse, until the start of the track is reached again. It performs this sequence four times across those two tracks, which makes the drive hiss. The program is able to read the sector as continuous data because the disk has consecutive quarter-tracks that are identical in their content and position.

7.11.7 Cross-talk

While cross-talk is normally something to be avoided, it can serve as a copy-protection mechanism, by intentionally allowing it to occur. It manifests itself in a manner similar to the effect of having excessive consecutive zero-bits being present in the stream, where reading the same stream repeatedly will yield different values. The lack of such an effect indicates the presence of a copy.

7.11.8 More tracks

Many disk drives had the ability to seek beyond track 34, and many disks also carried more than 35 tracks. However, since DOS could not rely on the presence of either of these things, it did not offer support for them. Some copy programs did not support the copying of additional tracks for the same reason. Of course, programmers who did not use DOS had no such limitation. While the actual number of available tracks could vary up to 40 or even 42, it was fairly safe to assume that at least one track existed, and could be read by direct use of the disk drive.

Faial uses this technique to place data on track 35.

7.11.9 SpiraDisc

No description of copy-protection techniques could be complete without including SpiraDisc. This program was a protection technology that introduced the idea of spiral tracking, though the implementation is not spiral tracking as we would describe it today. It is, in fact, a precise placement of multiple sectors on quarter-tracks, such that there is no cross-talk while reading them, but without a specific order. The major deviation from the current idea of spiral tracking is that there is no synchronization of the sectors beyond avoiding cross-talk. The program will allow a complete rotation of the disk to occur, if necessary, while searching for the required sector.

The first-stage boot loader is a single sector that is "4-and-4" encoded, and 768 bytes long. The second stage loader is composed of ten regular sectors that are "6-and-2" encoded. They are read one by one—there is no read-scattering here to speed up the process. Thereafter, reads use an alternative nibble table—all of the values from `#$A9-FF` from our first table. These values might have been chosen because they provide the least sparse array when used as indexes.

The encoding is not "6-and-2", either, it is "6-and-0" encoding. This requires 344 bytes per sector, instead of the regular 342 bytes. The decoder overwrites the addresses `$xxAA` and `$xxAB` (the program supports only page-aligned reads) twice in order to compensate for the additional bytes. The decoding is interleaved, so there is no denibbilisation pass.

The "6-and-0" encoding works by using the six-bit nibble as an alternating index into one of the arrays of six-bit or two-bit values. The code is both much faster (no fetching of the two-bit array) and much smaller (two-thirds of the size) than the one described in Race Conditions,(§7.10.16) but the decoding tables occupy 1.5kb of memory. The memory layout might have been chosen to avoid a timing

penalty due to page-crossing accesses. However, the penalty has no effect on the performance of the routine because the code must still spend time waiting for the bytes to arrive from disk. Therefore, the tables could have been combined into a 512-byte region instead, which is a closer match to the memory usage of the routine described in Race Conditions.

A Spiradisc-protected disk uses four sectors per track, but since the track stepping is quartered, the data density is equivalent to a single 16-sector track. Each sector has a unique prologue value to identify itself. When a read is requested, if a sector cannot be found on the current track, then the program advances the drive head by one quarter-track, and then attempts the read again. If the read fails again, then the program retreats the drive head by one quarter-track, and then attempts the read again. If the read still fails, then the program retreats the drive head by another quarter-track, and then attempts the read again. If the read fails at this point, then the disk is considered to be corrupted.

Given the behaviour of the read request, the data might not be stored on consecutive quarter-tracks. Instead, they might zig-zag across a span of up to three quarter-tracks. This is another deviation from the idea of spiral tracking. By coincidence, the movement is very similar to the one in the program Captain Goodnight and other Brøderbund titles.

Copying a SpiraDisc-protected disk is difficult because of the potential for cross-talk which would corrupt the sectors when they are read back. However, images produced by an E.D.Dċard will work in emulators, if the copy parameters are set correctly.

When run, the program decodes selected pages of itself, based on an array of flags, and also re-encodes those pages after use, to prevent dumping from memory. The decoding is simply an exclusive-OR of each byte with the value #$AC, exclusive-ORed with the index within the page.

At start-up, the program profiles the system: scanning the slot device space, and records the location of devices for which the first 17 bytes are constant (that is, they return the same value when read more than once), and which do not have eight bytes that match the first one within those 17 bytes. For example, Mockingboard has memory-mapped I/O space in that region, which are mostly zeroes. The program calculates and stores a checksum for slot devices which pass this check. The store was supposed to happen only if the checksum did not match certain values, but the comparison is made against a copyright string instead of an array of checksums. The first time around, all values are accepted. During subsequent profiling, the value must match exactly.

The program checks if bank one is writable, after attempting to write-enable it, and sets a flag based on the result. The program checksums the F8 and F0 ROM BIOS codes, watches for particular checksums, and sets flags based on the result. The original version of the program (as seen in 1981, used on the program Jawbreaker) actually *required* that the ROM BIOS code match particular checksums—either the original Apple][or the Apple][+—otherwise the program simply wiped memory and rebooted. (This prevented protected programs from running on the Apple][e or the Apple][c.) The no-doubt numerous compatibility problems that resulted from this decision led to the final check being discarded (as seen in 1983, used on the program Maze Craze Construction Set, but quite possibly even earlier), though the rest of the profiling remains. However, having even one popular title that didn't work on more modern machines was probably sufficient to turn publishers entirely off the use of the program.

The program probes all of memory by writing a zero to every second byte. However, it skips pages #0, #2, #4-7, and #$A8-C0, meaning that it writes data to all slot devices, with unpredictable results. The program also re-profiles the system upon receiving each request to read tracks. This re-profiling is intended to defeat memory dumps that are produced by NMI cards, and which are then transferred to another machine, as the second machine might have different hardware options.

The program also checksums the boot PROM prior to disk reads, and requires that it matches one particular checksum—that of the Disk][system—otherwise the program wipes memory and reboots. (This prevents protected programs from running on the Apple][GS.)

Interestingly, despite all of the checks of the environment, the program does not protect itself against tampering, other than using encoded pages. The memory layout is data on pages #$A8-B1, and code on pages #$B2-BF. The data pages are very sparse, leaving plenty of room for a boot tracer to intercept execution and disable protections.

The program uses a quarter-track stepping algorithm that activates two phases, and then deactivates only one of them. According to Roland

Gustafsson, this stepping technique allows for more precise positioning of the drive head, but it does not work on Rana drives. It was for this reason that he used the reduced-delay technique instead. (§7.11.2.) The reduced-delay technique is apparently the only one which works on an Apple][c, as well. Spiradisc predated the Apple][c by about two years, so it was just bad luck that an incompatible technique was chosen.

7.12 Illegal opcodes

The 6502 CPU has 151 documented instructions. There are quite a few additional instruction encodings for which the results could be considered useful, if the side-effects (e.g. memory and/or register corruption, or long execution time) were also acceptable. In some cases, the instructions were used to obfuscate the meaning of the code, since they would not be disassembled correctly. Some of these undocumented instructions were replaced in the 65C02 CPU with documented instructions with different behaviors, and without the unfortunate side-effects. In some cases, the code that used the undocumented instructions was not affected because the results of the undocumented instructions were discarded, and the documented replacement did not introduce especially unwanted behavior. Note that the instructions that were not replaced will cause the 65C02 CPU to hang.

The Datasoft version of the program Dig Dug uses this technique. It begins with an instruction which used to behave as a two-byte `NOP`, but which is now a zero-page `STZ` instruction. Since the program does not make use of the zero-page at that time, the store has no side-effects. It looks like this in 6502 mode:

```
  0801    74              ???
2 0802    4C B0 58    JMP    $58B0
```

In 65C02 mode, the same machine code interpreted differently.

```
  0801    74  4C      STZ    $4C
2 0803    B0  58      BCS    $85D
```


Beer Run uses this technique, but was unfortunate enough to choose an instruction which was not defined on the 65C02 CPU, so the program does not work on a modern machine. The code is run with the carry set much earlier in the flow, as a side-effect of executing a routine in the ROM BIOS. It is possible that the authors were not even aware of the fact.

```
  051B      LDX    #$00
2 ...
  051F      LDA    #$00
4 0521      STA    $00
  ...
6 ;FF  00  00
  0525      ISC    $0000,X
```

which, when executed, does this:

```
1 INC    $0000,X
  SBC    $0000,X
```

X is zero, so `$00` is first incremented to `#$01`, and then subtracted from `A`. `A` is zero before the subtraction, so it becomes `#$FF`. The resulting `#$FF` is used as a key to decipher some values later.

7.13 CPU bugs(!)

The original 6502 CPU had a bug where an indirect `JMP` (`xxFF`) could be directed to an unexpected location because the MSB will be fetched from address `xx00` instead of page `xx+1`. Randamn relies on this behavior to perform a misdirection, by placing a dummy value at offset zero in page `xx+1`, and the real value at address `xx00`.

While not a bug, but perhaps an undocumented feature—the breakpoint bit is always set in the status register image that is placed on the stack by the `PHP` instruction. Lady Tut relies on this behavior to derive a decryption key.

There is also a class of alternative behaviours between the 6502 and the 65C02 CPUs, particularly regarding the Decimal flag. For example, the following sequence will yield different values between

the two CPUs: $1B on a 6502, and $0B on a 65C02. These days, it would be used as an emulator detection method. Try it in your favorite emulator to see what happens.

```
  SED
2 SEC
  LDA #$20
4 SBC #$0F
```


7.14   Magic stack values

One way to obfuscate the code flow is through the use of indirect transfers of control. Rescue At Rigel fills the stack entirely with the sequence #$12 #$11 #$10, and then performs an RTI without setting the stack pointer to a constant value. Of course, it works reliably.

Since there are only three values in the sequence, there should be only three cases to consider. If the stack pointer were #$F6 at the time of executing the RTI instruction, then this causes the value #$12 and $1011 to be fetched from $1F7. If the stack pointer were #$F7 at the time of executing the RTI instruction, then this causes the value #$11 and $1210 to be fetched from $1F8. If the stack pointer were #$F8 at the time of executing the RTI instruction, then this causes the value #$10 and $1112 to be fetched from $1F9. The program has an RTS instruction at the first and last of those locations. That yields two more cases to consider. The RTS at $1011 transfers control to $1112+1. The RTS at $1112 transfers control to $1210+1. That leaves one more case to consider. The program has an RTS instruction at $1113. The RTS at $1113 transfers control to $1211. So, both $1210 and $1211 are reachable this way. Both addresses contain a NOP instruction, to allow the code to fall through to the real entrypoint.

Note the phase "there should be." There is one special case. The remainder of 256 divided by three is one. What is in that one byte? It's the value #$10. So the first and last byte of the stack page is #$10,

introducing an additional case. If the stack pointer were #$FD at the time of executing the RTI instruction, then this causes the value #$11 and $1010 to be fetched from $1FE. The program has an RTS instruction at $1010. The RTS at $1010 transfers control to $1112+1. The RTS at $1113 transfers control to $1211.

That's not all! We can construct an even longer chain. If the stack pointer were #$F9 at the time of executing the RTI instruction, then this causes the value #$12 and $1011 to be fetched from $1FA. The RTS at $1011 transfers control to $1112+1, but the RTS at $1113 causes the stack pointer to wrap around. The CPU fetches both #$10 values, so the RTS at $1113 transfers control to $1010+1. The RTS at $1011 transfers control again to $1112+1. The RTS at $1113 finally transfers control to $1211.

Championship Lode Runner has a smaller chain. It uses only two values on the stack: $3FF and $400. An RTS transfers control to $3FF+1. The program has an RTS at $400. The RTS at $400 transfers control to $400+1, the real entrypoint.

7.15   Obfuscation

7.15.1   Anti-disassembly (aka WTF?

This technique is intended to prevent casual reading of the code—that is, static analysis, and specifically targeting linear-sweep disassemblers—by inserting dummy opcodes into the stream, and using branch instructions to pass over them. At the time, recursive-descent disassembly was not common, so the technique was extremely effective.

Wings of Fury uses this technique, even for its system detection. The initial disassembly follows, with undocumented instructions such as RLA.

```
   9600    ORA    (0,X)
 2 9602    LDY    #$10
   9604    BPL    $9616
 4 9606    RLA    ($10,X)
   9608    NOP
 6 960A    BEQ    $95AC
   960C    NOP
 8 960E    STY    $84
   9610    STY    $18
10 9612    CLC
   9613    CLC
```

63

```
12  9614    BNE     $961C
    9616    CLC
14  9617    CLC
    9618    BNE     $960B
16  961A    SRE     ($51),Y
    961C    STY     $C009
18  961F    STX     $20,Y
    9621    ORA     ($10),Y
20  9623    CPX     $84
    9625    STA     $C008
22  9628    BEQ     $9672
    962A    LDA     $C088,X
24  962D    ORA     ($18),Y
    962F    ORA     ($10),Y
26  9631    ASL
    9632    LDX     #$27
28  9634    ASL
    9635    ASL
30  9636    LDY     #$10
    9638    BPL     $9630
32  963A    BRK
    963B    JMP     $93BD
34  963E    TYA
    963F    STA     $400,X
36  9642    BNE     $964C
    9644    BRK
```

```
    ;turn  off  the  drive
27  962A    LDA     $C088,X
    ;dummy  instruction
29  962D    ORA     ($18),Y
    ;dummy  instruction  masks  real  instruction
31  962F    ORA     ($10),Y
    ;dummy  instruction  in  first  pass
33  ;opcode  parameter  in  second  pass
    9631    ASL
35  ;length  of  error  message
    9632    LDX     #$27
37  ;two  dummy  instructions
    9634    ASL
39  9635    ASL
    9636    LDY     #$10
41  ;unconditional  branch
    ;because  Y  is  positive
43  9638    BPL     $9630
    963A    BRK
45  963B    JMP     $93BD
    963E    TYA
47  963F    STA     $400,X
    9642    BNE     $964C
49  9644    BRK
```

Upon closer examination, we see the branch instruction at `$9604` is unconditional, because the value in the `Y` register is positive. That leads to the branch at `$9618`. This branch is also unconditional, because the value in the `Y` register is not zero. That takes us into the middle of an instruction at `$960B`, and requires a second round disassembly:

```
1   ;store  #$64  at  $84
    960B    LDY     #$64
3   960D    STY     $84
    ;four  dummy  instructions
5   960F    STY     $84
    9611    CLC
7   9612    CLC
    9613    CLC
9   ;unconditional  branch
    ;because  Y  is  not  zero
11  9614    BNE     $961C
    ...
13  ;switch  to  auxiliary  memory  bank,  if
        available
    961C    STY     $C009
15  ;store  alternative  value  at  $84  ($20+#$64=
        $84)
    961F    STX     $20,Y
17  ;dummy  instruction
    9621    ORA     ($10),Y
19  ;compare  the  two  values
    ;will  differ  in  64kb  environment
21  9623    CPX     $84
    ;switch  to  main  memory  bank
23  9625    STA     $C008
    ;branch  if  128kb  memory  exists
25  9628    BEQ     $9672
```

A third round disassembly:

```
1   ;unconditional  branch
    ;because  Y  is  positive
3   9630    BPL     $963C
    ...
5   ;message  text
    963C    LDA     $9893,X
7   ;write  to  the  screen
    963F    STA     $400,X
9   ;unconditional  branch
    ;because  A  is  not  zero
11  9642    BNE     $964C
```

The obfuscated code only gets worse from there, but the intention is clear already.

7.15.2   Self-modifying code

As the name implies, this technique relies on the ability of code to modify itself at runtime, and to have the modified version executed. A common use of the technique is to improve performance by updating an address with a loop during a memory copy, for example. However, from the point of view of copy-protection, the most common use is to change the code flow, or to act as a light encoding layer. Self-modifying code can be used to interfere with debuggers, because a breakpoint that is placed on the modified instruction might be overwritten directly, thus removing it, and resulting in uncontrolled execution; or turned into an entirely unrelated (and

possibly meaningless or even harmful) instruction, with unpredictable results.

Aquatron hides its protection check this way. The initial disassembly looks like this, complete with undocumented instructions such as ISB:

#	Addr	Op	Operand
1	9600	DEC	$9603
	9603	ISB	$9603
3	9606	LDA	$9628
	9609	EOR	#$C9
5	960B	BNE	$960E
	960D	JSR	$288D
7	9610	STX	$18 ,Y
	9612	BNE	$9615
9	9614	JMP	$29A0
	9617	TYA	
11	9618	BCC	$961B
	961A	JSR	$59
13	961D	STX	$99 ,Y
	961F	BRK	
15	9620	STX	$C8 ,Y
	9622	BNE	$9617
17	9624	TYA	
	9625	BPL	$9628
19	9627	JMP	$2960

Upon closer examination, we see references to instructions at "hidden" offsets, and of course, the direct modification of the instruction at `$9603`.

Second round disassembly:

```
1  9600       DEC      $9603
   ;-> INC $9603
3  ;undo self-modification and continue
   9603       ISB      $9603
5  9606       LDA      $9628
   9609       EOR      #$C9
7  ;unconditional branch
   ;because A is not zero
9  960B       BNE      $960E
   960D       .BYTE    $20
11 ;replace instruction below
   960E       STA      $9628
13 9611       CLC
   ;unconditional branch
15 ;because A is not zero
   9612       BNE      $9615
17 9614       .BYTE    $4C
   9615       LDY      #$29
19 9617       TYA
   9618       BCC      $961B
21 961A       .BYTE    $20
   ;decode and store
23 961B       EOR      $9600,Y
   961E       STA      $9600,Y
25 9621       INY
   9622       BNE      $9617
27 9624       TYA
   ;unconditional branch
29 ;because Y is positive
   9625       BPL      $9628
31 9627       .BYTE    $4C
   ;self-modified by $960E to $A9 on first pass
33 ;restored to $60 on second pass
   9628       RTS
35 ;decoded by $961B-9620 on first pass
   ;re-encoded on second pass
37 9629       .BYTE    $29
```

Now we can see the decryption routine. It decodes the bytes at `$9629-96FF`, which contained a check for a sector with special format. If the checked passes, then the routine at `$9600` is run again, which reverses the changes that had been made — the bytes at `$9629-96FF` are encoded again, and the routine exits via the RTS instruction at `$9628`.

7.15.3   Self-overwriting code

When self-modification is taken to the extreme, the result is self-overwriting code. There, the RWTS routine reads sector data over itself, in order to change the execution behavior, and potentially remove user-defined modifications such as breakpoints or detours. LifeSaver uses this technique. The

loader enters a loop which has no apparent exit condition. Instead, the last sector to be read from disk contains an identical copy of the loader code, except for the last instruction which branches to a new location upon completion. When combined with a critically timing-dependent technique, such as reading a sector while the head is moving, it becomes extremely difficult to defeat.

7.15.4   Encryption and compression

Encryption (or, more correctly, enciphering) of code was a popular technique, but the keys were always very weak. The enciphering usually consisted of an exclusive-OR of the byte with a fixed key. In some cases, the key was a rolling value taken from the byte just deciphered. In some rarer cases, multiple keys were used.

Goonies uses a rotate operation. However, since the 6502 CPU does not have a plain rotate instruction—only rotate with carry — the program must set the carry bit correctly prior to the operation. The program does it this way:

```
1  ;save value
   0405       PHA
3  ;extract carry bit
   0406       LSR
5  ;restore value
   0407       PLA
7  ;rotate with carry
   0408       ROR
```

Compression of graphics was necessary to reduce the size of the data on disk, and to decrease load times, since the reduced disk access more than made up for the time spent to decompress the graphics. The most common compression technique was Run-Length Encoding (RLE), using a stream derived from every second horizontal byte, or vertical columns. More advanced compression, such as something based on Lempel-Ziv, was generally considered to be too slow to use.

Perhaps based on the assumption that LZ-based compression was too slow, compression of code seems to have been entirely absent until recently—all

of my releases use my decompressor for aPLib[54], for an almost exact or even slightly reduced load time, which shows that the previous assumption was quite wrong. Others have had success with my decompressor for LZ4[55] when used for graphics. A more recent LZ4-based project is also showing promise.[56]

7.16 Virtual machines

One of the most powerful forms of obfuscation is the virtual machine. Instead of readable assembly language that we can recognise, the virtual machine code replaces instructions with bytes whose meaning might depend on the parameters that follow them. Electronic Arts were famous for their use of pseudo-code (p-code) to hide the protection routines in programs such as Archon and Last Gladiator. That virtual machine was even ported to the Commodore 64 platform.

Last Gladiator uses a top-level virtual machine that has 17 instructions. The instructions look like this:

```
   00    JMP
2  01    CALL NATIVE
   02    BEQ
4  03    LDA IMM
   04    LDA ABSOLUTE
6  05    JSR
   06    STA ABSOLUTE
8  07    SBC IMM
   08    JMP NATIVE
10 09    RTS
   0A    LDA ABSOLUTE, A ;p-code A register
12 0B    ASL
   0C    INC ABSOLUTE
14 0D    ADC ABSOLUTE
   0E    XOR ABSOLUTE
16 0F    BNE
   10    SBC ABSOLUTE
18 11    MOVS
```

It has the ability to transfer control into 6502 routines, via the instructions that I named "call native" and "jmp native." The parameters to the instructions were XORed with different values to make the disassembly even more difficult. Since the virtual machine could read arbitrary memory, it was used to access the soft-switches, in order to turn the drive on and off. Once past the first virtual machine, the program ran a second one. The second

virtual machine is interesting for one particular reason. While it looks identical to the first one, it's not exactly the same. For one thing, there are only 13 instructions. For another, two of them have swapped places:

```
   0A    INC ABSOLUTE
2  0B    nothing
   0C    LDA ABSOLUTE, A ;p-code A register
```

HARD HAT MACK

These two engines were not the only ones that Electronic Arts used, either. Hard Hat Mack uses a version that had twelve instructions.

```
1  00    JMP
   01    CALL NATIVE
3  02    BEQ
   03    LDA IMM
5  04    LDA ABSOLUTE
   05    JSR
7  06    STA ABSOLUTE
   07    SBC IMM
9  08    JMP NATIVE
   09    RTS
11 0A    LDA ABSOLUTE, A ;p-code A register
   0B    ASL
```

Following that virtual machine was yet another variation. This one has only eleven instructions. Nine of the instructions are identical in value to the previous virtual machine. The differences are that "ASL" is missing, and the "LDA ABSOLUTE, A" instruction is now "INC ABSOLUTE."

However, in between those two virtual machines was an entirely different virtual machine. It is a stack-based engine that uses function pointers instead of byte-code. It looks like this, if you'll forgive handler address in place of names I wasn't able to identify.

```
   9DF2    .WORD xsave_retpc
2  9DF4    .WORD xpush_imm
   9DF6    .WORD $95FF
4  9DF8    .WORD xpush_imm
   9DFA    .WORD $A600
6  9DFC    .WORD xchkstk_vars
   9DFE    .WORD xbeq_rel
8  9E00    .WORD 4
   9E02    .WORD xdo_copy_prot
10 9E04    .WORD xjmp_retpc
```

---

[54] http://pferrie.host22.com/misc/aplibunp.zip

[55] http://pferrie.host22.com/misc/lz4unp.zip

[56] https://github.com/fadden/fhpack

This virtual machine is Forth. Amnesia, including its copy-protection (What You Know style), was written entirely in Forth. The Toy Shop used another virtual machine, which combined byte-code and function pointers, depending on which function was called, and all mixed freely with native code. Its identity is not known.

Of course, the most famous of all virtual machines is the one inside Pascal, an ancestor of Delphi that was very widely used in the eighties. Wizardry is perhaps the most well-known Pascal program on the Apple][system, and the Pascal virtual machine made it a simple task to port the program to other platforms. The advantage of a virtual machine is that only the interpreter must be ported, rather than the entire system. Since the language is much higher-level than assembly language, it also allows for a faster development time. It also makes de-protecting a program much harder.

7.17   ROM regions

The Apple][ROM BIOS is full of little routines whose intention is clear, but whose meaning can be changed depending on the context. That leads into an interesting area of obfuscation and indirection. For our first example, there is a routine to save the register contents. It is used by the ROM BIOS code when a breakpoint occurs. It has the side-effect of returning the status register in the `A` register. That allows a program to replace the instruction pair `PHP`; `PLA` with the instruction `JSR $FF4A` for the same primary effect (it has the side-effect of altering several memory locations), but one byte larger.

For our second example, there is a routine to clear the primary text screen. Since the Apple][has a text and graphics mode that share the same memory region, there is one routine for clearing the screen while in text mode, and another for clearing the screen while in graphics mode. However, it is possible to use the graphics routine to clear the screen even while in text mode. That allows a program to replace `JSR $FC58` with `JSR $F832` for the same major effect. (It has the side-effect of altering several memory locations.)

For our third example, there is a routine to compare two regions of memory. It is used primarily to ensure that memory is functioning correctly. However, it can also be used to detect alterations that as those produced by a user attempting to patch a program. All that is required is to set the parameters correctly, like this:

```
    LDA     #>beghi
 2  STA     $3D
    LDA     #<beglo
 4  STA     $3C
    LDA     #>endhi
 6  STA     $3F
    LDA     #<endlo
 8  STA     $3E
    LDA     #>cmphi
10  STA     $43
    LDA     #<cmplo
12  STA     $42
    JSR     $FE36
```

For our fourth example, there is an `RTS` instruction at a known location. A jump to this instruction will simply return. It is usually used to determine the value of the Program Counter. However, it can just as easily be used to hide a transfer of control, taking into account that the destination address must be one less than the true value, like this to jump to $200:

```
 1  LDA     #$01
    PHA
 3  LDA     #$FF
    PHA
 5  JMP     $FF58
```

And so on. The first three examples are taken from Lady Tut, though in the third example, the parameters are also set in an obfuscated way, using shifts, increments, and constants. The fourth is taken from Mr. Do!.

7.18   Sensitive memory locations

There are certain regions in memory, in which modifications can be made which will cause intentional side-effects. The side-effects include code-destruction when viewed, or automatic execution in response to any typed input, among other things. The zero-page is a rich source of targets, because it is shared by so many things.

The most commonly altered regions follow.

7.18.1   Scroll window

When the monitor is active, the scrollable region of the screen can be adjusted to allow "fixed" rows and/or columns. The four locations, left ($20), width ($21), top ($22), and bottom ($23) can also be adjusted. A program can protect itself from debugging attempts by altering these values to make a

very small window, or even to cause overlapping regions that will cause memory corruption if scrolling occurs.

7.18.2   I/O vectors

There are two I/O vectors in the Apple][, one for output—CSW ($36-37), and one for input—KSW ($38-39). CSW is invoked whenever the ROM BIOS routine COUT is called to display text. KSW is invoked whenever the ROM BIOS routine RD-KEY is called to wait for user input. Both of these vectors are hooked by DOS in order to intercept commands that are typed at the prompt. Both of these vectors are often forcibly restored to their default values to unhook debuggers. They are sometimes altered to point to disk access routines, to prevent user interaction. Championship Lode Runner uses the hooks for disk access routines in order to load the level data from the disk.

7.18.3   Monitor
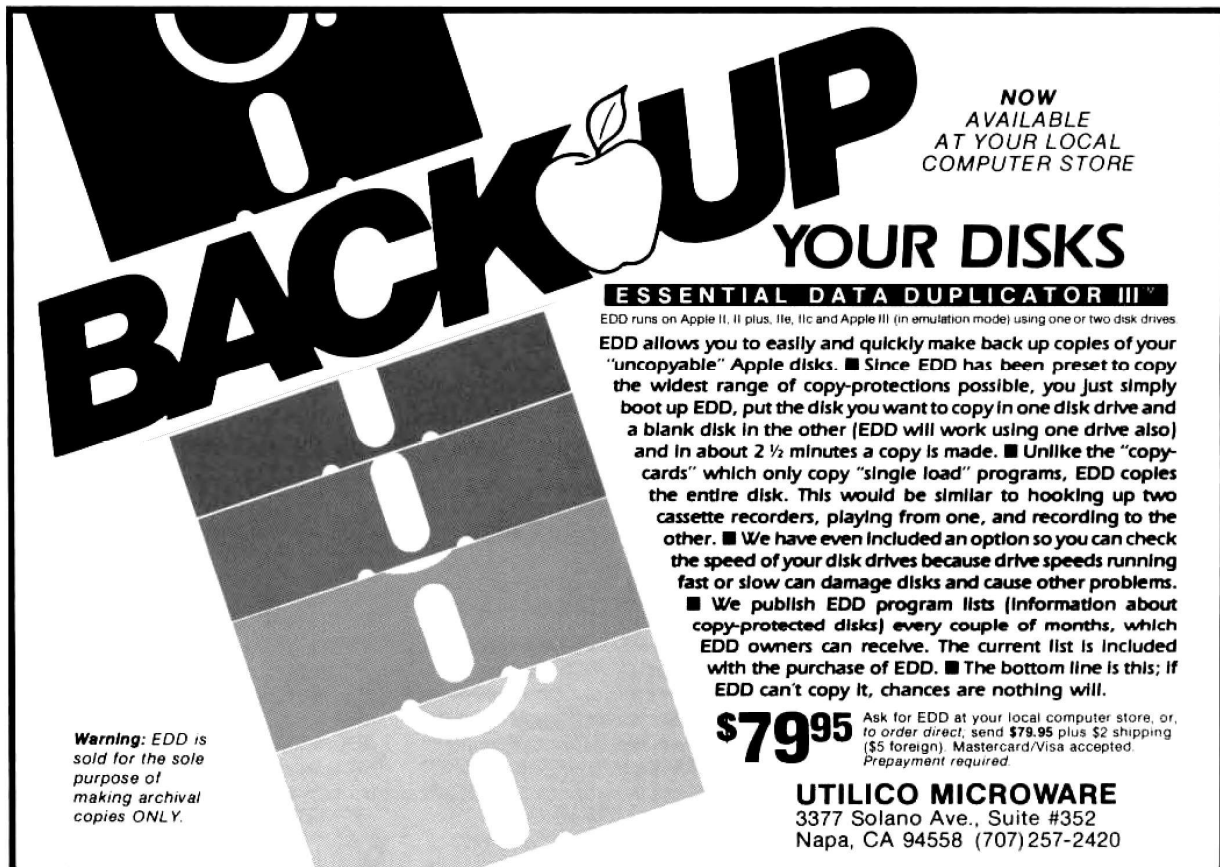
The monitor prompt allows a user to view and alter memory, and execute subroutines. It uses several zero-page addresses in order to do this. Anything that is stored in those locations ($31, $34-35, $3A-43, $45-49) will be lost when the monitor becomes active. In addition, the monitor uses the ROM BIOS routine RDKEY. RDKEY provides a pseudo-random number generator, by measuring the time between keypresses. It stores that time in $4E-4F.

Falcons uses address $31 to hold the rolling checksum, and checks if $47 is constant after initialising it.

Classmate uses addresses $31 and $4E to hold two of the data field prologue bytes.

7.18.4   The "LOCK" mystery

There is a special memory location in Applesoft ($D6) which is named the "AppleSoft Mystery Pa-

rameter" in What's Where In The Apple. It is also named "LOCK" in the Applesoft Internals disassembly, which gives a better idea of its purpose. When set to #$80, all Applesoft commands are interpreted as meaning "RUN." This prevents any user interaction at the Applesoft prompt. Tycoon uses this technique.

7.18.5 Stack

The stack is a single 256-bytes page ($100-1FF) in the Apple][. Since the standard Apple][environment does not have any source of interrupts, the stack can be considered to be a well-defined memory region. This means that code and data can be placed on the stack, and run from there, without regard to the value of the stack pointer, and modifications will not occur unexpectedly. (The effect on the stack of subroutine calling is an expected modification.) If an interrupt occurred, then the CPU would save the program counter and status register on the stack, thus corrupting the code or data that existed below the current stack pointer. (The corruption can even be above the stack pointer, if the stack pointer value is low enough that it wraps around!) Correspondingly, any user interaction that occurs, such as breaking to the prompt, will cause corruption of the code or data that exist below the current stack pointer. Choplifter uses this technique.

7.18.6 Stack pointer

Since the standard Apple][environment does not have any source of interrupts, the stack pointer can be considered to be a register with well-defined value. This means that its value remains under program control at all times and that it can even be used as a general-purpose register, provided that the effect on the stack pointer of subroutine calling is expected by the program. Beer Run uses this technique.

LifeSaver also uses this technique for the purpose of obfuscating a transfer of control—the program checksums the pages of memory that were read in, and then uses the result as the new stack pointer, just prior to executing a "return from subroutine" instruction. Any alteration to the data, such as the insertion of breakpoints or detours, results in a different checksum and unpredictable behavior.

7.18.7 Input buffer

The input buffer is a single 256-bytes page ($200-2FF) in the Apple][. Code and data can be placed in the input buffer, and run from there. However, anything that the user types at the prompt, and which is routed through the ROM BIOS routine GETLN ($FD6A), will be written to the input buffer. Any user interaction that occurs, such as breaking to the prompt, will cause corruption of the code in the input buffer. Karateka uses this technique.

7.18.8 Primary text screen

The primary text screen is a set of four 256-bytes pages ($400-7FF) in the Apple][. Code and data can be placed in the text screen memory, and run from there. The visible screen was usually switched to a blank graphics screen prior to that occurring, to avoid visibly displaying garbage, and perhaps causing the user to think that the program was malfunctioning. Obviously, any user interaction that occurs through the ROM BIOS routines, such as breaking to the prompt and typing commands, will cause corruption of the code in the text screen. Joust uses this technique to hold essential data.

7.18.9 Non-maskable interrupt vector

When a non-maskable interrupt (NMI) occurs, the Apple][saves the status register and program counter onto the stack, reads the vector at $FFFA-FFFB, and then starts executing from the specified address. The ROM BIOS handler immediately transfers control to the code at $3FB-3FD, which is usually a jump instruction to the complete NMI handler. For programs that were very heavily protected, such that inserting breakpoints was difficult because of hooked CSW and KSW vectors, for example, one alternative was to "glitch" the system by using a NMI card to force a NMI to occur. However, that technique required direct access to memory in order to install the jump instruction at $3FB-3FD, since the standard ROM BIOS does not place one there.

On a 64kb Apple][, the ROM BIOS could be copied into banked memory and made writable. The BIOS NMI vector could then be changed directly, potentially bypassing the user-defined NMI vector completely.

7.18.10 Reset vector

On a cold start, and whenever the user presses Ctrl-Reset, the Apple][reads the vector at $FFFC-FFFD, and then starts executing from the specified address. If the Apple][is configured with an Autostart ROM, then the warm-start vector at $3F2-3F3 is used, if the "power-up" byte at $3F4 matched the exclusive-OR of #$A5 with the value at $3F3[57]. The values at $3F2-3F4 are always writable, allowing a program to protect itself against a user pressing Ctrl-Reset in order to gain access to the monitor prompt, and then saving the contents of memory. The typical protected program response to Ctrl-Reset was to erase all of memory and then reboot.

On a 64kb Apple][, the ROM can be copied into banked memory and made writable. When the user presses Ctrl-Reset on an Apple][+, the ROM BIOS is not banked in first, meaning that the cold-start reset vector can be changed directly, and will be used, potentially bypassing the warm-start reset vector completely. On an Apple][e or later, the ROM BIOS is banked in first, meaning that the modified BIOS cold-start reset vector will never be executed, and so the warm-start reset vector cannot be overridden.

7.18.11 Interrupt request vector

Despite not having a source of interrupts in the default configuration, the Apple][did offer support for handling them. When an interrupt request (IRQ) occurs, the Apple][saves the status register and program counter onto the stack, reads the vector at $FFFE-FFFF, and then starts executing from the specified address. However, there is also a special case IRQ, which is triggered by the BRK instruction. This instruction is a single-byte breakpoint instruction, and is intended for debugging purposes. The ROM BIOS handler checks the source of the interrupt, and transfers control to the vector at $3FE-3FF if the source was an external interrupt. On the Autostart ROM, the ROM BIOS handler transfers control to the vector at $3F0-3F1 if the source was a breakpoint. (Pre-Autostart ROMs simply dumped the register values to the screen, and then dropped to the monitor prompt instead.) The values at $3F0-3F1, and $3FE-3FF are always writable, allowing a program to protect itself against a user inserting breakpoints in order to break when execution

reaches the specified address. The typical protected program response to breakpoints was to erase all of memory and then reboot. An alternative protection is to point $3F0-3F1 to another BRK instruction, to produce an infinite loop and hang the machine. Bank Street Writer III uses this technique.

On a 64kb Apple][, the ROM BIOS can be copied into banked memory and made writable. The BIOS IRQ vector can then be changed directly, potentially bypassing the user-defined IRQ vector completely.

7.19 Catalog tricks

7.19.1 Control-"Break"

On a regular DOS disk, there is a sector called the Volume Table Of Contents (VTOC), which describes the starting location (track and sector) of the catalog, among other things. The catalog sectors contain the list on the disk of files which are accessible by DOS. For a file-based program, apart from the DOS and the catalog-related structures, all other content is accessible through the files listed in the catalog. DOS "knows" the track which holds the VTOC, since the track number (usually #$11) is hard-coded in DOS itself, and sector zero is assumed to be the one that holds the VTOC.

Since the files are listable, they can also be loaded from the original disk, and then saved to a copy of the disk. One way to prevent that is to insert control-characters in the filenames. Since control-characters are not visible from the DOS prompt, any attempt to load a file, using the name exactly as it appears, will fail.

Classmate uses this technique. It is also possible to embed backspace characters into the filename. Filenames with backspace characters in them cannot be loaded from the prompt. Instead, a Basic program must be written with printable characters as placeholders, and then the memory image must be altered to replace them with backspace characters.

7.19.2 Now you see it

Since the VTOC also carries the sector of the catalog, it can be altered to point to another location within the track that holds the VTOC. That causes

---

[57]This is true only when the full warm-start vector is not #$00 #$E0 #$45 ($E000 and #$45). If the vector is $E000 and #$45, then the cold-start handler will change it to $E003, and resume execution from $E000. This behavior could have been used as an indirect transfer of control on the Apple][+, by jumping back to the cold-start handler, which would look like an infinite loop, but it would actually resume execution from $E003.

the disk to display a "fake" catalog, while allowing a program to access the real catalog sectors directly.

The Toy Shop uses this technique to show the program title, copyright, and author credits.

7.19.3 Now you don't

Since DOS carries a hard-coded track number for the VTOC, it is easy to patch DOS to look at a different track entirely. The original default track can then be used for data. Any attempt to show the catalog from a regular DOS disk will display garbage.

Ali Baba uses this technique, by moving the entire catalog track to track five.

7.20 Basic tricks

7.20.1 Line linking

Circularly

In Basic on the Apple][, each line contains a reference to the next line to list. As such, several interesting effects are possible. For example, the listing can be made circular, by pointing to a previous line, causing an infinite loop of listing. The simplest example of that looks like this:

```
801:01 08 00 00 3A 00 00 00
```

This program contains one line whose line number is zero, and whose content is a single ":". An attempt to list this program will show an infinite number of "0 :" lines. However it can be executed without issue.

Missing

The listing can be forced to skip lines, by pointing to a line that appears after the next line, like this:

```
801:10 08 00 00 3A 00 10 08 01 00 BA 22
80D:31 22 00 16 08 02 00 3A 00 00 00
```

Listing the program will show two lines:

```
1  0  :
   2  :
```

However, there is a second line (numbered "one") which contains a PRINT statement. Running the program will display the text in line one.

Out-of-order

The listing can list lines in an order that does not match the execution, for example, backwards:

```
801:13 08 03 00 BA 22 30 22 00 1C 08 01 00 BA
22
810:31 22 00 0A 08 03 00 BA 22 32 22 00 00 00
```

This program contains three lines, numbered from zero to two. The list will show the second and third lines in reverse order. The illusion is completed by altering the line number of the first line to a value larger than the other lines. However, the execution of the first line first cannot be altered in this way.

Out-of-bounds

The listing can even be forced to fetch from arbitrary memory, such as the graphics screen or the memory-mapped I/O space:

```
801:55 C0 00 00 3A 00 00 00
```

This program contains a single line whose line number is zero, and whose content is a single ":". An attempt to list this program will cause the second text screen to be displayed instead, and the machine will appear to crash. Further misdirection is possible by placing an entirely different program at an alternative location, which will be listed instead⊚

Imagine the feeling when the drive light turns itself on while the program is being listed!

It might even be possible to create a program with lines that touch the memory-mapped I/O space, and activate or deactivate a stepper-motor phase. If those lines were listed in a specific order, then the drive could be enticed to move to a different track. That track could lie about its position on the disk, but carry alternative content to the proper track, resulting in perhaps subtly different behavior. Are we having fun yet?

7.20.2 Start address

The first line of code to execute can be altered dynamically at runtime, by a "POKE 103, <low addr>" and/or "POKE 104, <high addr>", followed by a "RUN" command. Math Blaster uses this technique.

7.20.3 Line address

Normally, the execution will generally proceed linearly through the program (excluding instructions that legally transfer control, such as subroutine calls and loops), regardless of the references to individual lines. However, the next line (technically, the next

token) to execute can be altered dynamically at run-time, by a "POKE 184, <low addr>". The first value at the new location must be a ':' character. For example, this program:

```
0 POKE 184,14 : END : PRINT "!"
```

will skip the "END" token and print the '!' instead. It is also possible to alter the high address by a "POKE 185, <high address>" as well, but it requires that the second POKE is placed at the new location, which is determined by the new value of the high address and the old value of the low address. It cannot be placed immediately after the address of the first POKE, because that location will not be accessed anymore.

7.20.4 "REM crash"

```
801:0E 08 00 00 B2 0D 04 50 52 23 36 0D 00 00
00
```

This program contains one line, which looks like the following, where the "^" character stands for the Control key.

```
0 REM^M^DPR#6^M
```

When listed with DOS active, it will trigger a reboot. It works because the same I/O routine is used for displaying the text as for typing commands from the keyboard. Zardax uses this technique.

7.20.5 Self-modification

A program can even modify itself dynamically at runtime. For example, this program will display "2" instead of "1". The address of the POKE corresponds to the location of the text in memory.

```
0 POKE 2064,50 : PRINT "1"
```

A program can also extend its code dynamically at runtime:

```
0 DATA 130,58
1 FOR I=0 TO 1 : READ X : POKE 2086+I,X :
```

A "FOR" loop must be terminated by a "NEXT" token, in order to be legal code. Notice that the program does not contain a "NEXT" token, as expected. Instead, the values in the DATA line supply the "NEXT" token and a subsequent ":". The inclusion of a ":" allows extending the line further, simply by adding more values to the "DATA" line and altering the corresponding address of the "POKE".

By using this technique, even entirely new lines can be created.

7.21 Rastan

Rastan is mentioned here only because it is a title for an Apple][system (okay, the IIGS) that carried the means to bypass its own copy-protection! The program contained two copy-protection techniques. One was a disk verification check, which executed shortly after inserting the second disk. The other was a checksum routine which performed part of the calculation between each graphics frame, until it formed the complete value. If the match failed, only then would it display a message. It means that the game would run for a little while before failing, making it extremely difficult to determine where the check was performed.

7.21.1 The Rastan backdoor

In order to avoid waiting for the protection check every time a new version of the code was built, the author[58] inserted a "backdoor" routine which executed before the first protection check could run. The backdoor routine had the ability to disable both protection checks in memory, as well as to add new functionality, such as invincibility and level warping. And where was this backdoor routine located? Inside the highscore file!

Yes. The highscore file had a special format, whereby code could be placed beginning at the third byte of the file. As long as the checksum of the file was valid (an exclusive-OR of every byte of the file yielded a zero), the code would be executed.

Here is the dispatcher code in Rastan:

```
.A16
;checksum data
2000D    JSR    $21216
;note this address
20010    JSR    $2D1C2
```

---

[58] https://twitter.com/JBrooksBSI

73

Here is the checksum routine:

```
 1  .A16
    ;source address
 3  21216     TXA
    ;taken if no highscore file
 5  21217     BEQ     $21240
    ;length of data
 7  21219     LDA     $0,X
    2121D     TAY
 9  2121E     SEP     #$20
    .A8
11  21220     PHX
    ;checksum seed
13  21221     LDA     #0
    ;checksum data
15  21223     EOR     $0,X
    21227     INX
17  21228     DEY
    21229     BNE     $21223
19  2122B     PLX
    2122C     REP     #$30
21  .A16
    2122E     AND     #$FF
23  ;taken if bad checksum, no copy
    21231     BNE     $21240
25  ;length of data
    21233     LDA     $0,X
27  21237     DEC
    21238     LDY     #$D1C0
29  ;copy to $2D1C0
    2123B     MVN     #2, #0
31  2123E     PHK
    2123F     PLB
33  21240     RTS
```

We can see that the data are copied to $2D1C0, the first word is the length of the data, and the first byte after the length (so $2D1C2) is executed directly in 16-bit mode. By default, the file carried an immediate return instruction, but it could have been anything, including this:

```
 1  ;always pass protection
    ;(BRA $+$0F)
 3  2D1C2     LDA     #$0D80
    2D1C5     STA     $22004
 5  ;always pass checksum
    ;(BRA $+$19)
 7  2D1C8     LDA     #$1780
    2D1CB     STA     $3CAD0
 9  2D1CE     RTS
```

7.22 Conclusion

There were many tricks used to protect programs on the Apple][, and what is listed here is not even all of them. Copy-protection and cracking were part of a never-ending cycle of invention and advances on both sides. As the protectors came to understand the hardware more and more, they were able to develop techniques like delayed fetch, or consecutive quarter-tracks. The crackers came up with NMI cards, and the mighty E.D.D. In response, the protectors hooked the NMI vector and exploited a vulnerability in E.D.D.'s read routine. (This is my absolute favorite technique.) The crackers just boot-traced the whole thing.

We can only stand and admire the ingenuity and inventiveness of the protectors like Roland Gustafsson or John Brooks. They were helped by the openness of the Apple][platform and especially its disk system. Even today, we see some of the same styles of protections—anti-disassembly, self-modifying code, compression, and, of course, anti-debugging.

The cycle really is never-ending.

7.23 Acknowledgements