5 x86 Alchemy and Smuggling with Metalkit

by Micah Elizabeth Scott

Dear neighbors, today I humbly present a story of x86 alchemy and bit smuggling. It's an MBR you can take with you, the story of a lonely matryoshka egg, and a spark of something weird intentionally escaping from a place where weird machines are by definition broken.

5.1 Pong test

Two or three lifetimes ago, I was an architect for the desktop USB and GPU virtualization subsystems at VMware. Suffice to say, it was a complicated job handled by a small team of talented, dedicated, and fucking crazy engineers. The story begins with our effort to find new engineers to hire that were just the right kind of talented, dedicated, and crazy. We tried the usual tactics like looking for people who like the beers we do or testing candidates on the minutiae of IEEE floating point in specific GPU configurations. When that worked badly, we got creative. One of my coworkers made up an esoteric minimal instruction set and asked candidates to write programs in it. This was fun for the interviewer, at least. I liked to run the programs in my head and debug them as fast as the candidates wrote on the whiteboard.

One of my coworkers had a new plugin architecture for the part of our virtual machine runtime that handles user input and 2D display compositing, and he suggested we use it as an interviewing tool. So we had them play Pong. We developed a two-hour interview test where candidates wrote a plugin to play against a trivial opponent. The virtual machine boots directly into the game in retro black & white. The right paddle tracks the ball slowly. The left paddle is controlled by the mouse or keyboard. In the interview, I would work through this ridiculous Rube Goldberg contraption with the candidate, giving them just barely enough help so they'd succeed with the available time and materials. The process seemed to be quite good at revealing the candidate's approach toward the kind of ridiculous things we had to do on a daily basis.

To keep the difficulty level and time requirements appropriate, we needed the VM to generate very simple and consistent screen updates. Any general purpose OS would have a time-consuming bootup process, and the GPU commands would be littered with sporadic events that complicate the heuristics required to locate the ball and send the right mouse movements to have the paddle follow it.

The required speed and the level of control ruled out any operating system I knew of, so I wrote my tiny game to run on the virtual bare metal, communicating directly with the registers and command FIFO in our virtual GPU to set up a 2D framebuffer and enqueue just the right update rectangles. We also vastly simplified the interview problem by putting the mouse into absolute-coordinate mode using an extension in our virtual hardware. The very first version used some bare metal support libraries that other teams developed for automated testing of the ridiculously complicated virtual CPU, but I soon replaced those with pieces from an open source bootloader and 32 bit x86 bare metal support library of my own.

5.2 Metalkit

This game worked well for our interview process. My library, named Metalkit, satisfied an acute personal itch to write fiddly low-level code. I worked on my own time, hacking together dynamically generated interrupt vector trampolines while my boyfriend hacked at repetitive monsters in World of Warcraft. At VMware, I then forked a version of Metalkit into an open source library which would serve as public documentation for the virtual GPU device and part of an internal unit testing framework for it. I wanted to release this documentation with plenty of sample code. I ended up creating plenty of 3D rendering examples as a byproduct of creating a low-level unit testing framework for our virtual GPU. When I needed an example for the unaccelerated 2D dumb framebuffer mode, I ported my little PongOS to this library. This new version could be open source, and very tiny.

Metalkit is optimized for creating tiny binaries. Partly it was a personal challenge, but a tiny binary is often a teachable binary. Many a reader has had their first spark of curiosity for ELF after the inspiration of an especially minimal or delicately obfuscated binary. It seemed didactically useful to have a tool for

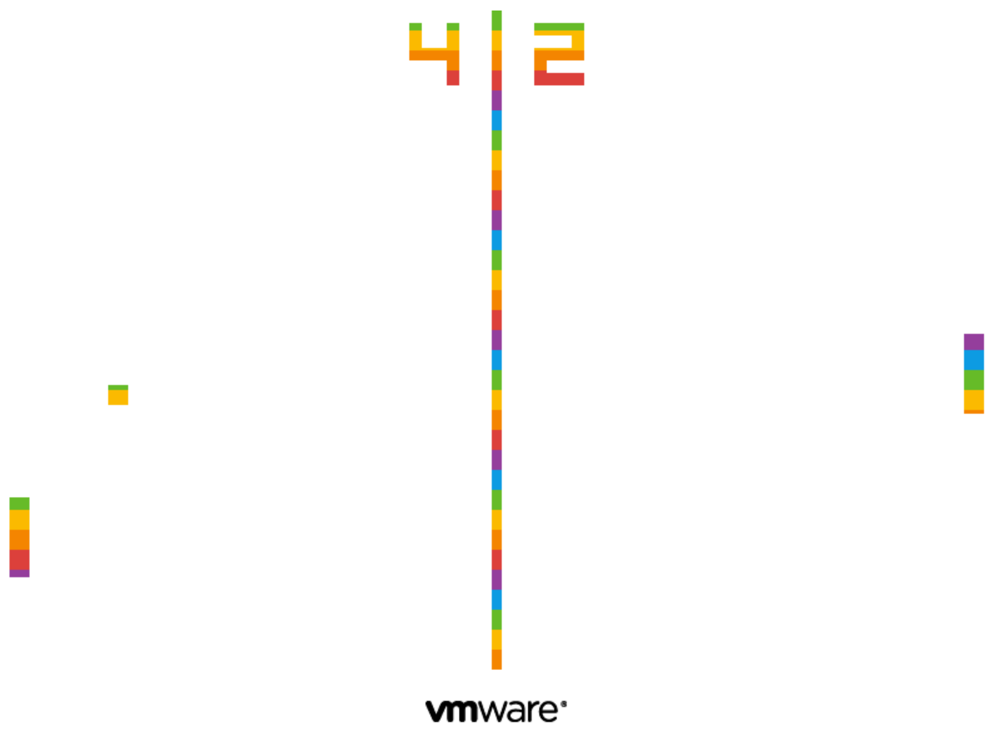


Figure 3: VMWare Pride

creating bare-metal binaries that are fairly easy to compile and also where it can be easy to identify the purpose of every byte in the file. Instead of using a large and complicated standard C library, it includes a very minimal library that's designed for readability, terseness, and a sense that it's possible to understand the whole system.

Readers who choose to study the internals of Metalkit may notice features that go to extremes in order to avoid unnecessary or repetitive code while also allowing complex behaviors. The ISR trampolines, for example, are tiny functions in RAM which wrap the C functions that handle each interrupt vector. These C functions have a simple calling signature that allows a handler to access its vector number and prior execution state as stack parameters. With the help of some macros, handler functions can inspect or write this saved execution state to implement features like task switching. There's a separate trampoline for each interrupt vector, and to save space in the disk image they're constructed in RAM during initialization by following a repeating pattern:

```
   60                pusha                         ; Save general-purpose regs
 2 68 <32 bit arg>   push   <arg>                  ; Call handler(arg)
   b8 <32 bit addr>  mov    <addr>, %eax
 4 ff d0             call   *%eax
   58                pop    %eax                   ; Remove arg from stack
 6 8b 7c 24 0c       mov    12(%esp), %edi         ; Load new stack address
   8d 74 24 28       lea    40(%esp), %esi         ; Addr of eflags on old stack
 8 83 c7 08          add    $8, %edi               ; Addr of eflags on new stack
   fd                std                           ; Copy backwards
10 a5                movsl                         ; Copy eflags
   a5                movsl                         ; Copy cs
12 a5                movsl                         ; Copy eip
   61                popa                          ; Restore general-purpose regs
14 8b 64 24 ec       mov    -20(%esp), %esp        ; Switch stacks
   cf                iret                          ; Restore eip, cs, eflags
```

In the spirit of teaching someone to fish rather than handing them a can, I thought it prudent to set the example of teaching machines to write the repetitive code, and how the runtime initialization might perform this task more efficiently than the compiler could. Readers accustomed to the luxuries and tragedies of ARM or x86-64 may need to adjust their spectacles to adequately behold the 32 bit ISR template above, as excerpted from the comments in Metalkit's `intr.c` module.

The most extreme example of design economy in Metalkit is the MBR. This 512 byte header is generated and placed with the help of a custom linker script. It includes a plausible partition table and a carefully crafted hunk of assembly that the BIOS will splat into low RAM and run for us in 16 bit Real Mode. For convenience and ease of use as a teaching and testing tool, I wanted a minimal and highly convenient bootloader. It should put the CPU into 32 bit mode, load a flat binary image into RAM, set up the execution environment, and call `main()`. I wanted it to be an effortless result of typing `make` in a project, but to also handle loading arbitrarily large images from devices like virtual CD-ROM drives and USB disks. Oh, and we should make it boot from GRUB too.

5.3 Boot from anything in under 512 bytes

People never use the BIOS any more. System geeks spend all this time making sure it works in every case, but nobody really notices. A modern BIOS has a huge library of available functionality. If you've ever programmed in DOS, you've seen BIOS interrupts.[7] They're like system calls, but with fewer rules. Decades and decades of backward compatibility happened, all with layers of emulation so you can happily keep calling interrupt 0x13 for WRITE DISK SECTORS without anyone but weird people like us worrying that the data's going to a solid state disk plugged into a hub on an xHCI USB 3.0 controller over PCIe rather than to a hunk of spinning rust from 1980 on a 4 MHz parallel bus.

[7] `http://www.ctyme.com/intr/cat-003.htm`

There are a bunch of reasons not to use these routines in modern code, chiefly that they need to run in 16 bit Real Mode, which can only address about the first megabyte of RAM. During the transition from DOS to 32 bit operating systems, various strategies emerged for dealing with the fact that the drivers in the PC's BIOS only work in 16 bit mode. Usually the BIOS functionality is reimplemented entirely in the OS for efficiency and maintainability, and this is feasible because the hardware is documented, standardized, or interesting enough to get reverse engineered. There are exceptions for sure, like XFree86 running 16 bit VESA BIOS video drivers in an emulator in order to run the GPU through proprietary mode switch sequences and obtain framebuffer access.

Even a modern bootloader will pass up the chance to use the BIOS as soon as it can load its own driver. GRUB has an MBR riddled with esoteric bug workarounds, its mission only to launch a 32 kiB or less stage2 binary from a prearranged sector on disk. The BIOS gained an unflattering reputation from decades of buggy drivers and a penchant for claiming 640 kiB is enough RAM for anyone.

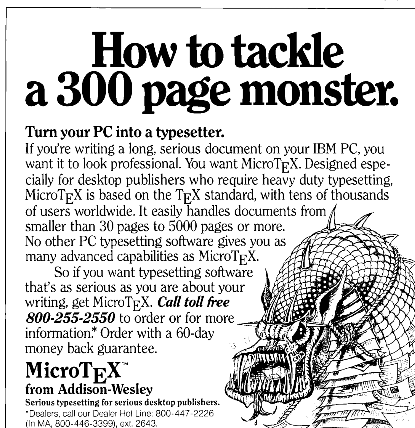
With Metalkit, we can try to move past that and see the BIOS as yet another niche where we can find reusable gadgets. If we can stomach a switch to 16 bit Real Mode and back for each batch of sectors, we can use the BIOS to read from the bootup disk (whatever stack of emulations that may be) into a small scratch buffer below 640 kiB. Then, back in 32 bit Protected Mode, we shuttle that data up above 1 MB. Repeat this enough times and we could load a whole CD-ROM into memory, 9 kiB at a time.

With the popularity these days of usermode programming and 64 bit portability it's easy to forget entirely that the CPU still knows how to execute 16 bit instructions. Of course, for compatibility it always starts in 16 bit mode, but typically a bootloader like GRUB will switch to 32 bit Protected Mode as soon as possible, and nobody looks back. With the advent of UEFI, we even have a 64 bit replacement for BIOS.

You may remember that darling of the late 90s, VM86 mode. I remember such thrills from the `vm86(2)` manpage when I first started monkeying with Linux. A system call to emulate 16 bit mode! In a sandbox! Using a built-in CPU feature! It was part of Wine, part of X. Now it's obsolete again, incompatible with 64 bit operating systems. We don't need anything so glitzy for this job, though. Being a bootloader with free rein of the processor's GDT and segment descriptors, we can toggle off Protected Mode and reload the segment registers to point them back at low memory. It can be tricky to debug code like this, but the low-level debuggers in both VMware and Bochs let you examine the CPU state directly during these critical mode switches.

Even our minimal and modern bootloader can't escape all the woe and pageantry of backward compatibility. The first thing we do is switch on the A20 gate, which if you haven't run across yet I would suggest you save to look up next time you'd like to spend some meditative time crying and/or laughing into Wikipedia.

For each disk read, we prefer to use the more modern Logical Block Address (LBA) addressing mode, where each disk sector has an index starting from zero like any sensible API would use. Of course, before LBA, disks didn't really have the API of a generic storage interface made from uniform and abstracted



PCYACC™
Version 2.0
PROFESSIONAL
LANGUAGE DEVELOPMENT TOOLKIT
Professional Version $395.00—Personal Version $139.00
Includes "Drop In" Language Engines for SQL, dBASE, POSTSCRIPT, HYPERTALK, SMALLTALK-80, C++, C, PASCAL, and PROLOG.
PCYACC Version 2.0 is a complete Language Development Environment that generates ANSI C source code from input Language Description Grammars for building Assemblers, Compilers, Interpreters, Browsers, Page Description Languages, Language Translators, Syntax Directed Editors, and Query languages.
Complete grammars, Lexical Analyzers, and Symbol Table Management for ANSI C, K&R C, ISO Pascal, dBASE III/Plus and IV, SQL, C++, Smalltalk-80, APPLE HyperTalk, C&M Prolog, YACC, LEX, and POSTSCRIPT are included. OS/2 and MAC versions are available.
Example application sources are provided to be used as skeletons for new programs. Examples include a desktop calculator, an Infix to Postfix Translator, a DBASE and SQL Syntax analyzer, an implementation of the PIC[ture] language, and a C++ to C translator.
- Lexical Analyzer Generator ABRAXAS PCLEX is included
- Quick Syntax analysis option
- Optional Abstract Syntax Tree
- Advanced Error recovery Support Provided
- Manual "Compiler Construction with PC'S" included
- All examples include FULL SOURCE
- 30 day money back guarantee
- No charge for shipping anywhere in the world
To order, please call
1-800-347-5214
ABRAXAS™
SOFTWARE, INC.
7033 SW Macadam Ave. Portland, OR 97219 USA
TEL (503) 244-5253 · FAX (503) 244-8375
AppleLink D2205 · MCI ABRAXAS

512-byte sectors; they had the API of a spinning magnetic stack and wubbling electronic wand, each with a particular shape and speed. This older form of addressing was known as Cylinder Head Sector (CHS). Metalkit will try LBA first, since it's necessary for newer devices like USB sticks and CD-ROMs, with CHS as a backup so that plain floppy disks work on any BIOS.

We read 18 sectors at a time, or 9 kiB. It's the same as one old-style magnetic track on a 1.44 MiB disk, to minimize the impact of CHS addressing on the size of the bootloader. After the BIOS returns, we have to do our first jump to 32 bit Protected Mode to copy that block into place:

```
1           ; Enter Protected Mode, so we can copy this sector to
            ; memory above the 1MB boundary.
3           ;
            ; Note that we reset CS, DS, and ES,
5           ; but we don't modify the stack at all.

7           cli
            lgdt    BIOS_PTR(bios_gdt_desc)
9           movl    %cr0, %eax
            orl     $1, %eax
11          movl    %eax, %cr0
            ljmp    $BOOT_CODE_SEG, $BIOS_PTR(copy_enter32)
13          .code32
copy_enter32:
15          movw    $BOOT_DATA_SEG, %ax
            movw    %ax, %ds
17          movw    %ax, %es

19          ;
            ; Copy the buffer to high memory.
21          ;

23          mov     $DISK_BUFFER, %esi
            mov     BIOS_PTR(dest_address), %edi
25          mov     $(DISK_BUFFER_SIZE / 4), %ecx
            rep movsl
```

The x86 architecture is full of features modern programmers prefer to sweep under the rug. The x86 segment registers are usually like this, vital in every DOS program but today unused aside from the inner workings of thread-local storage, language runtimes, exception handlers, OpenGL APIs, and the like. We may forget that these registers on x86 are actually a somewhat miraculous feat of backward-compatilogical engineering starting with the 80286 design.

The original 8086 architecture included four 16 bit segment registers. Each one was padded out to 20 bits, functioning as a selectable base for code and data addressing calculations on a 16 bit machine that could address a whole megabyte of RAM. In the 80286, the new Protected Mode was introduced. Instead of simple arithmetic, the segment registers were now processed via a lookup table, the Local Descriptor Table (LDT). This ancient hack introduced a magical quality to each segment register, remaining there inside every x86 to this day.

In this code segment, BOOT_DATA_SEG and BOOT_CODE_SEG are preprocessor macros that refer to particular entries in descriptor tables we set up earlier in boot. In Protected Mode, these next instructions contain some magic:

```
            movw    %ax, %ds
2           movw    %ax, %es
```

Friends, what looks like a straightforward register-to-register mov is anything but. The guiding tenet of Protected Mode is the fundamental right of abstraction for all segment registers. On an 8086, these instructions would save a 16 bit value from %ax in the 16 bit registers %ds and %es. Later, during address

calculations, the 16 bit value in the applicable segment register would be padded with zeroes on the right and added to the relevant offset to form a 20 bit address that could reach an entire Megabyte of physical memory. Protected Mode was a sort of Pandora's box. With the box open, a segment register is now just an idea, hopelessly modern and abstract, like the exact position of an electron. Writing an index to this register is taken as an instruction to fetch a descriptor from the named table entry, populating some internal and almost-invisible state variables within the processor.

After the copy, we reverse this machinery to descend back down to Real Mode and grab another 18 sectors. With Protected Mode disabled, writing 0 to %ds and %es actually just sets the offset to a 16 bit value of zero instead of loading from the descriptor table. There is a spooky in-between state nicknamed Unreal Mode where it's possible to be in real-mode with values lingering in the processor's segment descriptors that could only have been set by Protected Mode. I had some trouble with the BIOSes I tested, but all reliably operate their disk and USB drivers in this state.

```
        ; 2. Disable Protected Mode

        movl    %cr0, %eax
        andl    $(~1), %eax
        movl    %eax, %cr0

        ; 3. Load real-mode segment registers. (CS, DS, ES)

        xorw    %ax, %ax
        movw    %ax, %ds
        movw    %ax, %es
        ljmp    $0, $BIOS_PTR(disk_copy_loop)
```

Memory addressing may prove to be particularly mindboggling in an environment such as this. I wrote the bootloader to use GNU's assembler, which knows how to switch at any point between 16 bit and 32 bit code. But, of course, I also need to use different addressing schemes for both of these modes, and there's no help from the compiler on this job. I use a collection of linker script calculations and preprocessor macros to calculate 16 bit addresses, and I let the assembler assume 32 bit memory addresses everywhere. This works out better anyway, since GNU binutils doesn't help much when it comes to 16 bit anything.

The actual switch between 16 bit and 32 bit code is distinct from the switch to and from Protected Mode. In fact, the CR0 bit that enables Protected Mode really just changes this segment loading behavior. The other features we get, like segment limits, paging, and 32 bit code, are enabled with settings in the descriptors we load via this new flavor of segment register we get in Protected Mode. The bitness actually changes when we perform a long jump across segments after changing the segment descriptor for %cs and friends. To orchestrate the change, we need the processor bitness, assembler bitness, and calculated addressing to all line up just right:

```
        ljmp    $BOOT_CODE_SEG, $BIOS_PTR(copy_enter32)
        .code32
copy_enter32:
```

With these tricks, it's possible to load an arbitrarily large next stage into RAM and execute it. This could be a 6 kB Pong game, a 10 MB GPU unit test, Hello World, another bootloader stage, or maybe even an operating system kernel.

Using the BIOS for disk input and a tiny bit of display output, and including the bare minimum amount of backward-compatibility code, this functionality just barely fits into the 512 byte MBR. We even have room for a real partition table. In the celebration and recognition of polyglots everywhere, a GNU Multiboot header can sneak into any free 32 bytes within the first 8 kB and conveniently allow us to boot the image directly from GRUB as well.

Friends, think of Metalkit as My First 32 bit x86 Playset for Kids and Adults. I urge you, get the code and write a round-robin thread scheduler with your teenager tonight.[8]

5.4 Bug hunter

In the lopsided and sometimes oppressive culture of a rising Silicon Valley juggernaut, there were some small subversions I took pride in. I was so productive and worked so much that I often chose my own side-projects to mix things up a little. I'd fix little personal nitpicks. I'd look for security vulnerabilities. In my last year there, I wrote a Bluetooth stack mostly to avoid boredom.

I once spent some time to implement oldschool CGA graphics mode emulation to fix a robot game I like. It turns out that our BIOS had already inherited code to emulate these modes on top of VGA hardware. So the BIOS was trying to get there by telling our virtual GPU to be a VGA device in a mode that's almost correct. Then the BIOS flips a bit in the VGA device telling it to interpret the framebuffer in CGA's particular planar style. This was the missing piece. I implemented a new blitter in the emulation that handled this case, tested Robot Odyssey and Arcade Volleyball, and proudly resolved bug #3 in our tracker: "CGA mode does not work."

Along the way another bug caught my eye. #62382, "We don't have any easter eggs in our products." It was filed back in 2005 by a platform engineer with a healthy sense of humor. The bug gained comments from a range of people, from a curt "whatever" and temporary erasure to eventual revival and enthusiastic support. To me, easter eggs were more than just a cute toy. They were a way of leaving a distinctly personal artistic signature inside something that was intended to be a faceless commodity product. It was a subversion I was happy to play a role in, and I figured PongOS was the perfect solution this time: small enough nobody could complain about its size if anyone noticed it at all, isolated by the same sandbox we trust other VMs inside, and I had a very subtle strategy for storing and triggering the disk image payload.

In the pressure to satisfy increasingly convoluted backward compatibility requirements, platform engineers thrive by strategizing around and curating maps of undefined states. We specifically leave places where behavior is not specified by the design, leaving subtle traps to discourage developers from fouling the pristinely undefined by becoming reliant on our current unplanned placeholder behaviors.

I looked for a way to introduce an easter egg that could be triggered intentionally but which would stay out of the way by only appearing in a state that I decided was safely in one of these formerly unfriendly regions. The trigger I chose was a zero-byte floppy disk image attached to a desktop VM. This normally wouldn't do anything useful; there is no reason to have a zero-byte image attached instead of no image at all, and booting in this state would lead to an error message from the BIOS.

The inner workings of this egg could be obscure as well. The floppy disk emulation was a crusty piece of code few people would touch, and most of those who cared about and understood it had a lively sense of humor and individuality. We routinely had to monkey-patch our zoo of devices around some obscure operating system incompatibility. I wrote a patch that, as innocently as possible, included a header file with 6 kilobytes of hexadecimal data labeled as a "default parameter buffer," the implication being that it helped us in emulating some obscure floppy driver compatibility mode. When reading past the end of a floppy disk image (very different from no image at all), we would read from this default buffer. With a zero-byte disk image, we're reading entirely from this buffer and booting into PongOS.

Friends who worked a little farther from the metal added to each of the platform-specific user interfaces an obscure keyboard macro that would deploy a Paschal Ovum virtual machine with a zero-byte floppy image.

5.5 Revision

The egg would always be controversial among the small but influential group inside the company who knew about it. Many people could have prevented it from ever shipping, and indeed to some outsiders unfamiliar

[8] `git clone https://github.com/scanlime/metalkit`
VMWare fork at `http://vmware-svga.sourceforge.net/`

with the sausage-making process inherent in software development, it could seem strange that such whimsical code would ever make it past the strict QA processes.

But it should be apparent to any developer and obvious to any security researcher that it's impossible to test for the absence of a feature like this, and in reality the complex systems software we all rely on is so fiendishly complex that it's possible nobody completely understands even a single OS kernel. Those who come the closest to a complete understanding tend, in my experience, to have a jaded and pessimistic view of kernels, device drivers, and communications stacks everywhere. The most jaded and curmudgeonly would never want us to support graphics virtualization at all, and from a purely security position they would probably be right.

In an unfortunate but probably inevitable string of events, someone inadvertently triggered the easter egg on a VM that normally wouldn't have booted, then they misunderstood the outcome and posted to the forums about a "virus." This eventually almost got the egg pulled, but we reached a compromise: I could keep it if I added a VMware logo to the screen.

Now I had a challenge for myself. For starters, I'd create a new binary image that's no larger than before, with a nice looking logo. I wanted to go further, hiding an additional easter egg in the program. By carefully pruning down and further optimizing the code in Metalkit, I saved entire kilobytes. I used a tiny 4 bit RLE format for storing an anti-aliased logo image, and trimmed down all the math, graphics, and PCI code as small as possible. The details are too numerous to list, but the intrepid reader will find the bytes in the attached disk image number few enough to comfortably reverse engineer without too much despair.

For the nested easter egg, I added an obscure state machine to the keyboard ISR, toggling a drawing mode when it detects the sequence of scancodes that make up `{'p','r','i','d','e'}`. With the special drawing mode, a new color lookup table is activated and cycled when filling each scanline. I wanted this layer of the egg to be a representation of the hidden struggles we go through and often keep to ourselves in our work. And perhaps it was also a subtle nod to the specific rainbow in the Apple II logo, and the love that myself and many of my coworkers recently put into creating our first virtualization product for the Mac.

5.6 Call to remix

Within this PDF, readers will find PongOS attached in the form of an Ableton sampler preset for those who wish to, at various octaves, test their own perception for sonic-executable synesthesia in densely packed uncompressed x86 code.

For other uses, rest assured a few lines of your favorite snake-based language are sufficient to make the image suitable for boot or disassembly again.

```
1 >>> import struct
  >>> aiff = open("egg.aiff", "rb").read()
3 >>> floats = struct.unpack(">6710f", aiff)
  >>> bytes = [chr(int((i + 1) * 128)) for i in floats[36:-18]]
5 >>> open("egg.img", "wb").write("".join(bytes))

7 -rw-r--r--  1 micah  staff  6656 Sep 20 00:07 egg.img
  0a710d1776f0687170b7d547c1d70354d6bba548  egg.img
```

With or without the enclosed, I encourage you all to express yourself in ways nobody thinks possible. Remember the old proverb: a wise explorer learns more about television with a magnet than a couch.