4 The FaceWhisperer for USB Glitching; or, Reading RFID with ROP and a Wacom Tablet

by Micah Elizabeth Scott

Greetings, neighbors!

Today, like most days, I would like to celebrate the diversity of tiny machines around us. This time I've prepared a USB magic trick of sorts, incorporating techniques from the analog and the digital domains.

Regular readers will be well aware that computer peripherals are typically general-purpose computers themselves, and the operating system often trusts them a little too much. Devices attached to Thunderbolt (PCI Express) are trusted as much as the CPU. Devices attached to USB, at best, are as privileged as the user, who can typically do anything they want albeit slowly and using interfaces designed for meat.[11] If that USB device can exploit a bug in literally any available driver, the device could achieve even more direct levels of control.

[11] `unzip pocorgtfo13.pdf meat.txt`

Not only are these peripherals small computers with storage and vulnerabilities and secrets, they typically have very direct access to their own hardware. It's often firmware's responsibility to set up clocks, program power converters, and process analog signals. Projects like BadUSB have focused on reprogramming a USB device to attack the computer they're attached to. What about using the available low-level peripherals in ways they weren't intended?

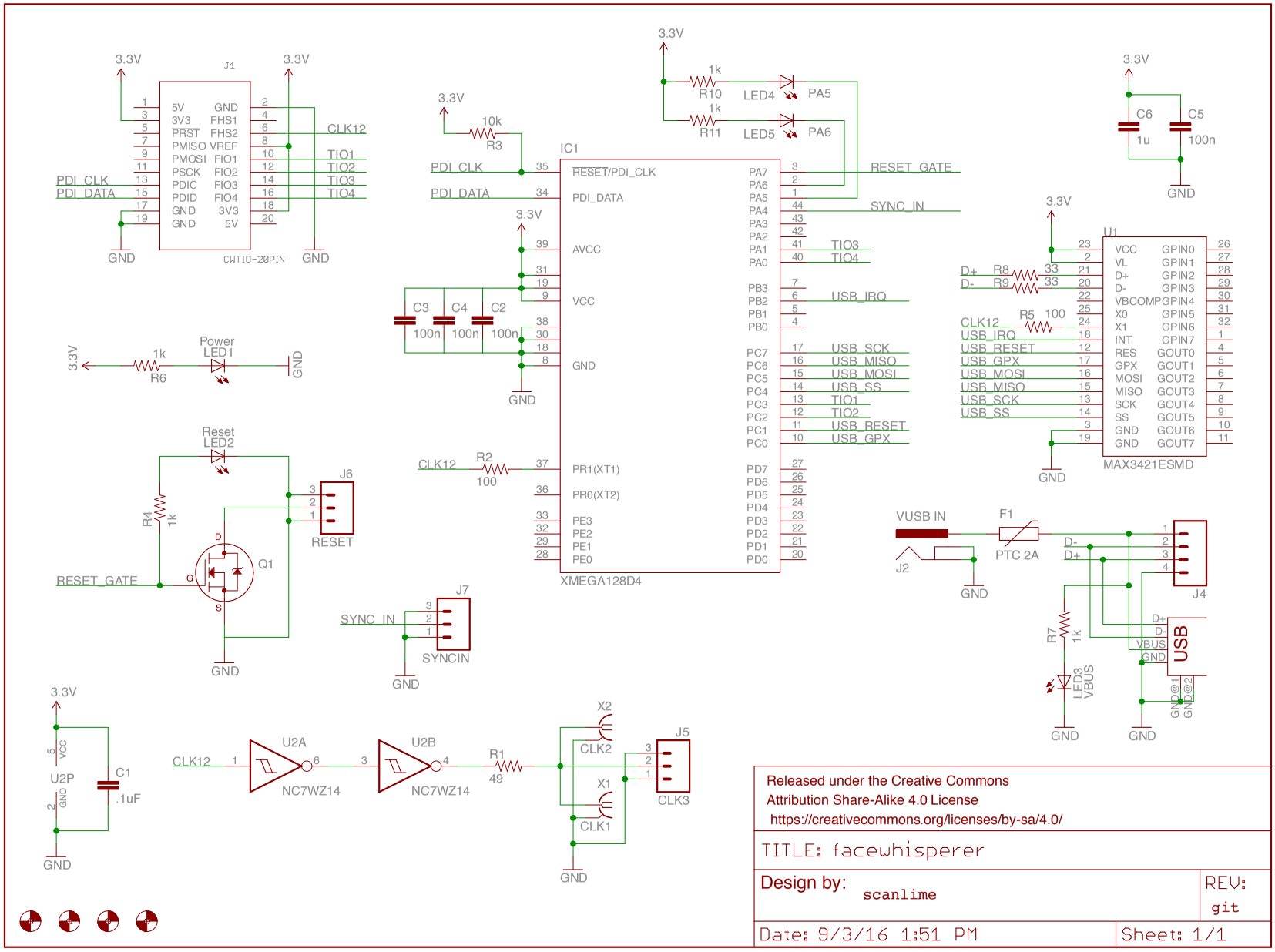
I recently made a video, a "Graphics Tablet Primer for Hackers," going into some detail on how a pen tablet input device actually works. I compared the electromagnetic power and data transfer to the low-frequency RFID cards still used by many door access control systems. At the time this was just a convenient didactic tool, but it did start me wondering just how hard it would be to use a graphics tablet to read 125 kHz RFID cards.

I had somewhat arbitrarily chosen a Wacom CTE-450 (Bamboo Fun) tablet for this experiment. I had one handy, and I'd already done a little preliminary reversing on its protocol and circuit design. It's old enough that it didn't seem to use any custom Wacom silicon, recent enough to be both cheap and plentiful on the second-hand market.

4.1 A Very Descriptive Descriptor

Typically you need firmware to analyze a device. Documented interfaces are the tip of the iceberg. To really see what a device is capable of, you need to see everything the firmware knows how to do. Sometimes this is easy to get. Back in PoC||GTFO 7:3 when I was reversing an optical drive, the firmware was plainly available from the manufacturer's web site. Usually you won't be so lucky. Manufacturers often encrypt firmware to hide their crimes or slow down clones, and some devices don't appear to support firmware updates at all.

This device seemed to be the latter kind. No firmware updates online. No hints of a firmware updating process hidden in their drivers. The CPU was something I didn't recognize at first. I posted



TITLE: facewhisperer

Design by: scanlime

REV: git

Date: 9/3/16 1:51 PM

Sheet: 1/1

the photo to Twitter, and Ladyada recognized it as a Sanyo/ONsemi LC87, an 8-bit micro that seems to be mostly used in Japanese consumer electronics. It comes in both flash and ROM versions, both of which I would later find in these tablets. Test points were available for an on-chip debugger, but I couldn't find the debug adapter for sale anywhere nor could I find any documentation for the protocol. I even found the firmware for this mysterious TCB87-TypeC debug adapter, and a way to disassemble it, but the actual debug port was implemented by a custom peripheral on the adapter's CPU. I tried various bit twiddling and pulse pushing in hopes of getting a response from the debug port, but my best guess is that it's been disabled.

At this point, the remaining options are more direct. A sufficiently funded and motivated researcher could certainly break out the micropositioners and acid, reading the data directly from on-chip busses. But this is needlessly complex and expensive. This is a USB device after all, and we have a perfectly good off-chip bus that can already do many things. In fact, when you attach a USB device to your PC, it typically hands very small pieces of its firmware back to the PC in order to identify itself. We think of these USB Descriptors as data tables, not part of the firmware at all, but where else would they be stored? On an embedded device where RAM is so precious, the descriptor chunks will be copied directly from Flash or Mask ROM into the USB endpoint buffer. It's a tiny engine designed to read parts of firmware out over USB, and nearly every USB device has code like this.

If this code is functioning properly, it will read back only the USB descriptor tables, and nothing else. If there's a bug in the size calculation, you may be able to request more data. If there isn't already a bug, you can introduce one via clock or power glitching.

Introducing a bug at just the right time can be tricky, so this is where it helped to build a new tool. Well, a tiny add-on for a masterful existing tool: the ChipWhisperer-Lite by Colin O'Flynn. The ChipWhisperer is an open source platform for side-channel power analysis and glitching. The joy of having both power analysis and glitching in the same platform is that they can be on the same reference clock. With one oscillator, you can deterministically step your target device through its paces, measure its activity via the power consumption waveform, and deliver glitches to specific clock cycles. By removing as many sources of jitter as possible, glitches can be delivered more reliably to the intended operation within the target's firmware.

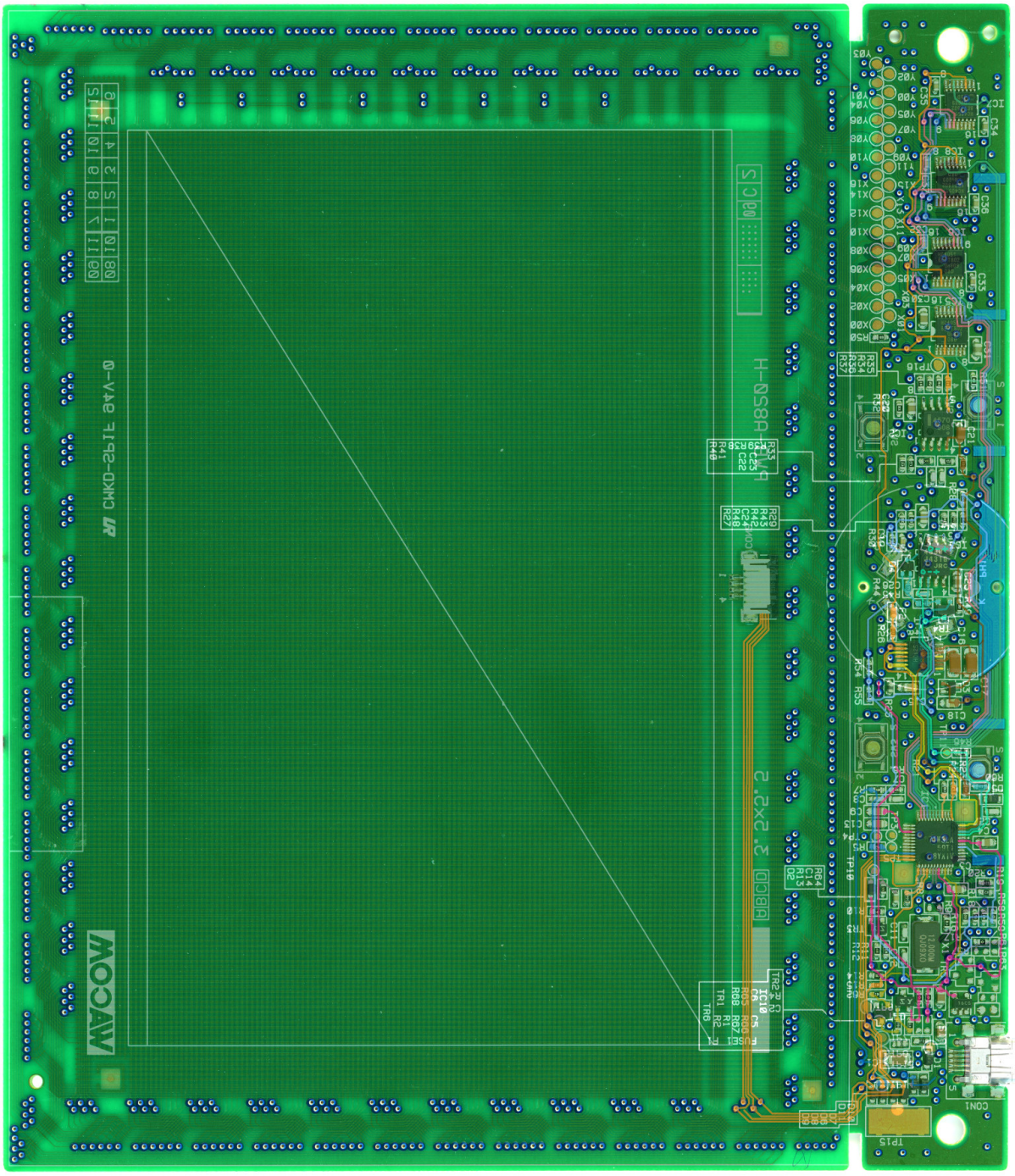
My humble addon is the FaceWhisperer, a USB host controller based on the MAX3421E chip, inspired of course by Travis Goodspeed's Facedancer21 tool. Whereas the USB host controller in your PC will be subject to many influences far outside your control, the USB host in the FaceWhisperer can be precisely synchronized with both the target device and the ChipWhisperer itself.

Putting everything on the same clock is necessary but not sufficient for cycle-accurate timing repeatability. The LC87, like many microcontrollers, will boot from a free-running RC oscillator before switching to the external clock under software control. This means it's necessary to synchronize with the running firmware somehow before starting up the USB host. In this case, I'm using a comparator input on the FaceWhisperer to precisely wait on a debug signal that indicates the beginning of a tablet scanning cycle.

The `GET_DESCRIPTOR` request we're interested in comes in several parts: a `SETUP` token that describes what descriptor we'd like to read, some IN tokens that each ask the device to send back one more packet, and finally an OUT for acknowledgment. These phases each drive a forgetful state machine that wakes up on each interrupt and leaves notes to itself for what needs to be done to the next packet. Unlike antique asynchronous serial ports, USB devices can never speak to the host unless they're offered a timeslot with an IN token, so no matter how badly we glitch the firmware we do need to follow this flow in order to read back data from the device.

This firmware extraction glitch works by disrupting the calculation and/or storage of the descriptor length, between that `SETUP` and the first IN. To extract as much data as possible, the `SETUP` can have a length limit of `0xFFFF` and the FaceWhisperer can continue spamming IN tokens until something fails. With this infrastructure in place, the ChipWhisperer's Glitch Explorer can hone in on timing offsets and glitch parameters that give us longer than usual descriptor responses. By briefly interrupting power at slightly different timing offsets after the `SETUP` packet, a variety of glitched behavior can be observed.

The descriptor we'll be reading is the USB Configuration Descriptor, typically one of the longest descriptors a device will provide. This device has a



34-byte descriptor that we'll be trying to glitch into something much longer. Usually the whole thing comes back in one packet:

```
IN
09022200010100801E0904000001030102000921
0001000122920007058103090004
rcode 5 total 34
```

Sometimes our glitches occur while copying the IN data itself. These aren't useful on their own, but they can give some feedback on how well the glitch is working:

```
IN
09022200010100801E0904000001030102000921
21FFFFFFFF20D227FFFFFFFFFF20
rcode 5 total 34
```

When you're getting close, you start to see non-corrupted descriptors that have a longer than expected length:

```
IN
09022200010100801E0904000001030102000921
0001000122920007058103090004090222000101
00801609040000010301020009210001000122920
00070581030900040902 3B000201008016090400
0001030102000921000100012292000705810309
0004090401000103000000092100010001220F00
0705820340000404030904 1E035700610063006F
006D00200043006F002E002C004C00740064002E
0010034300540045002D0034003500300010034
00540045002D0036003500300010034D00540045
002D0034003500300010034D00540045002D0036
0035003000680268016802680168026800 6803F0
00F001F003F002700170027000700370007003700
00B801B800B801B8
rcode 5 total 268
```

Only a little more of that, and we find a glitched configuration descriptor that's 65,534 bytes long, more than enough to reconstruct the entire 32 kB firmware ROM. You only get the memory prior to the descriptor if the address space wraps, but fortunately for us this was the case. All that's left is to determine the address offset by looking for clues like an IVT at the beginning or unused memory near the end of the image, and correctly align the resulting 32 kB image.

If you'd like to try this technique on your own devices with the ChipWhisperer, you can grab the PCB design and source for FaceWhisperer and play along.[12]

This sort of side-channel analysis still requires a bit of PCB surgery in order to set up the device's power rails and clock for glitching and monitoring. It also helps to have a reset signal and some sort of GPIO that can be used as a timing reference. It would be interesting future work to see how far this setup could be reduced. Could the glitching be performed solely via the USB port, even through whatever power regulation and conditioning the device includes?

4.2 Coding in Disappearing Ink

The documentation for the LC87 architecture is sparse. I eventually found an instruction encoding table buried in some product-line-specific appendix, but for a while the only resource I could find was a freeware toolchain, including a compiler and an on-chip debugger. I had already taken a look at this debugger in an attempt to awaken the debug port on my tablet. It wouldn't do much without this mysterious TCB87-TypeC dongle, but I tried simulating the TCB87 with a GreatFET that mostly just pretends things are okay and tells this RD87 debugger whatever it wants to hear. When I get the debugger to start up, it begins populating the hex views with zeroes. After a quick look with the USB analyzer, I easily find the requests that are the same size as the device's memory and begin answering those with my firmware dump. Now I have a debugger that I can use for static analysis!

I was looking for some kind of update mechanism. I would later discover that this tablet (firmware 1.16) used mask ROM whereas many earlier tablets (1.13) used flash memory. Those 1.13 tablets do seem to have a bootloader of some kind available, but I haven't looked into it yet. With the 1.16 tablet I had been analyzing, though, I became fairly certain there was no intended way to modify the device's program memory. This gave me a new constraint, which turns out to be interesting anyway: Turn the tablet into an RFID reader without modifying its firmware. We'll do this entirely via RAM and return-oriented programming.

The next step was much easier than expected. There was plenty of hidden functionality in the firmware. These are things that aren't part of any

[12] `git clone https://github.com/scanlime/facewhisperer`
`unzip pocorgtfo13.pdf facewhisperer.tar.bz2`

standard and aren't used by the official drivers, but presumably exist for factory test purposes. There's a mode you can put the tablet in which enables an additional USB endpoint that returns loads of timers and internal debug info. Oh, and there's a HID request that will just write exactly 16 bytes into RAM anywhere you like!

I think this was used in conjunction with another routine that isn't called anywhere, which tests the custom silicon Sanyo added for Wacom. Oh, custom silicon. I was hoping not to find that here. Newer tablets have chips that are obviously designed by Wacom to be complete analog frontends. I wanted to start with an older tablet that would have fewer custom parts. But perhaps the "W" in LC871W32 stands for Wacom. The analog frontend is made from discrete components in this tablet; multiplexers to select from an array of coils, op-amps to integrate the received signals, a buffer to excite the coils with a carrier wave. When I first looked at the circuit, it seemed like the 750 kHz carrier wave itself as well as the other timing signals would be generated using general-purpose peripherals on the micro. But when I look for the corresponding GPIO pins, nothing. More reverse engineering, and it was clear that I was facing custom hardware. I've been calling it `FEB0h`, after its I/O address. At first I thought it was a serial engine of some sort that was being misused to run the tablet, but now it's clear that this hardware is purpose-built. More on that later. For now, it's enough to know that the hardware or the mask ROM itself had enough engineering risk that they thought it prudent to include such a powerful test feature.

This is enough to start testing the waters and building up more and more complex ROP code. The ROM is only 32kB, and barely half full, but there are some useful gadgets. We can make function calls, do memcpy, RAM-to-RAM and ROM-to-RAM. Interrupts are tricky. I tried coexisting with them for a while, but had to give up on that due to USB packet corruption issues I couldn't track down. Write an arbitrary byte? Look up where we'd find that in ROM and do a memcpy. Loops are the slowest. These ROP stack frames can only execute once before they're corrupted, so we must copy the code each time it's run. It's slow, but we're doing arbitrary things to this peripheral that we haven't even written any code to. We can even return it to normal operation if we like, by jumping back to the main loop and restoring a normal stack.

This is not typically the sort of operation your OS requires elevated privileges for. The underlying Send Feature Report operation is typically associated with harmless device-specific features like toggling your keyboard LEDs, not with writing arbitrary instructions to a Turing-complete processor that is trusted by the OS just as much as you are. Applications can typically reserve access to any HID device that doesn't already have a driver loaded. It's easy to imagine some desktop malware that unloads or subverts the default driver long enough to load some malware into a peripheral's RAM without subsequent detection by either the user or the driver.

4.3 Amplitude Modulation Alchemy

Wacom pens and passive RFID cards are broadly similar, in that they both use a resonant LC circuit to pick up some energy from the reader's changing magnetic field, then they send back data bits with backscatter modulation, selectively shorting out the coil. The specific mechanism is a bit different though, and it will make our job harder. A typical 125 kHz RFID reader is sending out either a continuous carrier, or perhaps sending long bursts a few times a second to save energy. During this burst, the reader is continuously listening for a modulated response, with hardware filters specifically tuned to this job.

Wacom tablets, by contrast, are all about sequentially scanning an array of coils. This CTE-450 tablet has 12 short and wide horizontal coils on the front side (Y00 through Y11) and 17 tall and thin vertical coils on the back side (X00 to X16). When it has no idea where the pen might be, it has to scan everywhere. After locating the pen, it can adjust the scanning pattern to take differential measurements from the tablet coils nearest the pen coil. Instead of transmitting and receiving simultaneously, the filtering can be simplified by toggling between two modes. When transmitting, a 74HC125 buffer drives the coil with the tablet's carrier wave. During this time, the analog integrator is zeroed. Then the tablet switches modes, and begins integrating the received signal.

These resonant LC circuits are like electromagnetic tuning forks. An RFID tag or a Wacom pen have a tuning fork at a specific frequency, and some circuitry that communicates each bit by either damping the oscillations or letting them ring. The Wacom tablet shouts at the tuning fork's frequency, quickly and abruptly, and immediately listens for the reverberation. The whole protocol is designed around this mode switch. Gaps in the carrier indicate the bit boundaries, and longer bursts divide packets.

The trick here is to use this mechanism to read some common RFID access card. Between the slow return-oriented programming and the limited analog frontend, I picked an easy target for the PoC. The EM4100 is a common 125 kHz tag with a fixed 40-bit ID. It's no more secure than a pin tumbler lock for sure, but it isn't too far from the tags used in many access control systems.

The EM4100 pads the 40-bit code out to a 64-bit repeating pattern with the addition of a 9-bit header and a matrix of parity bits. Each bit is Manchester encoded; 0 becomes 10, 1 becomes 01. Each half-bit lasts 32 clock cycles, giving us a conveniently slow data rate.

The pulsed carrier is a problem. The RFID card does have its little tuning fork, and it keeps ringing a little bit, but not as much as you might think, especially when the EM4100 chip is trying to power itself from this stored energy and the external carrier has disappeared. A clock cycle or two, but not nearly as long as the tablet's A/D conversion takes. This little bit of unpredictability, though, has so far foiled every plan of mine to stay in sync with the signal in order to sample it at or below the bit rate. My workaround has been to use a short enough carrier pulse in order to have multiple samples per bit, allowing me to occasionally use a pile of filters and heuristics to recover the correct bits with appropriate deference to Nyquist. The problem with using a shorter carrier pulse is that it lowers our carrier duty cycle, delivering less power to the RFID card. So, there's a delicate balance: long enough to power the card, short enough for the resulting data to be intelligible through this intermittent sampling.

The returned signal is quite weak, since the tablet's filters are looking for resonance at a very different frequency. This is an area where I've seen much difference between individual RFID tags. Under unrealistic conditions, with the RFID tag placed directly on the tablet circuit board, many tags read successfully without much trouble. With an unmodified and fully assembled tablet, I've had very difficult to reproduce results, occasionally reading only one of the several tags I tried the setup with.

If you want to try this experiment or others, you can find my simple ROP toolkit and signal processing for the CTE-450 and try your luck with the return-oriented analog hacking.[13]

4.4 More to do

Although so far I've only managed to transform this tablet into an extremely bad RFID reader, I think this shows that the overall approach may lead somewhere. The main limitations here are in the reliance on slow ROP, and the relatively low quality A/D converter on the LC871. I've done my best to try

[13] `git clone https://github.com/scanlime/cte450-homebrew/`
`unzip pocorgtfo13.pdf cte450-homebrew.tar.bz2`

and separate the signal from the noise, but I'm no DSP guru. It's possible that a signal processing expert could be snooping tags with a better success rate than I've been seeing. As a proof of concept, this shows that the transformation from tablet to RFID reader is theoretically doable, though without a significant improvement in range it's hard to imagine this approach succeeding at reading access cards casually left against a victim's graphics tablet.

It could be interesting to examine newer tablets. The custom silicon in `FEB0h` turned out to be one of the best things about the CTE-450 tablet, making it relatively easy to change the timing and carrier frequency. If newer tablets have a nicer A/D converter and a programmable filter on the receive path, they could make a decent RFID reader indeed. A brief look at my newer Intuos Pro tablet shows a Renesas processor that likely has reprogrammable flash.

There's certainly more work to do in discovering the scope of devices vulnerable to glitched `GET_DESCRIPTOR` requests. What other devices that we usually think of as black-box peripherals might have firmware that can be read out, or RAM that we can temporarily hide code in?

It may be possible to mitigate these glitched `GET_DESCRIPTOR` firmware readouts by adding additional verification steps in the device's USB stack, which would each also need to be glitched. Reducing the number of invalid states that eventually result in spilling data will make the glitching process much more tedious.

In practice, though, I would argue that the best security is not to rely on secret firmware at all. Algorithms shouldn't need secrecy to keep them secure. Debug features that are too dangerous to leave should be disabled, not hidden. If any sensitive data must be reachable from the CPU, it should be unmapped whenever possible, especially when some USB controller asks for your life story.