3 Coastermelt

by Micah Elizabeth Scott

3.1 Getting Inside Your Optical Drive's Head

This is the first of perhaps several articles on the adventures of `coastermelt`, an art-hacking project with the goal of creating cheap laser graffiti using discs burned by Blu-Ray drives with hacked firmware.

3.1.1 Art Hacking Manifesto

If an engineer is a problem solver, hackers and artists are more like problem tinkerers. Some of the most interesting problems are so far beyond the scope of any direct solution that it seems futile to even approach them head-on. It is the artist's purview to creatively approach these problems, sideways or upside down if necessary.

When an engineer is paid to make a tool, is it not the money itself that ultimately decides the tool's function? I believe that to be a hacker is to see tools as things not only to make but to re-make and subvert. By this creative reapplication of technology, research and problem-solving need not be restricted to those who own the means of production.

So says the Maker's manifesto: if you can't open it, you don't own it. I'd like to build on this: if we work together to open it, we all own it. And maybe we can all learn something along the way.

3.1.2 I heard there were laser robots?

Why yes, laser robots! Optical discs may be all but dead as a data storage medium, but the latest BD-RW drives contain feats of electromechanical engineering that leave any commercial 2D or 3D printer in the dust. Using a 405 nm laser, they can create marks only 150 nm long, with accuracy better than 70 nm. Tiny lenses mounted on a fast electromagnetic suspension can keep perfect focus on grooves only 320 nm apart as the disc spins at over 7 m/s.

A specialized system-on-chip generates motor and laser control signals, amplifies and demodulates the light signals captured by a photodiode array, and it does all of this in the service of fairly pedestrian tasks like playing motion pictures and making backups of cat photos.

My theory is that, with quite a lot of effort, it would be possible to create new firmware for a common Blu-Ray burner such that we could burn discs with arbitrary patterns. Instead of the modulated binary data that stays nicely separated into the tracks of a spiral groove, I think we can treat the whole disc surface as a canvas to draw on with sub-100 nm precision.

If this works, it should be possible to create patterns fine enough that they diffract interestingly under red laser illumination. By bouncing a powerful laser pointer off of a specially burned BD-R disc and targeting a flat surface, perhaps we can control the shape of the eventual illumination well enough to project words or symbols.

This is admittedly a very long shot. Perhaps the patterns have nowhere near enough resolution. Perhaps the laser pointer would need to be much too powerful. If this works out, I dream of creating a mobile printing press for light graffiti. If not, I suspect the project may still lead somewhere interesting.

3.1.3 Device Under Test

For `coastermelt` I chose the Samsung SE-506CB optical drive, a portable USB 2.0 burner that's currently quite popular. It retails for about $80. Inside, I found an MT1939 SoC, an undocumented and highly application-specific chip from MediaTek. It was easy to find some firmware updates which became a starting point for understanding this complicated black box.

My current understanding is that the MT1939 contains a pokey ARM7 processor core along with a lot of strange application-specific peripherals and about 4 MB of RAM. There's also an 8-bit 8051 processor core in there, which shares access to the USB controller. The USB software stack seems to be confusingly split between the ARM firmware and the tiny 8051 firmware, for still-unknown reasons.

There are two customized and undocumented motor control chips from TI, which drive a stepper motor, brushless motor, and the voice coils that quickly position and focus the lenses. As far as I can tell, these chips just act as high-power load drivers. All of the logic and timing seems to be within that MT1939 chip.

3.1.4 How did we get here anyway?

This has been a complex journey full of individual hacks that could each make an interesting story. In my experience, reverse engineering is much like playing a point-and-click or text adventure game. There's a huge world to explore, and so much of your time can be spent on probing the boundaries of that world, understanding who the characters are and what their motivations are, and suffering through plenty of enlightening but frustrating dead-ends.

I wanted to share this process as best I could, in a way that could be documentation for the project, an educational peek into the world of reverse engineering, and an invitation to collaborate. I created a video series[3] with two episodes so far. I won't repeat those stories here; let's go somewhere new.

3.1.5 Down the Rabbit Hole

If you take the blue pill, the story ends, and you wake up believing your optical drives only accept standard SCSI commands that read and write data according to the established MMC specifications.

Of course, that is a convenient fairy tale. Firmware updates exist, and so we know the protocol must be Turing-complete already. In this tiny world, our red pill is a patched firmware image that adds a backdoor[4] with enough functionality to implement a simple debugger. After installing the patch,[5] we can go in:

```
backdoor micah$ ./cmshell.py


                        __                             __ __
.----.-----.---.-.-----|  |_.-----.----.--------.-----|  |  |_
|  __|  _  |  _  |__ --|   _|  -__|   _|        |  -__|  |   _|
|____|_____|___._|_____|____|_____|__| |__|__|__|_____|__|____|
--IPython Shell for Interactive Exploration--------------------

Read, write, or fill ARM memory. Numbers are hex. Trailing _ is
short for 0000, leading _ adds 'pad' scratchpad RAM offset.
Internal _ are ignored so you can use them as separators.

    rd 1ff_ 100
    wr _ 1febb
    ALSO: rdw, wrb, fill, watch, find
          bitset, bitfuzz, peek, poke, read_block

Disassemble, assemble, and invoke ARM assembly:

    dis 3100
    asm _4 mov r3, #0x14
    dis _4 10
    ea mrs r0, cpsr; ldr r1, =0xaa000000; orr r0, r1
    ALSO: tea, blx, assemble, disassemble, evalasm
```

[3]https://vimeo.com/channels/coastermelt
[4]https://github.com/scanlime/coastermelt
[5]There's a Getting Started section in the README that should help.

```
Or compile and invoke C++ code with console output:

    ec 0x42
    ec ((uint16_t*)pad)[40]++
    ecc println("Hello World!")
    ALSO: console, compile, evalc

Live code patching and tracing:

    hook -Rrcm "Eject button" 18eb4
    ALSO: ovl, wrf, asmf, ivt

You can use integer globals in C++ and ASM snippets,
or define/replace a named C++ function:

    fc uint32_t* words = (uint32_t*) buffer
    buffer = pad + 0x100
    ec words[0] += 0x50
    asm _ ldr r0, =buffer; bx lr

You can script the device's SCSI interface too:

    sc c ac              # Backdoor signature
    sc 8 ff 00 ff        # Undocumented firmware version
    ALSO: reset, eject, sc_sense, sc_read, scsi_in, scsi_out

With a hardware serial port, you can backdoor the 8051:

    bitbang -8 /dev/tty.usb<tab>
    wx8 4b50 a5
    rx8 4d00

Happy hacking!    -- Type 'thing?' for help on 'thing' or
~MeS`14              '?' for IPython, '%h' for this again.

In [1]:
```

Such a strange debugger! At a basic level everything works by *peek* and *poke* in memory with the occasional *call*. The shell is based on the delightful IPython, with commands for easy inline C++ and assembly code. Integer variables and register values are bridged across languages when possible.

3.1.6 GO NORTH; LOOK

You have entered a console full of strange commands. The CPU seems to be an ARM. You don't know what it's doing now, but it runs your commands when asked. Before you appears a vast 32-bit address space, mostly empty.

You happen to see a note on the ground, a splotchy Hilbert curve napkin sketch followed by a handwritten table of hexadecimal numbers with uncertain names scrawled nearby.

Flash, 2 MB	00000000 - 001fffff
...write-protected bootloader, 64 kB	00000000 - 0000ffff
...loadable, 1863 kB	00010000 - 001e1fff
...storage, 120 kB	001e2000 - 001fffff
DRAM, 4 MB	01c08000 - 02007fff
MMIO	04000000 - 043fffff

You can peek around at memory, and things seem to be as they appear for the most part. The flash memory can be read and disassembled, interrupt vectors pointing to code that can unfurl into many hours of disassembly and head-scratching. DRAM at this point is like a ghost town, plenty of space to build scaffolding or conduct science.

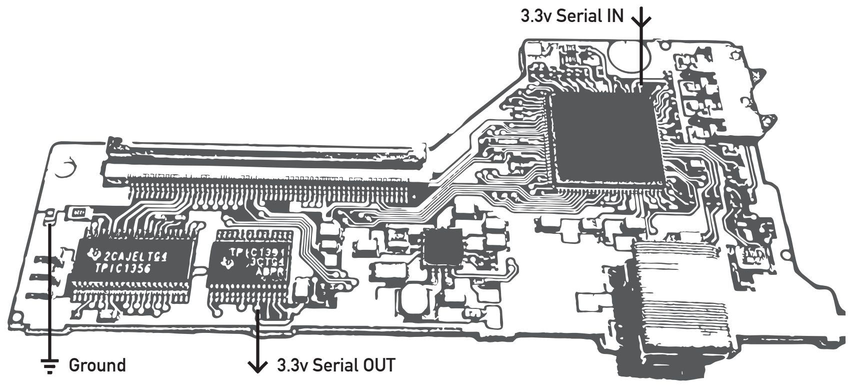
```
In [1]: ea mov r0, pc; mov r1, sp
r0 = 0x01e4000c, r1 = 0x0200067c

In [2]: rdw 200067c 30
0200067c  01000000 01e40000 01ffc290 00000007 0000000d 01ffc2a8 0004bad7 00000000
0200069c  01ffc290 02000cf8 01ffc290 02000cf8 0001efa9 00000000 00000000 02000cdc
020006bc  01ffb76c 02000c0e 0001ec2f 00000000 02000cdc 01ffb76c 00018c07 00000000
020006dc  00018e31 00000032 02000cdc 00167558 00000000 00000000 00000000 00000000
020006fc  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0200071c  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Using some inline assembly, we find the program counter and stack pointer, and separately we dump the memory where the top of the stack was. These can't tell us what the firmware would have been doing had we not rudely interrupted with our backdoor, but these are breadcrumbs showing us some of the steps the firmware took just before we intervened.

3.1.7 30 Gauge Enamel–Coated Freedom

Direct physical access is of course the ultimate hacking tool. With the USB backdoor we can send the ARM processor cutesy little notes asking it or even daring it to run instructions for us, but this will end in heartbreak if we expect to hold the CPU's attention for longer than one fleeting SCSI command.

Heartbreak is a complicated thing though, sometimes it can act like a forest fire leaving the ground fertile for fresh inspiration. If the ARM and the SCSI driver were to never speak again, how could we still contact the ARM? This is where we need to warm our soldering irons. If there's blue wire there's a way. Let's add a serial port for the next step.



3.1.8 GET WALKTHROUGH

In the first `coastermelt` video, I got as far as using this serial port to build an alternate debug backdoor that can break free from the control flow in the original firmware.

```
In [1]: bitbang -8 /dev/tty.usbserial-A400378p
* Handler compiled to 0x2e8 bytes, loaded at 0x1e48000
* ISR assembled to 0xdc bytes, loaded at 0x1e48300
* Hook at 0x18ccc, returning to 0x18cce
* RAM overlay, 0x8 bytes, loaded at 0x18ccc
* Connecting to bitbang backdoor via /dev/tty.usbserial-A400378p
* Debug interface switched to <bitbang.BitbangDevice instance at 0x102979998>
 305 / 305 words sent
* 8051 backdoor is 0xef bytes, loaded at 0x1e49000
* ARM library is 0x3d4 bytes, loaded at 0x1e490f0
* 8051 backdoor running
```

In the second video, I introduced a CPU emulator that can run the ARM firmware on your host computer, proxying all I/O operations back to the debug backdoor while of course logging them.

```
In [2]: sim
 235 / 235 words sent
* Installed High Level Emulation handlers at 01e00000
- initialized simulation state
[INIT] ...........0 ----- >00000000       ldr      pc, [pc, #24]
  r0=00000000  r4=00000000  r8=00000000 r12=00000000
  r1=00000000  r5=00000000  r9=00000000  sp=00000000
  r2=00000000  r6=00000000 r10=00000000  lr=ffffffff
  r3=00000000  r7=00000000 r11=00000000  pc=00000000
```

Now we can follow in the normal firmware's footsteps, mapping out the tiny islands of I/O scattered through this sea of memory addresses. As the `%sim` command churns away, every instruction and memory access shows up in `trace.log`. In the video you can see a demo where a properly arranged replay of these register writes can trigger motor movement.

This trace log is like a walkthrough, showing us exactly how the normal firmware would use the hardware. It's helpful, but certainly not without its limitations. There's so much data that it takes some clever filtering to get much out of it, and it's quite slow to run the simulation. It's a starting point, though, and it can offer clues and memory addresses to use in other experiments with other tools.

At this point in the project, we have some basic implements of cartography, but there isn't much of a map yet. Do you like exploring? I have the feeling there's some really neat stuff in here. With so much interesting hardware to map out, there's enough adventure to share. Take an interesting journey, and be sure to tell us what you find.