PoC || GTFO;

brings that

OLD TIMEY EXPLOITATION

with a

WEIRD MACHINE JAMBOREE

and our world-famous

FUNKY FILE FLEA MARKET

not to be ironic, but because

WE LOVE THE MUSIC!

November 25, 2014

6:2 On Giving Thanks

6:3 Dolphin Emulator Internals (PPC)

6:4 TAR/PDF Polyglots

6:5 Pong Easter Eggs in VMWare

6:6 Anti-Emulation for MIPS

6:7 Cracking AngeCryption with `ECB.py`

6:8 PCB Reverse Engineering

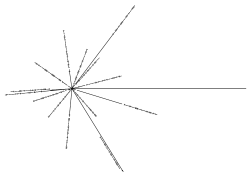
6:9 Davinci Self-Extractor

6:10 Observable Metrics

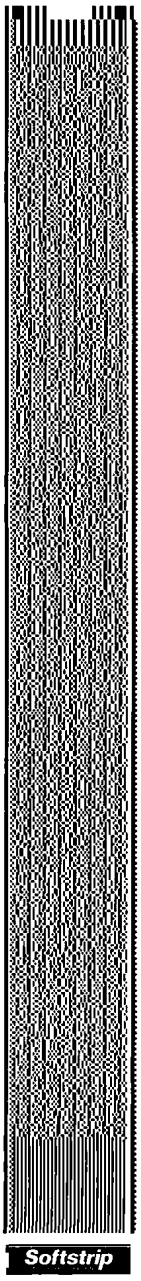
6:11 Donate to Laphroaig's 0day Charity

Plymouth, Massachusetts:

Published at Considerable Financial Loss by the
Tract Association of PoC||GTFO and Friends,
to be Freely Distributed to all Good Readers, and
to be Freely Copied by all Good Bookleggers.



€0, $0, £0. `pocorgtfo06.pdf`. Это самиздат; please copy this floppy!



Legal Note: Our intern has yet to forgive us for rejecting his copyright statement that repeatedly cites the Alien Tort Claims Act of 1789, and having blown our legal budget on scotch, there's nothing to threaten you with in this space. You should take this opportunity to make tons of paper and electronic copies to share with your friends.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror–don't merely link!–`pocorgtfo06.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse.

Technical Note: This issue is a polyglot with microdots that can be meaningfully interpreted as a ZIP, a PDF, or a TAR. It is filled with easter eggs, and if you are a very good reader, you will also hunt through it with a hex editor.

Printing Instructions: Pirate print runs of this journal are most welcome, but please do it properly! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland. Secret government labs in Canada may use P3 (280 mm x 430 mm) if regulations demand it. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge pocorgtfo06.pdf -o pocorgtfo06-booklet.pdf
```

F 8 C W (depuis 1929)

a le plaisir de confirmer son QSO

avec	date	heure	A1	R	
			A3	S	
		TU	BLU	T	MHz

Jean SERRIÈRE
4, Rue Alfred Dormeuil
78290 CROISSY SUR SEINE
FRANCE

Preacherman	Reverend Doctor Pastor Manul Laphroaig
Ethics Advisor	The Grugq
Poet Laureate	Ben Nagy
Editor of Last Resort	Melilot
Carpenter of the Samizdat Hymnary	Redbeard
Funky File Formats Polyglot	Ange Albertini
Minister of Spargelzeit Weights and Measures	FX

1 Sacrament of Communion with the Weird Machines

Neighbors, please join me in reading this seventh release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first six issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, or the fifth in Montréal, or the sixth in Las Vegas.

This release is dedicated to Jean Serrière, F8CW, who used his technical knowledge and an illegal shortwave transceiver to fight against the Nazi occupation of France. His wife Alice Serrière once, when asked "Where are the tubes?" showed occupying soldiers the leaky pipes in their basement.

In Section 2, the Pastor reminds us that there are things that we must be thankful for, with a parable freshly drawn from the Intertubes.

In Section 3, Fiora shares with us a collection of nifty tricks necessary to emulate modern Nintendo Gamecube and Wii hardware both quickly and correctly. Tricks involve fancy MMU emulation, ways to emulate PowerPC's `bl`/`blr` calling convention without confusing an X86 branch predictor, and subtle bugs that must be accounted for accurate floating point emulation.

Continuing the tradition of getting Adobe to blacklist our fine journal, `pocorgtfo06.pdf` is a TAR polyglot, which contains two valid PoC, as in both Pictures of Cats and Proofs of Concept. In Section 4, Ange Albertini explains how this sleight of hand is performed.

In Section 5, Micah Elizabeth Scott shares the story of the Pong Easter Egg that hides in VMWare and the Pride Easter Egg that hides inside that!

In Section 6, Craig Heffner shares two effective tricks for detecting that MIPS code is running inside of an emulator. From kernel mode, he identifies special function registers that have values distinct to Qemu. From user mode, he flushes cache just before overwriting and then executing shellcode. Only on a real machine—with unsynchronized I and D caches—does the older copy of the code execute.

In Section 7, Philippe Teuwen extends his coloring book scripts from PoC‖GTFO 5:3 to exploit the AngeCryption trick that first appeared in PoC‖GTFO 3:11.

In Section 8, Joe Grand presents some tricks for reverse engineering printed circuit boards with sand paper and a flatbed scanner.

Continuing this issue's theme of tricks that allow or frustrate debugging and emulation, Ryan O'Neill in Section 9 describes the internals of his Davinci self-extracting executables in Linux. Here you'll learn how to prevent your process from being easily debugged, sidestepping `LD_PRELOAD` and `ptrace()`.

In Section 10, Don A. Bailey treats us to a fine bit of Vuln Fiction, describing a frightening Internet of All Things run by a company not so different from one that shipped a malicious driver last month.

Finally, in Section 11 we pass around the old collection plate, because—in the immortal words of St. Herbert—*the PoC must flow!*

2 On Giving Thanks

a Sermon for the Holidays
by Pastor Manul Laphroaig.

The turkey is ready and waiting, neighbors, and so are the traditional arguments with loved ones around the dinner table. But let us spend a few moments reflecting on the few things besides the turkey and the family that we are thankful for, the things that shine on our sunny days and make the rainy ones possible to stand. Let us think of what keeps our worst nightmares at bay.

A wise neighbor once said, "I value Mathematics so highly because it leaves no place for hypocrisy and vagueness, my two worst nightmares." You might think, "How are these things the worst? I can think of a lot worse than those!" But it is so concise and true! Imagine a world where there would be no corner to hold against hypocrisy and vagueness, where any statement whatsoever could be twisted and turned by those who thrive on such twisting and turning to gain advantage of and power over their neighbors, where $2 + 2$ would indeed be, as an old Soviet joke put it, "whatever the Party orders it to be." Imagine a world where no false promise could be ever taken to account because the lying liars who gave it would fall back to the vagueness of their words every time. This would be a miserable world, neighbors, a nightmare world.

We get a taste of this nightmare every time politics forces its way into places that used to manage to keep it out—merit and skill no longer matter, demagogues get to run the place, sooner than later its original creators get thrown out, and then it collapses into mediocrity and pent-up unhappiness. Imagine that there would be no tool that would lay better to our hand than to that of the aggressors, that we had nowhere to retreat and nothing to fight them with that they could not suborn. Why fight if there is no chance to win, ever, anywhere?

Lucky for us, in every age there are things in the world that resist hypocrisy and vagueness, things that create the oases where we gather and hold.

We are doubly lucky because for us Mathematics has taken physical form. It has clothed itself in silicon and electricity, and now we can wield it not only among ourselves but also show it to others who need not understand its language, but are content to see its results. To see just how much luckier we are, neighbors, than the geeks of Leonardo da Vinci's times, just read his resume that he sent to the ruler of Milan. To support himself while exploring the niftiness and awesomeness of nature and math, he had few other options than promising to construct superior war machines. We are damn lucky, neighbors, that we can build machines that deliver better privacy rather than better war if we so choose!

No sooner did I write this, neighbors, than real life^{TM} provided a case study, as if on cue. Tor is run by evil scientists in the pay of the government! News around the clock, on this website only! Ominous geek conspiracy unmasked!

Tor, as you already know if you read its *About* page, was originally funded as a US Navy research project, and is still occasionally funded by some clueful parts of the US government that care about people getting news and other info that their governments happen to not approve of. Given that this sermon got to you neighbors by traveling for at least some of its path along a series of tubes ordered by another US military research agency, it is not surprising that such clue still exists; let's hope that it persists, neighbors, as we sure could use more of it, the way things are generally going in those quarters these days.

Thanks to this clue, and also to the selfless dedication of Tor developers who made this project go the way few government-funded projects ever do, we have the Internet-scale equivalent of a Large Hadron Collider for low-latency onion routing. Unlike the LHC, this experiment is not just open to the public, but also immediately useful. Which is where the "revelations" come in: are "evil scientists" tricking the public?

Luckily, Tor is science, and totally open science at that—the best kind that hides nothing. It requires no permission or special access to be attacked in the only meaningful way that scientific claims are questioned and their subject-matter is improved—by experiment. Indeed, many good neighbors did so and helped improve it—and you should read their papers, because their work is nifty[1]. And when you hear someone attack open science not with experiments or calculations but with FUD about money or attitude, either

[1] Especially because it's all open-access. Please enjoy the Freehaven Selected Papers in Anonymity. `http://www.freehaven.net/anonbib/`

that someone doesn't understand how science works, or has another angle.

There's a bar analogy for everything in life (it's a more fun cousin of the car analogy), so here's one for how this hustle works. Imagine that someone is loudly embarrassing himself and annoying neighbors in a bar with a foolish story. Being good neighbors, wouldn't you be moved to step in (hey, it's a bar *and* a good deed!) and gently correct him? Except, you discover that the bar has a hefty cover charge, and the loud silliness is actually quite profitable.

That's one bar it's good to pass, neighbors, because it's not in the business of enriching minds with good stories while cheering hearts up with a hearty drink. All it's serving is the poisoned Kool-aid of clickbait.

A clickbait purveyor[2] who happened to read the *About* section of the Tor website must have thought he struck a mother lode. An "evil scientist" story with a garnish of government conspiracy—what a clickbait oil well!

The "evil scientists" trope is a like perpetual motion machine for clickbait. Scientists aren't the most glib and suave communicators to begin with; they tend to become annoyed when bullshit is heaped upon them, letting their annoyance show. This in turn is clear proof that they are evil and holding something back! Quick, attack them again, and spare no personal detail, because there are hundreds of ways that the geeks are geeky, and for each one there are some folks that will be persuaded that geeks can't be trusted because of it.

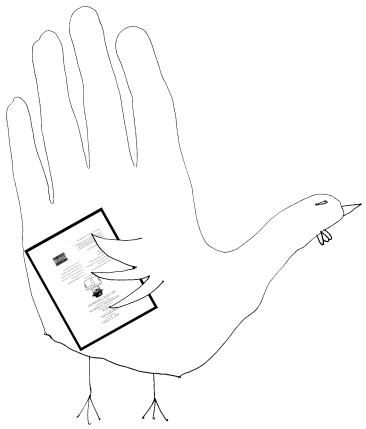
The point of all this noisy commotion, neighbors, is to make the public forget that science and technology are in the business of making things that can be judged on their own, regardless of their creators' or detractors' motives, personalities, employers or lack thereof, or in fact any other circumstances where FUD, vagueness, and hypocrisy may be brought to bear. A scientific artifact stands on its own, the same way a formula is either correct or meaningless, regardless of whose hand wrote it. Trying to guess what directed that hand is worse than pointless if the point is to know if we should put our trust in the artifact—because good motives don't make good science, and suspecting the scientist of a conspiracy adds precisely zero bits of information, and clouds thinking.

Over what criteria should one evaluate Tor, then? As one should any other engineered artifact: whether it does what it says on the label, whether it does anything *not* specified on the label, and whether the operating conditions under which it can successfully function are present. Are the operators of the nodes that make up your Tor circuit actually independent and uncompromised, or are Sibyl attacks an important concern—and from whom? Is there enough mutual information between packets entering and exiting Tor to deanonymize users—and from what perspective on the network is that information available?

In clickbait, you will not find these questions asked, much less their answers. Not sure whether an article's clickbait or not? Try suggesting to those responsible for it what questions they *could* have asked. If the answer is a wave of harassment rather than a follow-up, congratulations, you've found clickbait. Worse, you are in the land of hypocrisy and vagueness; get out fast.

Once we remember that, neighbors, the FUD clouds of zero-information verbiage dissipate, and the saving light shines through. Technology is not magic that must be judged only by the kind of witches and wizards who create it, tainted by evil or doom unbeknownst to mere mortals. It is knowable and dissectible, and our predecessors left us the greatest gift of understanding that, and of approaching it just so.

If we got any further out from under the shadow of vagueness and hypocrisy, it was thanks to that legacy and to that principle. And so we will walk out of this Valley of clickbait and bullshit, and we shall not fear, because they will hold no power over us. And for this we are thankful.



[2]Astronomy and astrology are not in the same business even though they both have to do with stars; so with journalism and clickbait generation. Be kind to good journalists, neighbors! They are few and far between, and their battles with bullshit tend to be a lot more uphill than ours.

3 Gekko the Dolphin

by Fiora

3.1 The Porpoise of Dolphin

Dolphin is one of the most popular emulators, supporting games and other applications for the GameCube and Wii game consoles. Featuring a highly optimized just-in-time (JIT) compiler and graphics unit that translates GPU opcodes into vertices, textures, and shaders, Dolphin is able to emulate almost all GameCube and Wii games at high speeds on a modern x86 CPU.

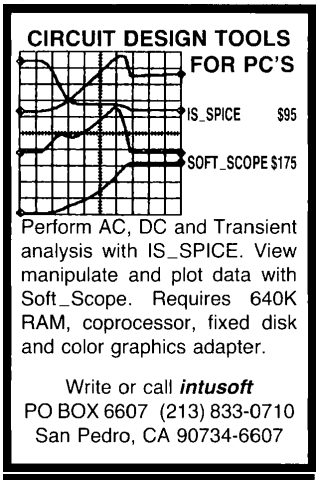
Instead of trying to do a detailed anatomy of the entire system, much of which is beyond my current understanding, in this PoC||GTFO article I'm going to focus on some particularly evil assembly optimizations and interesting bug fixes in the Dolphin JIT from the past two months—some large and dramatic, others small and elegant (or horrifically hacky, depending on your perspective!) But first, let's do a quick overview of how Dolphin works and some of the biggest difficulties inherent in Gamecube/Wii emulation.

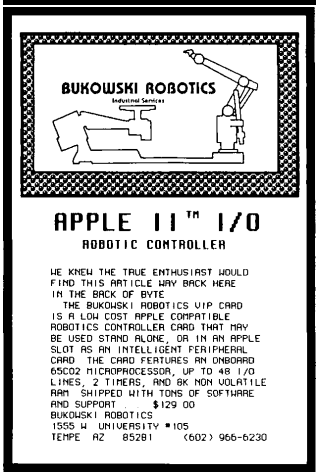
Dolphin's JIT is superficially similar to a typical PowerPC emulator, but things are not nearly so simple as they appear. The GameCube Gekko CPU (and the extremely similar Broadway CPU on the Wii) has a number of particularly odd features that aren't present on a typical PowerPC.

- A "paired singles" SIMD unit, somewhat similar to 3DNow! but complicated by some of PowerPC's inherent weirdnesses with floating-point (32 bit floats are represented as 64 bit internally, similar to x87).
- Built-in "graphics quantization" registers, which allow quantized loads and stores based on runtime-variable parameters, up to and including the data type to be converted to and from.
- A complex memory layout with mirrored regions and a slew of MMIO features, including a memory-mapped FIFO usually connected to the GPU, but which can also be repurposed for other uses by games.
- The ability to directly access—and modify—the active GPU frame buffer.
- Complex cache manipulation features, such as the ability to enable a "locked cache" and access memory as cached or uncached.
- A floating point unit with its own very unique definition of the word "multiply."

Making emulation even more difficult, games tend to abuse every aspect of the system imaginable, from the precise rounding of every floating point instruction to self-modifying code to behavior that isn't even defined in IBM's specification for the CPU. Additionally, games typically run in supervisor mode, giving them the ability to abuse a wide variety of features user-mode applications can't. All of this leads to severe limits on the shortcuts Dolphin can take; the most benign-seeming optimization often results in a slew of unintended consequences. Dolphin can't even reorder memory loads; an attempt to do this resulted in a real game failing because of exception handling semantics not being maintained.[3]

[3] Dolphin-Emu issue 5864





00AA AAAA 0000 0BBB 00CC CCCC 0000 0DDD	
AAAAAA	6 bit code representing the quantization factor (2^{-32} to 2^{31}) for loads.
BBB	3 bit code representing the data type for loads (float, S8, U8, S16, or U16).
CCCCCC	6 bit code representing the quantization factor (2^{-32} to 2^{31}) for stores.
DDD	3 bit code representing the data type for stores (float, S8, U8, S16, or U16).

Figure 1: GQR Register Format

Yes, there are applications that require precise emulation of MMU mechanics, including post-exception rollback. Yes, there are applications that intentionally try to execute an address of `0x00000001` to trigger a custom exception handler, and won't run unless this behavior is properly emulated. Yes, there are applications that modify code without properly flushing the CPU instruction cache and rely on the mere hope that the old code will have been since replaced in the cache. And yes, there are applications that may do many of these things with the intent of sabotaging Dolphin emulation.

Yet we still have to emulate a 729 MHz PowerPC CPU on a 2-3 GHz x86 CPU, all while trying to run programs that may very well be trying to prevent us from doing so.

3.2 Reserved bits are really just shy

A number of games were breaking in mysterious fashion with the JIT implementation of "paired singles" quantized loads and stores. Some crashed, while others had wildly broken lighting effects or other strange artifacts. Yet, even upon very close inspection, the JIT implementation was nearly identical to the (order-of-magnitude slower) interpreter implementation, which worked correctly. What could games possibly be doing here to break the JIT?

To understand this bug, it is crucial to understand the precise layout of the Gekko CPU's eight graphics quantization registers (GQRs). Each quantized load and each quantized store references one of these eight registers to act as its parameters. Figure 1 describes the format of the GQR registers.

The manual describes the other bits as being zero, but unfortunately, that isn't quite true. They were assumed to be zero, but the CPU never enforced this. Games could–and half a dozen games did–smuggle flag bits through these reserved register bits. Whether this was a bug, or perhaps done for some attempt at anti-emulation code, or even a strange sort of thread-local storage, we may never know.

The JIT's flawed assumption caused the implementation to either read out of bounds in the quantization array or even outright jump to an invalid function pointer. Fortunately, masking out those bits was just a single `and` operation; the main cost of this glitch was days of debugging by puzzled developers.

Since resolving this issue, I've written hardware tests to test reserved bits in other system registers too, which revealed all sorts of strange behavior. For example, the `XER` (fixed-point exception register), is laid out as follows.

```
1 [SO][OV][CA]0 0000 0000 0000 0000 0000 0AAA AAAA
```

`SO` is the summary overflow flag, OV is the overflow flag, and CA is the carry flag, with `AAAAAAA` being a 7 bit control code for string load/store instructions.

But on the Gekko, the actual bits that the CPU allowed to be set in XER were `0xE000FF7F`; it apparently supported setting the 8 bits in XER[16-23] even though it doesn't support the associated instruction, the string compare instruction `lscbx` (load string and compared byte indexed, similar to `rep cmpsb` on x86). I sincerely doubt any games used those bits in XER, but one can never be quite certain of such a thing.

3.3 Practice your multiplication, or you might become a GameCube CPU when you grow up!

For as long as it's existed, Dolphin has had trouble with replays, like those in racing games (Mario Kart, F-Zero) and fighting games (Super Smash Brothers). Emulation often desynced dramatically within seconds of the start of a console-recorded replay, with cars flying off the racetrack or Mario tripping off the side of the stage. The same happened in reverse, when emulator-recorded replays were transferred to a physical console. This was particularly dramatic in the case of Mario Kart's ghost feature, in which the game let you play against "ghosts" recorded by the developers of the game. The ghosts would very quickly drive into a wall, making victory quite trivial, if not very satisfying.

The source of this strange yet consistent desyncing was the way these games recorded replays. Instead of recording the movement of the karts or characters, the games record the player's input. This is a much more compact representation, but unfortunately, it means the most minuscule error on playback can accumulate until the result desyncs completely. To make replays, ghosts, and other similar features function correctly, Dolphin's floating point unit would have to match the Gekko's to the last bit of rounding.

For many months the Dolphin developer Magumagu exhaustively attempted to reverse-engineer the hardware FPU and make a software implementation. One by one, precise versions of instructions were implemented. Among the first victims were `frsqrte`, approximate inverse square root, and `fres`, the approximate reciprocal, which were replaced with table-driven versions matching the actual Gekko hardware. But it still wasn't enough; replays still constantly desynced, and bizarrely, the trouble seemed to trace back to the multiply instruction.

Some consoles do use non-IEEE floating point, like the Playstation 2; the curiosities of emulating this could make for an article of its own. Yet the Gekko was supposedly equipped with an IEEE-compatible floating point unit, denormals and all! How could multiplies on a GameCube give different results than on a typical desktop PC even with identical rounding flags set?

The problem, as Magumagu discovered, traced back to exactly how the floating point unit's internals were implemented. A double-precision float has 53 bits of mantissa; combined with three guard bits, this makes a 56 bit input. Accordingly, the Gekko had a 56x28 bit multiply and performed double-precision multiplies by combining the results of two 56x28 bit multiplies. Single precision multiplies were done with just one execution of the multiply unit.

But on the Gekko, all floating-point numbers are stored as 64 bit doubles. Single precision operations have reduced output precision and clamp their output to 32 bit precision, but are still stored as 64 bit doubles. Technically, according to the manual, you're not supposed to perform single-precision operations on double-precision values; the result is supposedly undefined. But, of course, countless games did it all over the place, so we still have to emulate it in a way that matches the behavior of the hardware.

Most single-precision operations seemed to be fine with double-precision input; a single-precision floating-point add, for example, seemed to be identical to performing a double-precision add and then rounding to single-precision. But, as Magumagu discovered, multiplies were their own unique brand of bizarre: they rounded the right hand side operand's mantissa to 25 bits of precision (for 28 including guard bits), then performed a 56x28 bit multiply. Note that 25 bits gives neither single nor double precision; it's something in between.

Fortunately, it took just four SSE instructions to perform this rounding operation for each multiply:

```
1 movapd xmm1, xmm0
  pand   xmm0, [truncate_mantissa] ; 0xFFFFFFFFF8000000
3 pand   xmm1, [round_bit]         ; 0x0000000008000000
  paddq  xmm0, xmm1
```

The overall performance loss was barely measurable compared to the literally dozens of games with fixed replays or physics, ranging from Zelda: The Wind Waker to Donkey Kong Country.

As Dolphin's primary tester, Justin Chadwick, once said, "Fiora, I hate how in your build the AI no longer bounces off the track in Mario Kart Wii. It makes it a lot harder to win."

3.4 Dolphin intentionally makes thousands of segfaults

Emulating one CPU's virtual memory subsystem on another CPU is hard. Doing so quickly is even harder. A direct approach would be to map one host page to each emulated page, but that's impossible on Windows because the Alpha AXP CPU didn't have a "load 32 bit integer" instruction. I'm not making this up.[4] The existence of MMIO, VRAM being directly mapped into CPU memory, and mirrored sections of the memory map certainly don't help.

The simplest approach would be to send every load and store through software address translation, but this proves to be fantastically slow. (Remember, we can only spend about three or four x86 cycles per Gekko CPU cycle!) Dolphin does support a variant of this as "full MMU emulation mode," which a few games with particular complex memory layouts do require. But for most games, it gets away with a vastly more elegant—or horrific—solution. Which one applies to you depends on how you feel about intentionally triggering thousands of segfaults.

For every memory access, Dolphin first tries to perform address constant propagation—if we know which area of memory an address is in, we can directly pass off the load or store to wherever it's supposed to go; usually a direct RAM access or a push to the FIFO. For the rest of the memory accesses, it shouts "YOLO" and just goes for it, with seemingly no care for what might happen if the access isn't to valid RAM.

But Dolphin has an ace up its sleeve: it's replicated the rough address space layout of the Gekko CPU in virtual memory using the operating system's shared memory features. Yes, that's a four gigabyte chunk of contiguous address space, including mirrored sections. (Addresses `0x8010000` and `0x0010000` map to the same place due to mirroring.) Sections that aren't directly mapped to physical RAM are marked as inaccessible.

When the "YOLO" access fails, a segfault is thrown by the operating system and caught by Dolphin's handler, which proceeds to backpatch the x86 code that caused the segfault to jump to a trampoline which then redirects to the slow, safe memory access handler. Thus, only the few memory accesses that actually go to non-RAM addresses take the slow route, while the rest are simply a `mov` and `bswap`.

This feature, called "fastmem," isn't at all new to Dolphin, but is nevertheless among a core reservoir of hacks that keep Dolphin's JIT fast. Tests suggest it provides at least a 15-20% CPU performance benefit over runtime address range checking.

3.5 Wasting all your cache is a good way to go bankrupt

As mentioned in the previous section, a few games make sufficient use of the GameCube's fancy MMU features that they need to take the slow path—full MMU emulation. While address translation (which is hopelessly unoptimized in Dolphin) is a significant cost, the greatest speed cost actually comes from the other consequences of full MMU mode. One of these is that it must check exceptions manually after every single memory operation, and if so, flush the register state, revert any address update that occurred in the load, and jump to the handler. It's all rather painful and an optimizer's worst nightmare, as it generates massive code bloat and places great constraints on instruction reordering and other aspects of optimization.

Because of all this, full MMU games tend to require incredible amounts of CPU power to emulate. While a few are at least playable on a very fast PC, others aren't so lucky. Rogue Squadron 2, for example, was developed by Factor 5, a game developer notorious for their ability to squeeze performance never thought possible out of consoles. In the Nintendo 64 era, they rewrote the GPU firmware to render five times more polygons than it was ever meant to. In Rogue Squadron 2, their incredible stressing of the Gamecube has led to a game that runs at half-speed in Dolphin on a 4 Ghz Intel Haswell CPU.

In addition, likely due to Dolphin's incomplete MMU implementation, a number of full MMU games simply don't boot at all: Rogue Squadron 3, Toy Story 3, and Disney Infinity among them. Particularly in the case of the latter, this might very well be anti-emulation code.

Profiling Rogue Squadron 2 with VTune suggested L1 instruction cache misses occurred at a rather high rate. The cost of cache misses is hardly a new topic in the optimization world, but code cache misses tend to be glossed over. Modern x86 CPUs have vast instruction fetch bandwidth, long pipelines to absorb fetch miss

[4] `unzip pocorgtfo06.pdf 64k.txt`

bubbles, and while performance can certainly be improved by reducing code size, it's often not considered a major factor.

Regardless of this, I figured I would see how much could be gained. I created a "far code buffer" in which to stuff all the rarely-used generated code (like exception handling and recovery for each memory access) instead of having it inline. Maybe this would get us a few percent of a speed increase?

With one rather simple commit, Rogue Squadron 2 sped up over 30% on my Ivy Bridge. The bloating of the generated code had cost so much that the CPU spent roughly 40% of its time sitting idle, waiting for new instructions to come in. The gain was even larger—over 50%—on another developer's Haswell, most likely because the Haswell has even higher instructions per clock-cycle count, and is thus even more susceptible to being front-end bound. Even in POV-Ray, a heavily floating-point-bound benchmark that doesn't use the MMU and was hardly known for its binary size, the gain was roughly 6% overall.

Never underestimate the value of instruction cache on modern CPUs. With a Haswell's four ALUs, two load units, and one store unit, it might very well be able to chew through instructions much, much faster than you can feed it.

3.6 It's normally abnormal for denormals to renormalize

I mentioned previously how the Gekko CPU internally stores all its floats—even 32 bit ones—as 64 bit doubles. This means that Dolphin has to convert floats to 64 bit on load, and convert back to 32 bit on store, at least if the `lfs` (load float single) and `stfs` (store float single) instructions are used. Hypothetically, if a value was loaded immediately and then stored, an optimizing recompiler could remove the conversion, but this can only sometimes be proven safely.

This wouldn't be an issue normally, outside of the small speed cost of a single extra conversion operation on each load and store. But unfortunately, yet again, games are not so kind. A strangely large number of games use `lfs` and `stfs` to copy integer data, which means the conversion process of float-to-double-to-float must be lossless, regardless of input. This would normally work, but at the same time, a large number of games also set the flush-to-zero (FTZ) floating point flag, which causes denormal floating point results to be set to zero by the CPU. Unfortunately, this also applies to our float-to-double and double-to-float conversions, so any game copying integer data that happens to look like a denormal float will have its data corrupted.

We can't turn off FTZ, because that would result in floating point arithmetic errors of the same sort that motivated the multiplication rounding changes mentioned previously. We also can't toggle FTZ off then back on again; the floating point control registers on x86 take upwards of fifty cycles to modify. The initial solution was to set rounding flags for SSE2, then do the load/store conversions using x87 (which, conveniently, doesn't even support FTZ). The one tricky part was fixing up the `NaN` flags afterward, as x87 handles `NaN` differently from SSE2, setting an exception flag instead. This is what the double-to-float code looked like.

```
movsd [temp64], xmm0
movsd     xmm1, xmm0
fld   [temp64]
ptest     xmm1, [double_exponent] ; 0x7FF0000000000000
fstp  [temp32]
movss     xmm0, [temp32]
jnc .dont_reset_qnan_bit
pandn     xmm1, [double_qnan_bit] ; 0x0008000000000000
psrlq     xmm1, 29
vpandn    xmm0, xmm1, xmm0
.dont_reset_qnan_bit:
```

This is better than fifty cycles per load and store, but it's still inefficient and gross enough to make x86 assembly writers everywhere squirm in discomfort. The overall speed penalty was around 20% on Super

Smash Brothers Melee—but there was little choice, since the alternative was inaccurate emulation that broke many games.

Fortunately, there is one other way. What if we just checked for denormals, passed them off to a slow, rarely-taken code path, and sent everything else through SSE? This has the bonus effect of not needing to fix up the `NaN` bit, since only denormals (not `NaN`s) would take the x87 path. The resulting code looks like the following.

```
movq      rax, xmm0
shr       rax, 55
sub        al, 0x6D
cmp        al, 3
jbe .x87conversion
cvtsd2ss xmm0, xmm0
jmp .continue
movsd [temp64], xmm0
fld [temp64]
fstp [temp32]
movss xmm0, [temp64]
.continue:
```

The comparison at the top is a bit tricky and designed to minimize code size, since this code will be duplicated countless times throughout generated JIT code. The only actual exponents that need to take the slow path are those in the range [`0x369`, `0x380`], but sending a few more to minimize the size of the comparison has negligible effect on performance (in this case, [`0x368`, `0x387`]). The comparison could be simpler if zeroes are also sent to the slow path, but testing shows that there's a very large proportion of zeroes—as many as a third of the inputs. With the check shown here, only 0.01% of floats take the slow path and the overall performance penalty for this change drops from 20% to 2%.

As a side note, the official IBM manual claims that the Gekko/Broadway CPU uses denormals-are-zero (DAZ) in addition to FTZ when the non-IEEE (NI) flag is set. Curiously, actual hardware testing shows that the CPU doesn't ever seem to actually do this.

3.7 Hey I just `RET` you, and this is crazy, but here's my address, so `CALL` me maybe?

Modern x86 CPUs typically have a built-in return stack, designed to predict where a `ret` instruction is heading, with the assumption that every call is paired with exactly one `ret`. This is a pretty good assumption, and in the rare cases where it fails, the performance cost is typically equivalent to a branch misprediction. Without this prediction, a return would be relatively costly and difficult to predict—little different from an indirect branch `jmp [rsp]` or similar.

PowerPC has its own similar call and return instructions: $\widehat{\text{bl}}$ (branch with link) and `blr` (branch to link register). The first jumps to a location and stores the old location in the link register (the return address), while the latter jumps to the location stored in the link register. When emulating `blr`, Dolphin treats it as an indirect jump to the link register. This is the natural translation for such an instruction, but it is costly from a branch misprediction standpoint, since such a branch is extremely difficult to predict correctly. Profiling shows a non-trivial number of micro-ops lost to branch mispredictions.

Comex's idea was to re-use the CPU's existing return prediction stack. On a `bl` instruction, instead of jumping to the target function, he would push the emulated destination address onto the stack and then `call` the target JIT'd function. When emulating a `blr` instruction, instead of jumping to the given link register, he compares the link register against the one stored on the stack at `[rsp+8]`, and if the two match, returns with `ret`. If functions call and return as expected, this approach should give near-perfect branch prediction. Despite the seeming increase in instruction count, this led to roughly an eight percent overall speed increase across nearly every game merely from improved return prediction.

The one danger of this is the possibility of the stack overflowing. If a game uses `bl` without an associated `blr`, the return stack will continually grow until Dolphin crashes. Comex's first solution was to clear the

stack whenever a misprediction occurred; this reduces the problem to the pure evil case of an application that used `bl` hundreds of thousands of times in a row without any `blr`. Out of curiosity and being a bit pedantic about correctness, he decided to support this case as well, writing a short test case that triggered the problem and setting up guard pages and extending the signal handler to catch any failure.

The core concept of this optimization is not too different from fastmem. Hijack a hardware CPU feature (in that case, memory protection, in this case, return address prediction) and use it to help emulate the same feature of the target CPU, even if it wasn't really intended for that purpose.

3.8 Through the `SUBFIC` and the `SRAW` we carry on

Like x86, PowerPC has a number of instructions that set flags based on their result. Unlike x86, there are two ways in which this can happen. There's condition flags (`GT`, `LT`, `EQ`, `SO`) which can be set by a comparison operation or an arithmetic instruction with the `Rc` bit set. This is a lot more convenient than x86, because one can generally avoid clobbering the flags when they're not needed, which makes code more efficient and, coincidentally, emulation easier.

Carry flags, on the other hand, are not quite so friendly. Some common instructions set carry unconditionally (`subfic`, `sraw`, `srawi`), enough so that carry calculation becomes a significant cost even in code that doesn't make heavy use of carry bits. The calculation of carry bits for `sraw` and `srawi` in particular is a bit non-trivial, easily requiring a half-dozen or so extra instructions on x86 to emulate.

The first step to optimizing carries was to enhance `PPCAnalyst`, the class that performs dependency analysis on instructions. If an instruction calculates a carry bit, but that bit is overwritten before being used or before reaching a JIT block exit, we can omit the calculation of that carry bit entirely.

`PPCAnalyst` also has an instruction reordering pass that uses dependency information to reorder instructions wherever it can be sure doing so is safe. This was originally just used to move comparison instructions next to branches so the two can be merged, but it can be extended to support a wide variety of operations.

I modified the instruction reordering pass to attempt to "stick" pairs of carry-using instructions next to each other. A large number of common PPC idioms use sequences such as `subc`+`subfe`; not merely arithmetic on variables larger than the register size. One example is `r0 = (r1 != r2)`.

```
subf  r3, r1, r2
addic r0, r3, -1
subfe r0, r0, r3
```

The PowerPC Compiler Writer's Guide lists a number of these in the appendix.[5]

The third and final step was to take advantage of this; if the next instruction is going to consume the carry bit, take advantage of the x86 carry flag instead of storing the carry bit in the emulated CPU state. This is a slightly tricky (and limited) optimization, since it requires the instructions to follow each other directly, since most instructions will clobber the x86 flags.

Combined with the "sticky" reordering, these changes were able to drastically reduce instruction count in carry-heavy code; some recompiled sequences dropped in size by a factor of two or more. Some games, such as Virtual Console games (an emulator inside an emulator!) went as much as 12% faster just with these carry optimizations.

An interesting future optimization might be to recognize some of the aforementioned multi-instruction compiler idioms and transform them into equivalent idiomatic x86 code; this could be even better than merely optimizing the individual instructions!

3.9 Capturing performance from the flags

As mentioned in the previous section, many integer operations, such as comparisons and operations with the `Rc` (record) bit set, have the ability to set result flags in the PowerPC condition register. The condition

[5] `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600755 8C6`

register is split into eight 4 bit sections, each of which represents one result, consisting of the `LT`, `GT`, `EQ`, and `SO` flags. This is in sharp contrast to x86, for which most instructions set flags unconditionally. It only has a single condition flags register instead of eight.

Emulating operations on these flags efficiently is critical to performance in Dolphin. It's often difficult to prove that an update to the flags register won't be used again following its most immediate use (e.g. a conditional branch), so the relevant calculations can't be omitted.

Delroth and Calc84maniac discovered a brilliant way to optimize Dolphin's internal flag representation to minimize the work required to set and read flag bits. These two operations represent the vast majority of operations on flags; everything else, such as boolean operations between flag bits and reading out the flags register, is practically a rounding error by comparison. In addition, reading out flag bits is done almost entirely by conditional branch operations.

The flag representation they invented involves the flags being stored as a 64 bit integer. Bit 63 is equal to `!GT`, bit 62 equal to `LT`, bit 61 equal to `SO` (a flag not fully emulated by Dolphin, but also rarely used except as the output of a boolean flag operation), bit 32 always set, and bits 0-31 set to zero if `EQ`.

This representation has the useful property that it can be calculated using a single instruction from the result of any integer operation; a 32− >64 bit sign extend (`movsxd` on x86_64). Individual flags can also be read out with single operations:

```
GT = (s64)CR > 0
LT = CR & (1 << 62)
EQ = (s32)CR == 0
SO = CR & (1 << 61)
```

While this dramatically complicated operations such as loading the flags register, the overall performance effect was tremendous. Performance improvements in typical games ranged from six to fourteen percent merely from being able to omit most of the instructions (and code bloat) involved in flag calculation. This change also inspired later optimizations, like splitting carry bits into their own emulated register instead of storing them in `XER`. There's no requirement that an emulator maintain the same data representations the ISA describes, so long as it transparently performs whatever conversions are necessary for correct emulation.

3.10 With Dolphin, Wii have a bright future

Dolphin still has a long way to go. The graphics engine is imperfect and still missing a few rather difficult features, like zfreeze and OpenGL line-width support. Dual-core mode is still sometimes a bit finicky with timing-sensitive games. GPU to CPU data transfer can be a speed issue, as well as vertex loading for geometry-heavy games. There are still many driver issues, like the long compilation times for shaders, that cause unwanted stutter and slowness.

The HLE audio engine is good but not perfect, with some games still requiring low-level emulation to avoid glitches. Countless minor bugs, from subtle depth buffer issues to issues with non-normal floating point numbers and console glitches not being reproducible in Dolphin, still exist. On the CPU side, even with many optimizations, some games are still slow, and a few still don't even boot properly.

But improvements like these are a start. Already, many games that were far too slow to be playable on all but the fastest overclocked Haswell CPUs are accessible to a much wider audience. And while Dolphin is not and probably never will be a perfectly cycle-accurate emulator (in fact, because of DVD read times and NAND write times, no two physical consoles will even produce identical results!), it may now be accurate enough to create at least some console-verifiable replays and speed runs.

Figure 2 gives some examples of the performance improvements, measured on a variety of synthetic benchmarks and games known for being performance-intensive, between revision 2301 (late July of 2014) and revision 3378 (late September of 2014), as measured on my Ivy Bridge CPU.

Dolphin is hardly a new project; it was open-sourced six years ago and developed as a closed-source project for many years before that. It's far too easy to assume that relatively stable, mature projects don't

POV-Ray	62%	faster
LUA “binary trees” benchmark	48%	faster
Sonic Colors	39%	faster
Rogue Leader	103%	faster
F-Zero GX	110%	faster
The Last Story	38%	faster
Xenoblade Chronicles	40%	faster

Figure 2: Dolphin Performance Improvements

have much room for improvement; as new contributors, we have to resist the urge to shy away from projects like this, because often there are still vast gains to be had.

Thank you so much to Comex and Delroth for their part in these two months of incredible CPU emulation performance improvements. Thanks also to Justin Chadwick (JMC4789) for his unmatched testing and bug bisection skills across hundreds of games, as well as the monthly Dolphin progress report writeups. And thanks to all the other devs: Ryan Houdek, Skidau, Lioncash, Shuffle2, Magumagu, Calc84maniac, Rachel Bryk and many others, for their tireless work on the other aspects of Dolphin, bug fixes, and assistance with the endless ignorant questions I asked on the way to learning the inner workings of Dolphin's CPU emulation engine.

Dolphin has been the most approachable project of any I've yet tried to contribute to, from the helpful developers to the relatively clean codebase. I somehow managed to become the go-to woman for the JIT in a mere six or so weeks, despite having never conceived before that I could ever contribute meaningfully to an open source project.

For anyone looking to contribute, there's an abundant supply of interesting (or terrifying, depending on your perspective) emulation bugs just itching for someone to attack with the single-step debugger and `printf` hammer. Plus, with the brand new 64 bit ARM JIT, there are countless instructions that still need implementations—and there are certainly lots of missing optimizations for the x86 JIT too. Drop by #dolphin-dev on Freenode or drop us a pull request—any help is always appreciated!

4 This TAR archive is a PDF! (as well as a ZIP, but you are probably used to it by now)

by Ange Albertini

In this article we'll build a TAR/PDF polyglot file with a few simple tools that you already have if you write in TeX or LaTeX (if not, take a couple of days to learn—wouldn't it be just spiffy to submit your very own PoC||GTFO piece in ready-to-go LaTeX?).

4.1 What is a TAR file?

TAR, written in the days when tape drives were the only serious form of backup, stands for TApe aRchive. Not surprisingly, its design is tightly coupled with the mechanics of tape drives. Those drives were made by IBM and were invented for the IBM 650, which was produced in 1953.

Accordingly, in those archives files are stored without compression, lengths and checksums are stored in octal, and everything is 512-byte block based. Respect old age, neighbors—and remember that your own modern technology might not survive that long.

4.2 Abusing the format

A TAR file starts with a fixed-length record of one hundred bytes, where the archived file's original name is stored, padded with zeros.[6] We can abuse this record to store a PDF header and a dummy stream object to cover the rest of the archive.

We'll let `pdflatex` build the dummy stream object for us from a .TeX source. We just need to declare this object (with no compression) right after the `\begin{document}`:

```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
        file {archive.tar}
\endgroup
```

We then need to move the stream content so that it virtually starts at offset 0, fix the file name, and insert a valid `%PDF-1.5` signature.

After the initial hundred byte record, a TAR file contains a header checksum. We need to fix it, because unlike many other checksums, it is actually enforced. The fixing isn't too difficult, but the format is nevertheless rather awkward. Here is the procedure, with a python script to perform it.

1. Overwrite the checksum (at offset `0x94`, 8 bytes long) with spaces.
2. Add all the unsigned bytes of the header.
3. Write this value as octal, with leading zeroes.
4. End the checksum with a NULL character at the 6-byte offset into the field.

```
OFFSET = 0x94
# Wipe the checksum field with spaces.
for i in range(8):
  header[i + OFFSET] = " "

# Sum all bytes of the header to an unsigned int.
c = 0
```

[6] If the name is longer, something called a PaxHeader is used instead; we've come a long way since the 1950s, neighbors!

```
for i in header:
    c += ord(i)

# Store the unsigned int in octal, followed by NULL then space.
for i, j in enumerate(oct(c)):
  header[i + OFFSET] = j

header[OFFSET + 6] = "\0"
# The required space was already there.
```

Now our TAR checksum is valid again, with an archived file name buffer that has been abused to contain a valid PDF header and a stream object. Enjoy!

```
manul:pocorgtfo pastor$ xxd pocorgtfo06.pdf | head -n 21
0000000: 2550 4446 2d31 2e35 000a 25d4 c5d8 0a31  %PDF-1.5..%....1
0000010: 2030 206f 626a 203c 3c0a 2f4c 656e 6774   0 obj <<./Lengt
0000020: 6820 3830 3934 3732 2020 2020 0a3e 3e0a  h 809472    .>>.
0000030: 7374 7265 616d 0a65 0000 0000 0000 0000  stream.e........
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 3030 3030 3634 3400 3030 3030  ....0000644.0000
0000070: 3736 3400 3030 3031 3034 3000 3030 3030  764.0001040.0000
0000080: 3030 3030 3030 3000 3132 3431 3435 3637  0000000.12414567
0000090: 3137 3200 3032 3031 3631 0020 3000 0000  172.020161. 0...
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000100: 0075 7374 6172 2020 004d 616e 756c 0000  .ustar  .Manul..
0000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000120: 0000 0000 0000 0000 004c 6170 6872 6f61  .........Laphroa
0000130: 6967 0000 0000 0000 0000 0000 0000 0000  ig..............
0000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

P.S.: Sadly, that's not all we needed to do. Just when we thought that our polyglot finally worked well on all readers, it turned out that some further edits broke it on `Preview.app`, for no apparent reason, and in a weird way. Namely, `Preview.app` wouldn't display the contant width fonts in our PDF *unless* the PDF signature was placed exactly at offset 0.

Choosing between our Apple readers not being able to enjoy this special issue, having to debug the `Preview.app`, having to reinvent font storage, and missing our deadline, or putting the PDF signature back at offset 0, we chose the latter. With luck, we'll just sacrifice a single 512 byte block and one junk filename to improve our PDF's compatibility.

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

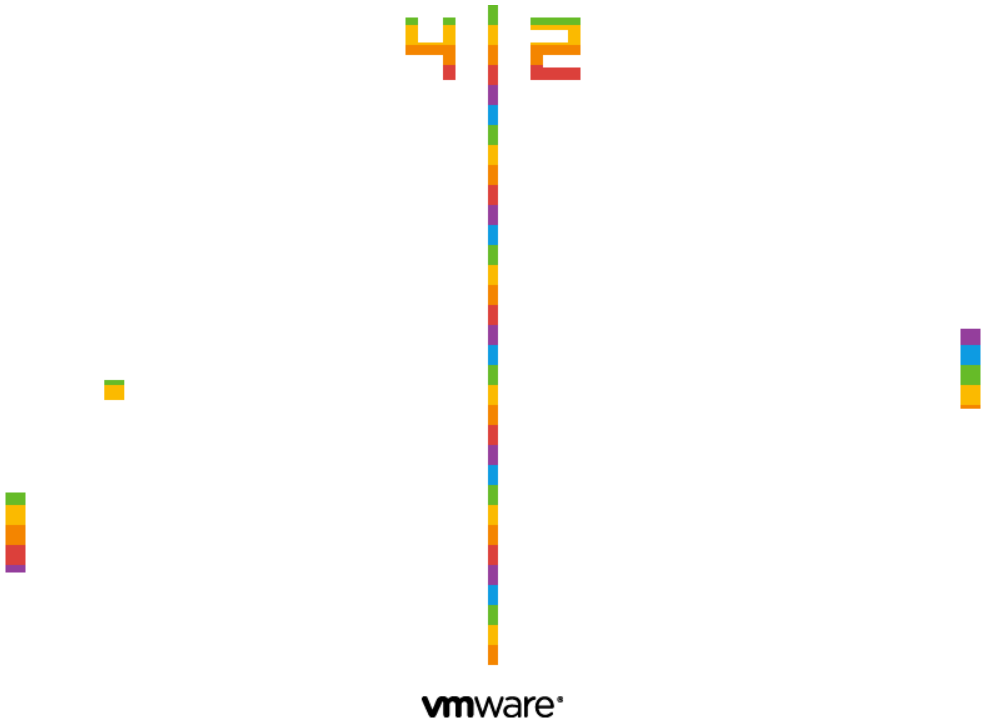


Figure 3: VMWare Pride

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.

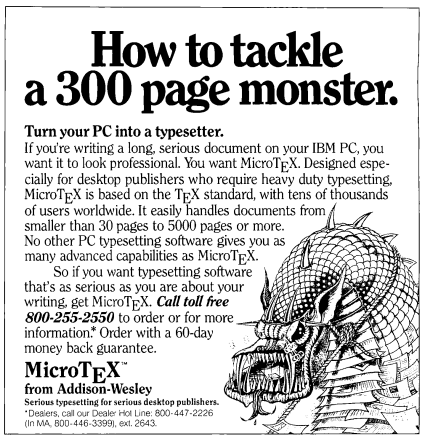
With Metalkit, we can try to move past that and see the BIOS as yet another niche where we can find reusable gadgets. If we can stomach a switch to 16 bit Real Mode and back for each batch of sectors, we can use the BIOS to read from the bootup disk (whatever stack of emulations that may be) into a small 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



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.

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

-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.

6 Detecting MIPS Emulation

by Craig Heffner

In this article, we'll look at some handy tricks for detecting the difference between real MIPS hardware and the Qemu emulator. First, in Section 6.1, we'll look at special function registers whose values in the emulator reveal the use of Qemu. Then, in Section 6.2, we'll intentionally run code which has a pending overwrite in the data cache to determine whether the instruction and data caches are synchronized with one other, as they are in Qemu but are not in real hardware. The techniques presented in this article were tested on Qemu v2.0.1.

6.1 Detection through hardware registers

Qemu can be identified with a reasonable level of certainty by examining discrepancies in the MIPS `CP0` (Coprocessor0) registers. The most obvious register to examine is the `PRId` (Processor ID) register, shown in Figure 4.

```
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                    |Company  Options|    Company ID   |      CPU ID     |     Revision    |
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
QEMU                |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 0 0 0 1 1|0 0 0 0 0 0 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Atheros AR7240 SoC  |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 1 0 0 1 1|0 1 1 1 0 1 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Ralink RT3352F SoC  |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 1 0 1 1 0|0 1 0 0 1 1 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Company Options	Reserved for use by the manufacturer.
Company ID	Uniquely identifies the manufacturer, but is set to 0 for older processors as it was not defined in the MIPS specification.
CPU ID	Identifies the specific MIPS CPU type. (MIPS 4KC, MIPS 24K, etc)
Revision	Used to specify the CPU core revision number.

Figure 4: Processor ID (`PRId`) Register

The `PRId` register can be read using the `mfc0` (move from coprocessor0) instruction.

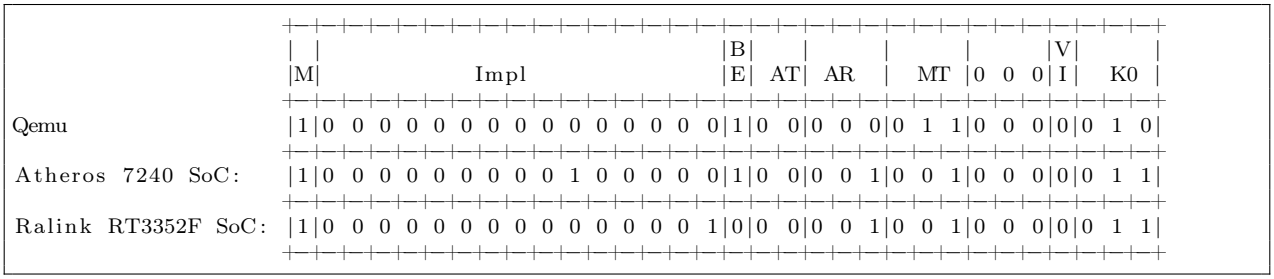
```
mfc0 $t0, $15   ; Move CP0 register 15 (PRId) into general purpose register $t0
```

Figure 4 also shows the differences between Qemu and two common system-on-chip devices that are found in real hardware. Note in particular the differences in the `Revision` field. Qemu sets this field to all zeros regardless of which MIPS core is being emulated, but most real-world systems will have this field set to a non-zero value representing the major/minor/patch version of the MIPS core in use by that CPU.[9]

It is also useful to examine the `Config` register. Much like `PRId`, the `Config` register can be read using the `mfc0` instruction.

```
mfc0 $t0, $16   ; Move CP0 register 16 (Config) into general purpose register $t0
```

[9] Programming the MIPS32 24K Core Family, Section 2.2



	M	Impl	BE	AT	AR	MT	0 0 0	VI	K0
Qemu	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1	0	0	0 1 1	0 0 0	0	0 1 0
Atheros 7240 SoC:	1	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0	1	0	0	0 1 1	0 0 0	0	0 1 1
Ralink RT3352F SoC:	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	0	0	0	0 1 1	0 0 0	0	0 1 1

`M`	`1` if there is another config register. (`Config1`)
`Impl`	implementation specific.
`BE`	`1` if the processor is big endian, `0` for little endian.
`AT`	Specifies whether the processor supports MIPS32 (0) or MIPS64 (1 == MIPS32 address map; 2 == full address map) encoding.
`AR`	Architecture Revision level (0 == MIPS32/64 release 1; 1 == MIPS32/64 release 2).
`MT`	Specifies the MMU type.
`0 0 0`	Unused
`VI`	Set to 1 if the L1 instruction cache uses virtual tagging.
`K0`	Specifies the MIPS kseg0 memory region's caching behavior.

Figure 5: `Config` register

Again, we can find some general differences in register values between different CPUs, which are shown in Figure 5. Most notably, `Impl` is zero in Qemu, while the Atheros and Ralink chips have this field set to non-zero values. The PIC32 datasheet also notes that it uses these bits to store information regarding segment caching and the SRAM bus interface.[10]

These register variations are generally reliable, and are particularly applicable if you expect to only run on one particular CPU, such as an exploit for a specific target.

6.2 Detection in Linux user space

Examining CPU hardware registers requires execution in kernel mode. But, for many Linux based MIPS devices, you may be executing from Linux user space. Here, you may simply examine `/proc/cpuinfo`, which in Qemu typically looks something like the following:

```
root@qemu:~# cat /proc/cpuinfo
system type             : MIPS Malta
processor               : 0
cpu model               : MIPS 24Kc V0.0  FPU V0.0
BogoMIPS                : 2097.15
wait instruction        : yes
microsecond timers      : yes
tlb_entries             : 16
extra interrupt vector  : yes
hardware watchpoint     : yes, count: 1, address/irw mask: [0x0ff8]
ASEs implemented        : mips16
shadow register sets    : 1
core                    : 0
VCED exceptions         : not available
VCEI exceptions         : not available
```

[10] PIC32 Reference Manual, `61113E.pdf`

First, most real MIPS systems will set `system type` to reflect the SoC vendor, such as "Ralink SoC" or "Broadcom BCM5357 chip rev 2". It would be extremely unlikely to see MIPS Malta on a production system.

More importantly, `BogoMIPS` as reported in Qemu is a reflection of the *host machine's* CPU speed. 2,097 BogoMIPS would be insane for a real MIPS processor, which typically clocks in around 400MHz. More realistic `BogoMIPS` values for MIPS CPUs would be in the 200-300 range.

6.3 Execution-based detection

While the above detection methods are useful, they could easily be changed or patched, either by an end user or in future Qemu releases. A far more reliable method of detection is through the use of fundamental architecture features that are not properly emulated by Qemu and not easily implemented.

Qemu can be reliably detected by exploiting cache incoherency, which is inherent in MIPS CPUs but absent from Qemu.[11]

The MIPS cache is divided into two sections: one for instructions, and one for data. When data is written to memory, that data is first stored in the data cache, and is eventually written back to main memory at a later time. Instructions, as you may well guess, are cached in the instruction cache.

This is a common issue during MIPS exploitation. Let's say that we write some shellcode to a buffer; that shellcode is treated as data, and cached in the data cache. If we try to jump into that shellcode, however, the CPU will go looking for it in the instruction cache; since it is not cached there, the CPU then fetches the instructions from main memory. But the shellcode isn't in main memory, it's in the data cache!

This problem is typically mitigated by first flushing the data cache back to main memory before jumping into the buffer containing the shellcode. Cache flushes can be performed explicitly in MIPS through the `synci` or `cache` instructions, or by simply waiting a bit (e.g., `sleep`(1)) and letting the kernel do a cache flush, which will typically need to happen periodically anyway.

Qemu does not even try to emulate this cache behavior, and we can use that to our advantage by
1) writing a block of code to an address in memory,
2) executing `synci` to make sure the code is written back from the data cache to main memory,
3) writing a second block of code to the same address in memory, and then
4) immediately jumping to the memory address.

When running on MIPS hardware, the second code block is still sitting in the data cache, and the *first* block of code will be fetched from main memory and executed. However, in Qemu, since caching is not emulated, the second code block will overwrite the first, and the *second* block of code will be executed.

Thus, we can execute two completely different sets of code from the same memory address; one piece of code will be executed when running in Qemu, and the other piece of code will be executed when running on real MIPS hardware:

```
1  /*
   * PoC code which executes different pieces of code from the same address
3  * in Qemu vs real MIPS hardware.
   *
5  * On real MIPS hardware, main should return 1.
   * In Qemu, main should return 2.
7  *
   * Tested against Qemu 2.0.1 and a Broadcom BCM5357 (MIPS 74K) SoC.
9  *
   * Requires a MIPS32r2 compliant compiler.
11 */

13 #include <stdio.h>
   #include <stdlib.h>
15 #include <string.h>

17 #define CODE_SIZE 8
```

[11] Linux MIPS Wiki, Qemu Processor

```
/*
 * ret1 contains a MIPS function that returns 1.
 * ret2 contains a MIPS function that returns 2.
 */

/*
 * Big endian
 *
char ret1[CODE_SIZE] =
     "\x03\xe0\x00\x08"  // jr $ra
     "\x24\x02\x00\x01"; // li $v0,1
char ret2[CODE_SIZE] =
     "\x03\xe0\x00\x08"  // jr $ra
     "\x24\x02\x00\x02"; // li $v0,2
*/

/* Little endian */
char ret1[CODE_SIZE] =
         "\x08\x00\xe0\x03"  // jr $ra
         "\x01\x00\x02\x24"; // li $v0,1
char ret2[CODE_SIZE] =
         "\x08\x00\xe0\x03"  // jr $ra
         "\x02\x00\x02\x24"; // li $v0,2

int main(void) {
    int(*s)(void);
    int retval = 0;
    char buf[CODE_SIZE] = { 0 };

    /* The s function pointer points to buf */
    s = (void *) &buf;

    /* 1. Copy ret1 into buf (ret1 is now in the data cache)
     * 2. Execute the synci instruction to flush the data cache (ret1 is now in main memory)
     * 3. Copy ret2 into buf (ret2 is now in the data cache)
     * 4. Call the function located in buf (should fetch and execute ret1 from main memory)
     */
    memcpy(buf, ret1, sizeof(buf));
    asm ("synci 0(%0)": : "r" (buf));
    memcpy(buf, ret2, sizeof(buf));
    retval = s();

    printf("retval = %d\n", retval);
    return retval;
}
```

Because `synci` is not a privileged instruction, this method can be used in both user and kernel space. The only downside is that `synci` was not introduced until MIPS32r2, so older MIPS processors don't support that particular instruction. Since MIPS32r2 was introduced in 2003, it's unlikely that this will be an issue unless you're dealing with an older processor; in such an event, you'll need to use some alternate method of flushing the cache. This can be done in kernel space with the `cache` instruction, or in Linux user space, you can simply replace `synci` with a call to `sleep`(1).

It's worth noting that in theory, the second block of code (`ret2`) could be executed when running on real MIPS hardware if the kernel flushed the cache behind your back in between the time that `ret2` is copied into `buf` and the time that you actually call into `buf`. However, this would be a very unlucky edge case which I have yet to encounter in practice, provided the time between the second `memcpy` to `buf` and the call to `buf` is minimized. `ret1` is never executed in Qemu.

7 More Cryptographic Coloring Books

by Philippe Teuwen

7.1 Weird crypto

In PoC||GTFO 5:3 we taught you kids why ECB is a weak encryption mode, as helpfully shown by the `ElectronicColoringBook.py` script.[12] As you may have guessed, we'll see that in some circumstances CBC deserves the same treatment!

Don't worry, though! Most of the time CBC mode is fine, but sometimes weirdos like our buddy Ange Albertini do impossibly fancy things with crypto such as *AngeCryption*. I wouldn't risk offending our PoC||GTFO's loyal readers by explaining AngeCryption all over again,[13] but please recall that it relies on the fact that you can *decrypt* plaintext to obtain ciphertext. This reverse-ciphertext *encrypts* back to the original plaintext because block encryption and decryption operations can be safely exchanged.

Let's try to reproduce the example given by Ange in his RMLL2014 presentation, available in a translated slide deck titled "Let's play with crypto."

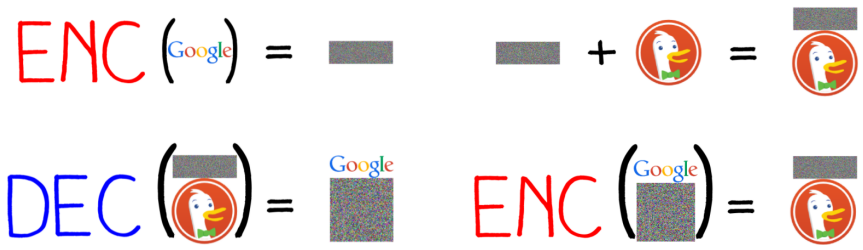


Figure 6: *"If we encrypt the final result, we get our first random data, followed by our target picture."*

This example uses PNG images, so we'll begin with two logos in PNG format and of equal width. We'll take those of Google and DuckDuckGo, with a small change: I removed subtle gradients from the original PNGs so that we get large areas of the same flat color. To better illustrate the vulnerability, we need to work on uncompressed, non-interlaced images. A tool called `advpng`[14] takes care of flattening the PNG images and minimizing the metadata by grouping all IDAT chunks into a single chunk.

```
$ advpng -z -0 google.png
$ advpng -z -0 duckduckgo.png
```

Now we can construct our AngeCryption example using Ange's *PNG-in-PNG* tool (Google for it with "corkami" and "src/angecryption/PiP/PIP.py" as search terms).

```
$ python PIP.py google.png duckduckgo.png combined.png CBC_can_fail_too
```

The resulting `combined.png` displays the Google logo and, when decrypted, displays the DuckDuckGo logo.

[12] `https://doegox.github.io/ElectronicColoringBook/`

[13] See PoC||GTFO 3:11 and its retrospectively funny quote: *"We'll use the standard AES-128 algorithm in CBC mode, which is proven to be semantically secure when used with a random IV."*

[14] `http://advancemame.sourceforge.net/`

Figure 7: `combined.png`

Ange's `PIP.py` does the opposite of what the slide proposes, just to show that it's also possible. So, to match the tool and the rest of the article you need to swap the ENC and DEC operations. It still remains pure AngeCryption.

Figure 8: *"If we decrypt the final result, we get our first random data, followed by our target picture"*

7.2 Time to fire up ElectronicColoringBook.py

```
$ python ElectronicColoringBook.py combined.png -p4 -c255
```

Figure 9: `combined.png` as seen through `ElectronicColoringBook.py`.

What can we see at this point?

We recovered the Google logo but it was not encrypted, so we aren't done yet. Still, we can see a few artifacts compared to what we obtained with ECB on a pure bitmap. It also looks like we couldn't recover the correct aspect ratio either. In fact, it did get correctly recovered, but the display included extra PNG metadata bytes, so the image got slightly skewed.

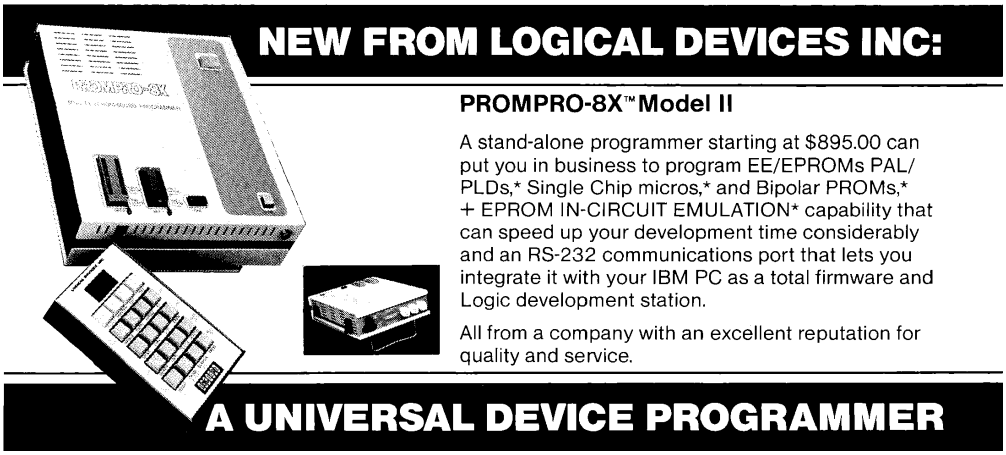
The artifacts in that image are due to the additional structure of the PNG format that is absent from a plain BMP. In a PNG image, each scan line is preceded by a byte of metadata describing which filter to apply to that line. In our case, those extra bytes are all null bytes indicating the absence of a filter. It is this one extra byte on each line that misaligns the blocks in our image recreation and skews it. It also breaks the uniform areas, so they are not that easy to paint over. Moreover, you can see a few blotches of gray here and there in the white area. That's because the image data, even when uncompressed, is still not raw pixels but a zlib stream encapsulating some DEFLATE data that has its own metadata[15] at the start of each 64 kB block.

Rather than adding additional complexity to our script to handle each of these specific quirks, it turns out that we can correct the misalignment due to the presence of metadata bytes by specifying a non-integer width:

```
$ python ElectronicColoringBook.py combined.png -p4 -o3 -c255 -x 600.345
```

Figure 10: `combined.png`, fine-tuned

[15] See `rfc1951.txt`.



The bottom of the image is completely black, which is how `ElectronicColoringBook.py` represents non-repeating blocks. That's what we expect from CBC-encrypted data, as opposed to ECB.

7.3 The downside

Now we can get to the second half of the story, the decrypted `combined.png` displaying the DuckDuckGo logo. We'll use `decrypt-PIP.py`, a helper script created by `PIP.py`, and then apply `ElectronicColoringBook.py` to the output `dec-duckduckgo.png`.

```
1 $ python decrypt-PIP.py
```

Figure 11: dec-duckduckgo.png

```
1 $ python ElectronicColoringBook.py dec-duckduckgo.png -p4 -o3 -c255 -x 600.345
```

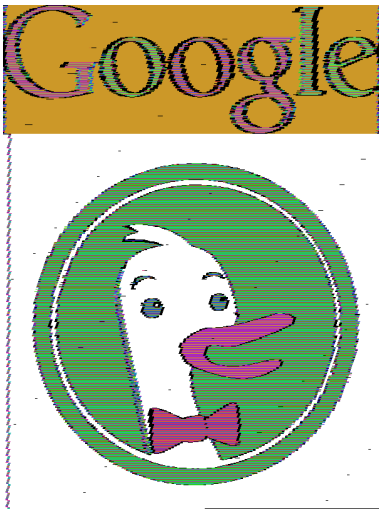


Figure 12: dec-duckduckgo.png as seen through ElectronicColoringBook.py

But what is this new devilry? Oh, no! The Google logo is still visible. Is the CBC gone all evil on us, so can't shake it off?

7.4 Why, oh why?

Recall that in the CBC mode, encryption of each block depends on all the previous blocks:

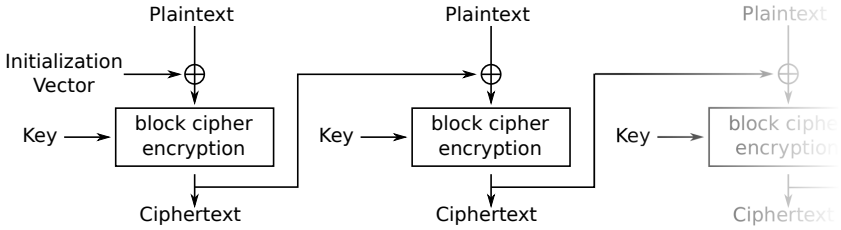


Figure 13: Cipher Block Chaining (CBC) mode encryption

But the Google part of the image is not the result of an encryption but of a decryption, remember? We must account for how these blocks feed into the CBC process.

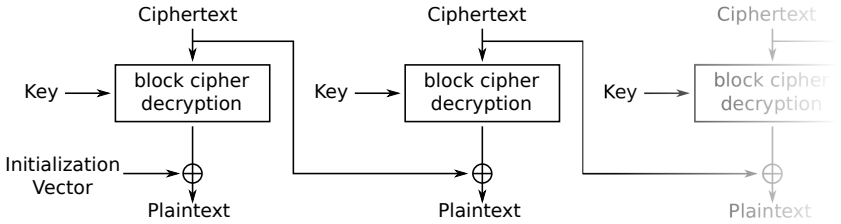


Figure 14: Cipher Block Chaining (CBC) mode decryption

Here, the ciphertext is that of the original Google image. For its image parts of constant color, we get the same ciphertext blocks over and over.

Plaintext blocks of that series will be $P_n = Dec_K(C_n) \oplus C_{n-1} \equiv Dec_K(C) \oplus C$ if all ciphertext blocks are the same.

The first plaintext block from a repetitive area depends on the previous (different) block. Thus its content is different from the following repetitive plaintext blocks.

So CBC in decryption mode is almost as bad as ECB: decrypting n repetitive blocks will give one arbitrary block followed by $n-1$ repetitive blocks (while ECB would give n repetitive blocks). That's why transitions around Google letters look slightly thicker.

In principle, we could paint over CBC when used in reverse mode as easily as we painted over ECB, but it's actually quite difficult in our example because, as you recall, the image data of PNG format is not merely raw pixels such as in the BMP or PNM formats.

In real life, decryption is usually used on data that previously went through encryption. Since the point of the CBC mode is to prevent repetitions in the ciphertext, we don't generally need to fear them, although, theoretically, they could still happen. (By a stroke of bad luck, we might get $Enc_K(C \oplus P) = C$ to occur for a given P for some combination of C and the key K.)

Let us recall another CBC fact: even if you only know the key but not the initialization vector (IV), you can still decrypt `combined.png` almost fully. Only the first block will be wrongly decrypted, which is not that hard to reconstruct; even if left corrupted, it won't prevent `ElectronicColoringBook.py` from revealing both images. Look back at Figure 14 to understand why.

So the upshot of our case study is that single-block encryption and decryption operations can still be exchanged almost safely, although the chaining mode does throw some gotchas into the process.

7.5 Exploring other chaining modes

So what about the other chaining modes that use an IV?

The CFB mode suffers of a similar problem because, in decryption mode, the block encryption depends only on the previous ciphertext. This previous ciphertext can be repeated under AngeCryption, so the resulting plaintext also repeats: $P_n = Enc_K(C_{n-1}) \oplus C_n \equiv Enc_K(C) \oplus C$.

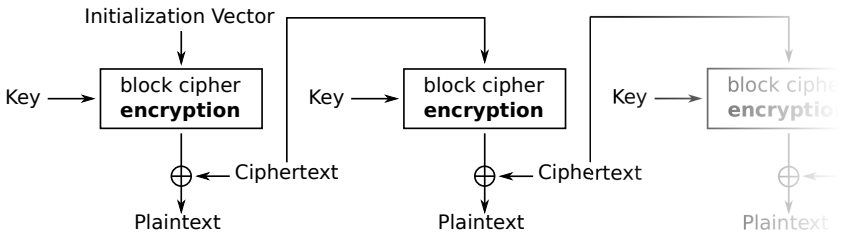


Figure 15: Cipher Feedback (CFB) mode decryption

The OFB mode makes a block cipher into a synchronous stream cipher and therefore doesn't have this issue. Encryption and decryption are just XOR with the same keystream, which only depends on the IV and the key K: $keystream_1 = Enc_K(IV)$, $keystream_n = Enc_K(keystream_{n-1})$ and $P_n = keystream_n \oplus C_n$.

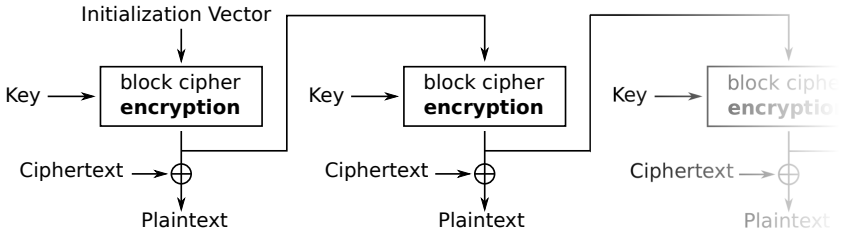


Figure 16: Output Feedback (OFB) mode decryption

Let's try this out. We modify `PIP.py` to replace `MODE_CBC` by `MODE_OFB` and inverse the order of operations to compute the IV. Indeed, if for CBC we computed $IV = Dec_K(C_1) \oplus P_1$, for OFB we must compute $IV = Dec_K(C_1 \oplus P_1)$. Then we repeat the same experiment:

```
1 $ python PIP_OFB.py google.png duckduckgo.png combined.png OFB_AngeCryption
  $ python decrypt-PIP.py
3 $ python ElectronicColoringBook.py dec-duckduckgo.png -p4 -o3 -c255 -x 600.345
```

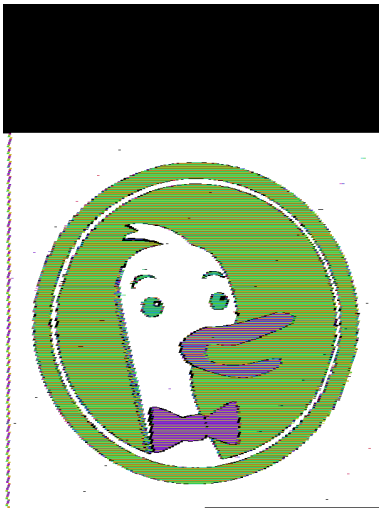


Figure 17: `dec-duckduckgo.png` (OFB version) as seen through ElectronicColoringBook.py

Finally! We get a "secure" version of AngeCryption. As a bonus, unlike CBC, if you only knew the key but not the IV, you wouldn't be able to recover anything.

Another alternative is the CTR mode, which is pretty similar to OFB: $P_n = Enc_K(counter{+}{+}) \oplus C_n$. The OFB initialization vector would play the role of the initial counter value: $counter = Dec_K(C_1 \oplus P_1)$. And, as for OFB, knowing only the key but not the initial counter value is useless.

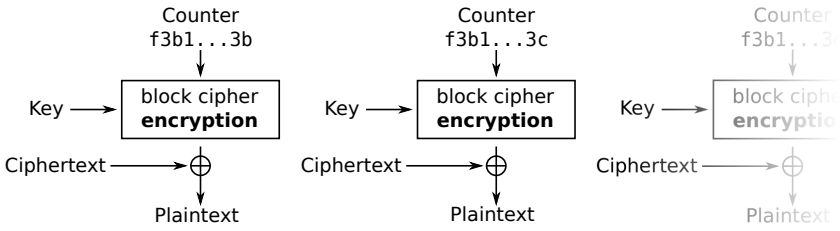


Figure 18: Counter (CTR) mode decryption

Note that both OFB and CTR have their own special limitations typical of stream ciphers: bitflipping attacks, keystream reuse, and so on. However, none of these are an issue in this unusual use case of ours.

The PCBC (Propagating CBC) mode would work as well, because each block decryption depends on the previous ciphertext *and* the previous plaintext: $P_n = Dec_K(C_n) \oplus C_{n-1} \oplus P_{n-1}$. It's not supported in PyCrypto, however, and is not very common.

7.6 Some more PoC

Before we wrap up, I'd like to circle back to a variation of AngeCryption suggested by Gynvael Coldwind, and so rightfully called GynCryption. GynCryption doesn't rely on IV forgery, but rather tries to find a key that transforms the plaintext into the ciphertext we want. For a PNG, it requires control over the first 16 bytes, but this cannot reasonably be done for an entire block. On the other hand, controlling the first 6 bytes of a JPG is enough to be able to insert a small comment section. GynCryption was originally based on ECB, but nothing prevents us from replacing ECB by CBC, CFB, OFB, or by CTR with a null IV or a reset counter respectively—as we've shown above, those are only slightly better than ECB. In this issue's

polyglot archive you can find two proofs of concept, `gyncryption_ofb.pdf` and `gyncryption_cfb.pdf` that you can decrypt into a JPG with a null IV/counter and the same key "@doegox_5f32c6e5".

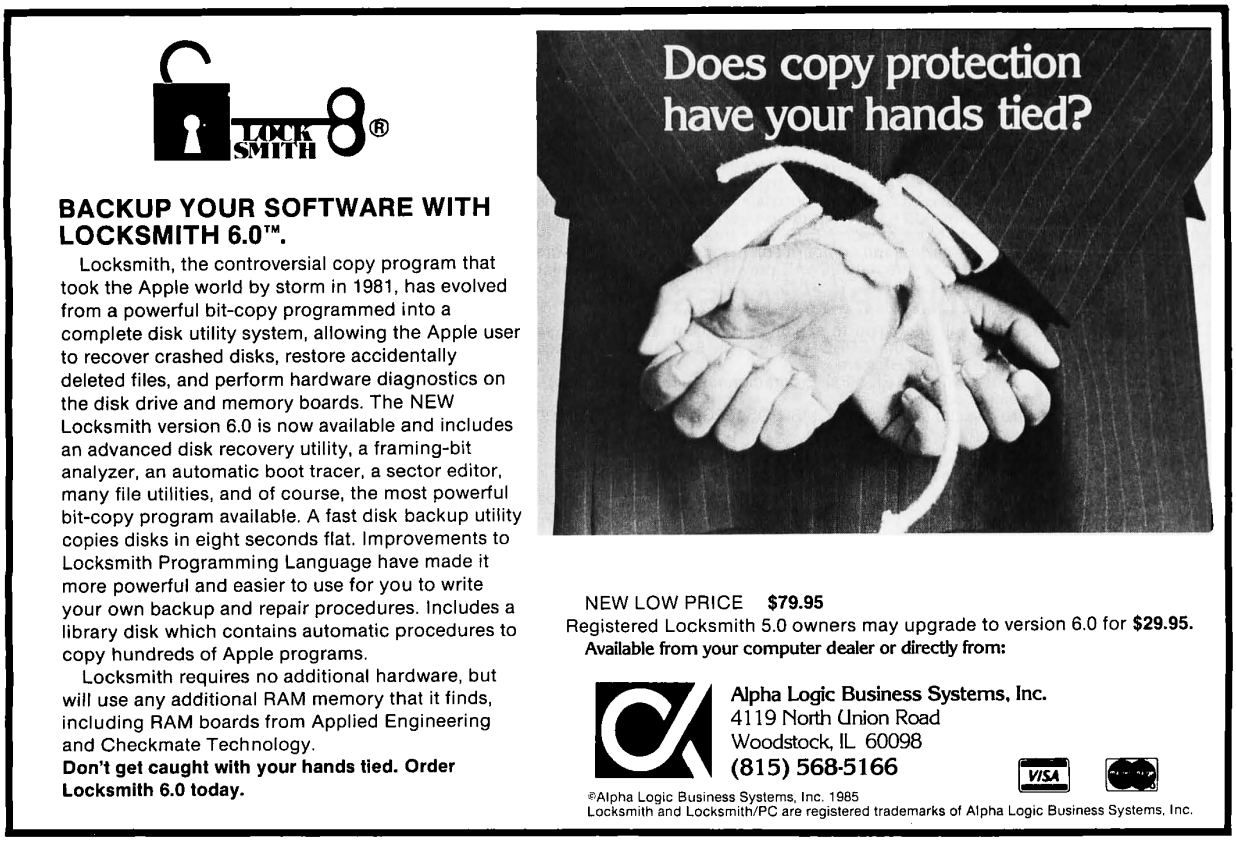
With OFB and CTR, once you have found such a key, you may be tempted to reuse it with any other (small) PDF and JPG, and it will work because they are similar to stream ciphers: a change in a plaintext block affects only the corresponding bits of the ciphertext, not the entire block. But remember that stream ciphers are only secure if you don't reuse the keystream—so don't reuse your key for the same mode, find another one! Otherwise a simple XOR of both files will result into the XOR of the plaintext data (and padding), and the keystream will be entirely removed.

7.7 Conclusions

Of course, since AngeCryption and GynCryption are far more likely to be used as crypto curios rather than as serious tools for serious situations, their security is not that crucial. Still, it is good to understand the risks associated with non-standard uses of block cipher modes—this understanding should serve as an antidote to their blind reuse in inappropriate contexts.

7.8 Acknowledgments

Special thanks go to Ange for his most neighborly help; without him this article would have never been possible!



8 Introduction to Delayering and Reversing PCBs

by Joe Grand

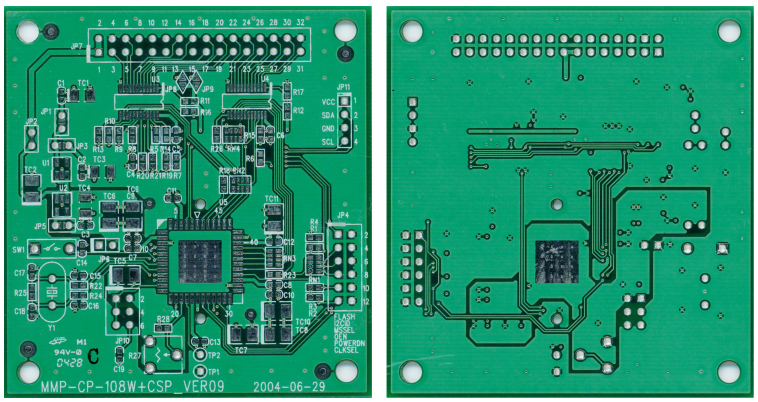


Figure 19: Our example PCB in its unmodified state. If only it knew the suffering that it was about to endure.

Figure 20: Sandpaper at work. You can see the copper of inner layer 2 starting to peek out from underneath the top substrate.

Printed Circuit Boards (PCBs) form the physical carrier for and provide electrical pathways between electronic components. They are created with layers of thin copper (conductive) foil laminated to insulating (non-conductive) layers. By accessing and imaging each individual copper layer of a PCB, it is possible to recreate the PCB layout. If the component types (and values, ideally) are known, you'll also be able to derive the schematic (a simplified, visual representation of the device's electronic design) or a desired portion thereof.

"Why bother?" you might ask. Maybe you want to understand how a particular product works, locate specific connections on the board (like JTAG or UART), clone the design, or figure out where you can modify

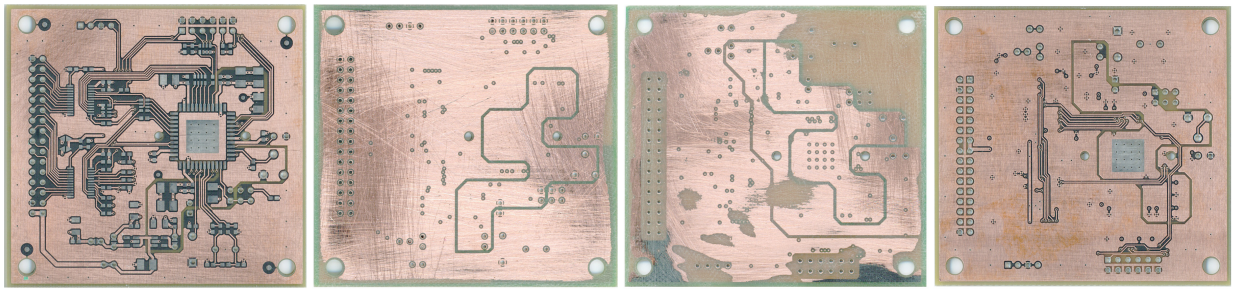


Figure 21: The four exposed layers of our example PCB.

it to inject malicious functionality. The techniques provided in this article might not be groundbreaking to those skilled in the hardware arts, but will serve as a resource for folks interested in meandering down the path of PCB reverse engineering.

8.1 Delayering

The first phase of the process is to obtain an image of each layer of the target circuit board. There are a variety of possible techniques, including low-tech, off-the-shelf solutions and those requiring expensive equipment and skilled operators. Some methods are destructive, meaning you'll never see your PCB again when you're done, and some are non-destructive, meaning the PCB will remain intact and unharmed. For now, we're going to focus on manual abrasion using sandpaper, which will destroy your board layer-by-layer, but is also the simplest and most accessible.

The top and bottom of a PCB are usually coated in solder mask, a non-conductive layer that protects the PCB from dust and oxidation and provides access to copper areas on the board that are intended to be exposed. You'll want to remove the solder mask so you have unobstructed access to the underlying copper. To do so, attach the PCB to your work surface with a clamp or double-sided tape. Then, use 60 to 220 grit sandpaper in even strokes at light pressure across the entire board. Optionally, you can put spare PCBs of the same height as the target on either side to help maintain planar motion and even sanding pressure. Holding the sandpaper by hand will give you the best control. If you're prone to repetitive stress injuries, a tool such as a Norton Sheet Sander may serve you well.

Once you've exposed the copper, it's time to capture an image of the layer. If you have access to a flatbed scanner, use that. Otherwise, a point-and-shoot camera will work. (When using a camera instead of a scanner, be aware that you may need to rotate and lens-correct the resulting image to make it appear as planar and true-to-form as possible.)

To access the inner layers, the process is similar to removing the solder mask. For this step, you'll need harder pressure and more elbow grease to deal with removing the layer of insulating substrate, a fiberglass/epoxy weave.

Figure 19 shows the top and bottom of our example PCB in its unmodified state. This board is 4-layer, 62 mil thick, with trace widths ranging from 12 to 48 mil. Figure 20 shows PCB delayering in action. After you've successfully accessed and imaged each layer of the PCB, you should end up with a sequence similar to Figure 21.

8.2 Image processing

With your PCB layer images in hand, the next phase is to use an image processing/manipulation tool of your choice to adjust the images, create a stack-up of the layers, and configure the opacity of each so that you can see all copper features at once: footprints, traces, vias, and fills. Suitable programs include Adobe Photoshop, GIMP, and Paint.NET.

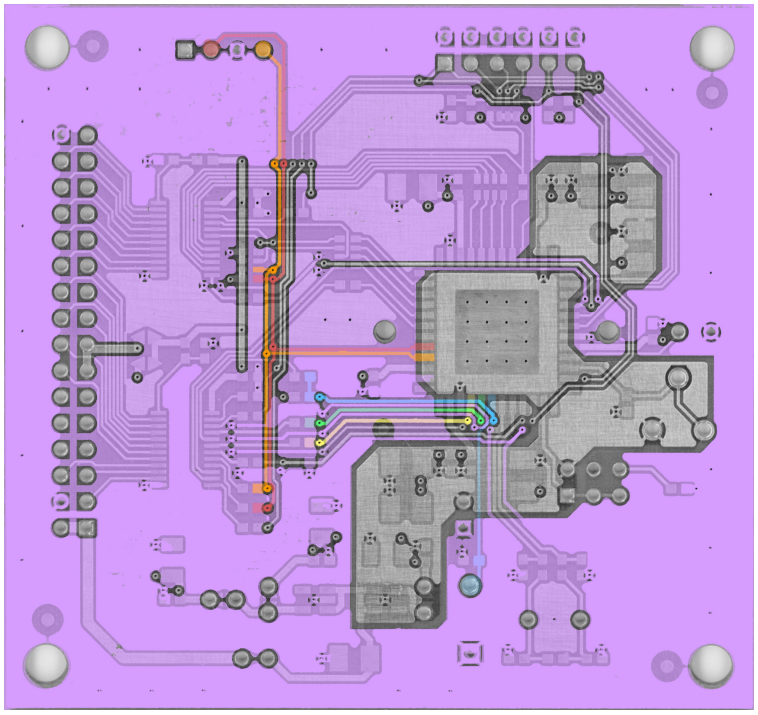


Figure 22: Layer stack-up of our example PCB. Layer opacity was adjusted to see through the board and arbitrary traces were colored using a flood fill.

The image processing tasks are as follows:

1. Rotate and mirror the images so they all have the same orientation. For reverse engineering purposes, you'll want a view of each layer as if you're looking down at it through the top of the board. This means that the bottom half of your image set will need to be flipped/mirrored vertically. Choose a feature of the PCB that exists on all layers, such as a mounting hole, test point, via, or through-hole footprint, and make sure that it's in the same position on the board in each of the images.

2. Adjust the images so the copper features on each layer are easily distinguishable from the underlying substrate. The exact adjustments you need to perform will vary depending on the quality of your deconstruction process and resulting images. At a minimum, you'll want to remove unnecessary features, adjust brightness/contrast, and desaturate to shades of grey or convert to black and white.

3. Merge the images into a single file, to create a stack-up of the layers, by placing each one on its own layer within your image processing tool. Set the opacity of each layer to 50% as a starting point, while leaving the bottom layer at 100%. This will let you see through the layers enough to identify the PCB features on each. Make sure that drill holes and other through-hole features match across the entire board surface. You may need to make small rotational or minor scaling adjustments to exactly align the layers.

8.3 Reverse engineering

The goal of this phase is to determine how components are physically interconnected on the board by visually following the copper, assisted by your image processing tool. If you want to make use of the information you glean from these efforts, you may want to have a modicum of electronics knowledge.

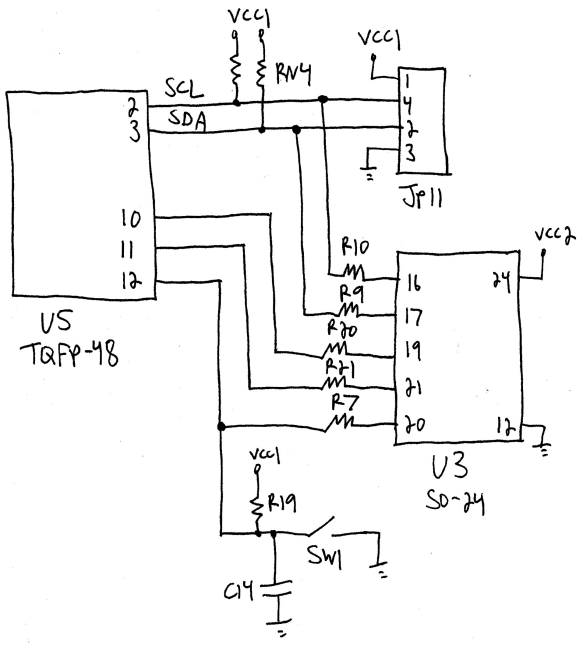


Figure 23: Schematic based on the colored signals of Figure 22. This kind of visual representation is much easier than a collection of PCB layer images.

To begin, identify the major component footprints on the board and pick a starting location on one of them. If component part numbers are known, obtain their associated data sheets for details about the component, its pinout, and pin functionality. Then, prepare yourself for a lot of repetition.

With your image processing tool, enable and disable the layers as needed while using a flood fill to set the color of the desired trace and anything it's in contact with. You'll find yourself hopping between the various layers and zooming in and out as you follow the trace around and through the board. Draw a schematic as you go, adding to it each time you finish coloring a route. Keep in mind that the PCB silkscreen often contains reference designators, part numbers, component values, and other useful information that you can incorporate into your schematic. A board's physical characteristics and actual layout can also be very important aspects of the design, but we'll ignore them for now. Repeat these steps until every trace is accounted for.

Figure 22 shows a working view of my PCB layer stack-up with a few arbitrarily selected connections traced and colored. Figure 23 shows the resulting schematic.

If you want to see a true master of signal tracing, watch any of Chris Tarnovsky's chip hacking presentations from Black Hat or DEFCON. For a different approach to PCB reverse engineering, take a look at Throbscottle's Instructable.

8.4 Next steps

As you might now be aware, the current state of PCB reverse engineering is a manual, time consuming, and often difficult task. The obvious progression of this work is to automate as much of the process as possible. I've started developing a toolkit to assist in recreating a complete schematic based on a collection of PCB layer images. Imagine Karsten Nohl, Starbug, and Martin Schobert's degate or Adam Laurie's rompar, but for circuit boards. I, for one, am excited about the possibilities.

9 Davinci Seal: Self-decrypting Executables

by Ryan O'Neill,
who also publishes as Elfmaster

In the pursuit of creativity and fun, I recently had the idea of creating self-protecting files. That is to say, any type of data that you want protected from analysis, with the ability to decrypt its own content when provided the correct key. The use cases for such a capability are debatable, but the idea is nevertheless fun, and only took an afternoon to implement. The goal was to create a program that can transform any file into an ELF executable whose sole purpose is protecting the file data embedded within its own body. I call these Davinci Seals.

9.1 Protection

The output executable should be able to protect the embedded data from static analysis and resist runtime analysis and `ptrace`-based debugging. An attacker should not be able to extract the content by setting breakpoints and reading the decrypted content from memory; thus, detection of such attacks should be in place. The executable should also be resistant to attackers modifying code or replacing anti-debug code with NOP instructions; this can be mostly prevented by using code watermarking. There are forms of dynamic analysis such as dynamic instrumentation with Pin, or using an IDA Emulator plugin, which Davinci does not mitigate, but we briefly discuss viable methods for protection against them.

9.2 Example of creating a Davinci seal

```
$ cat msg.txt

 ____________________
|The spice must flow |
 --------------------

$ ./davinci msg.txt msg.dvs p4ssw0rd -r
[+] The user who executes msg.dvs must supply password: p4ssw0rd
[+] Encoding payload data
[+] Encoding payload struct
[+] Building msg program
[+] (Optional) utils/stripx exists, so using it to strip section headers off of DRM archive
Successfully created msg.dvs

** NOTE: msg.txt was transformed into an ELF executable (A davinci seal) named msg.dvs

$ readelf -l msg.dvs

Elf file type is EXEC (Executable file)
Entry point 0x400492
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000918 0x0000000000000918  R E    200000
  LOAD           0x0000000000001000 0x0000000000601000 0x0000000000601000
                 0x0000000000800324 0x0000000000800338  RW     200000
  NOTE           0x0000000000000158 0x0000000000400158 0x0000000000400158
                 0x0000000000000024 0x0000000000000024  R      4
  GNU_EH_FRAME   0x00000000000006c0 0x00000000004006c0 0x00000000004006c0
                 0x000000000000007c 0x000000000000007c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10

```

```
$ ./msg.dvs
This message requires that you supply a key to decrypt

$ ./msg.dvs p4ssw0rd
 ____________________
|The spice must flow |
 --------------------
```

Voila! Our msg.txt file was transformed into msg.dvs, an ELF executable which lives and breathes only to protect the data within it, and reveal that data when supplied the encryption key.

9.3 Implementation

9.3.1 ELF stub and payload packaging

The goal here is to transform a file containing arbitrary data into an ELF executable whose sole purpose is to protect the data. The executable should decrypt and write the data to stdout if the proper password/key is supplied.

Our project consists of two parts. The first is the Protector, which creates the output program from the second, which we'll call the Stub.

The protector program takes an input file and generates a stub executable that contains the encrypted input file within it, as well as metadata describing the size and location of the data. The stub executable that it generates is written mostly in C, then compiled into bytecode and stored within the protector executable. To fully understand the protector, we must first understand the stub.

The basic principle of the stub is that it contains an encrypted file. This encrypted data must be stored somewhere with information about it. The best way to implement this is to append the data to the data segment of the stub executable, or even within the text segment using a reverse extension method. Both methods are common in virus infection and executable packers, but for the sake of POC and simplicity we will pre-allocate a fixed size within the initialized data section of the stub executable.

```
/* From davinci.h */
#define KEY_BUF_LEN 256
#define MAX_PAYLOAD_SIZE ((1024 * 1024) * 8)

typedef struct payload_meta {
        uint64_t payload_len;      /* Length of the encrypted file data */
        uint32_t keylen;           /* Length of the key used to encrypt */
        uint8_t key[KEY_BUF_LEN];   /* The key used to encrypt/decrypt */
        uint8_t data[MAX_PAYLOAD_SIZE];  /* The file data itself */
} payload_meta_t;

/* From stub.c */
payload_meta_t payload __attribute__((section(".data"))) = {0x0};
```

Since the data and metadata will be stored in the structure above, the protector can resolve the `payload` symbol to find where it needs to store the file data and key data within the stub.

```
-- Illustration of the work flow:

[input file (msg.txt)] /* The input file can be anything */
    |
    v
[protector] /* This program transforms msg.txt into msg.elf */
    |
    v
[output file(msg.elf)] /* The output is a compiled stub.c, instrumented with the encrypted
    input file, and metadata */
```

9.3.2 Anti-analysis protection

The goal is to transform an input file into an output executable that protects it. The input file is encrypted/obfuscated and embedded within an ELF executable that serves as a defensive shell. This defensive shell will decrypt the data if supplied the correct key, and write it to standard output. If you choose, you may tell the protector to store an obfuscated copy of the key within the binary so that it decrypts itself without a supplied password. This offers no real protection, of course, but may still have some application.

Our defensive shell, being an executable and all, is inherently vulnerable to reverse engineering, static analysis, and debugging (dynamic analysis) attacks. It would behoove the defending binary to have some protection against some of these attacks. We have three protections against static analysis:

1.) The body of the input file is encrypted within the output executable, though just with weak XOR for this proof of concept. The `payload_meta_t` structure is also encrypted, on top of the `payload.data` buffer. If Davinci is to become more than just a proof of concept, a real cipher must be used.

2.) The section header table is stripped from the ELF executable. String tables are zeroed out, and the symbol table is discarded.

This by itself makes the output executable far more difficult to navigate with a disassembler, since there is no information provided about symbols or specific sections. The program headers are suitable for loading and running a program, but without section headers, the program is more difficult to analyze, even for IDA Pro.

Stripping the ELF section headers effectively disables any tools that rely on section headers. It is an old and simple technique used by many neighbors.

```
--Prevents objdump disassembly
$ objdump -D msg.dvs
msg.dvs:     file format elf64-x86-64
$

--Prevents symbol lookups
$ readelf -s msg.dvs
$
```

3.) The output executable is further protected with UPX, the Ultimate Packer for eXecutables. This also takes care of shrinking the executable from the wasteful fixed-size of our buffer.

This feature is primarily for shrinking the output executable, because the stub is by default fixed at a large size. Initializing an 8 MB buffer in the `.data` section leaves room for files up to 8 MB. As mentioned earlier, another method, such as appending to the data segment, would be a better long-term design decision and would result in the executable growing in proportion to the input file size. For the sake of POC, we used the method of initializing fixed space in the `.data` section, which allows us to focus more on the principles and less on the implementation.

9.3.3 Anti-debugging tricks

Most debuggers, such as GDB, rely on the `ptrace` system call. If `ptrace`-based debugging can be prevented, we eliminate the most common types of dynamic analysis tools. `strace`, `gdb`, dumping `/proc/$pid/mem`, and other tricks will all break.

Anti-Ptrace Protection A process is only allowed to have one tracer. This means that we can simply use `ptrace` within our stub executable, so that it traces itself, preventing any other debuggers/tracers from attaching. If a debugger is attached before our stub calls `ptrace()`, then our call to `ptrace()` will return `-1` and we can abort the process.

The `enable_anti_debug()` function will prevent `gdb` and `strace` from analyzing our ELF executable.

```
/*
 * Notice that we use our own wrapper for the ptrace syscall.
 * This is good practice to prevent LD_PRELOAD bypasses --
 * even though our stub is compiled -nostdlib (in which case
 * an LD_PRELOAD bypass would not work anyway).
 */

static long _ptrace(long request, long pid, void *addr, void *data) {
        long ret;

        __asm__ volatile(
                        "mov %0, %%rdi\n"
                        "mov %1, %%rsi\n"
                        "mov %2, %%rdx\n"
                        "mov %3, %%r10\n"
                        "mov $101, %%rax\n"
                        "syscall" : : "g"(request), "g"(pid), "g"(addr), "g"(data));
        asm("mov %%rax, %0" : "=r"(ret));

        return ret;
}

void bail_out(void) {
        _write(1, "The gates of heaven remain closed\n", 34);
        _kill(_getpid(), SIGKILL);
        __exit(-1);
}

void enable_anti_debug(void) {
        if (_ptrace(PTRACE_TRACEME, 0, NULL, NULL) < 0)
                bail_out(); // if a debugger is already attached we bail out
        // a marker showing that an attacker didn't just jump over enable_anti_debug()
        data_watermark++;
}
```

Now what happens when we try to debug `msg.dvs` with `gdb`?

```
$ gdb -q msg.dvs
Reading symbols from msg.dvs...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/ryan/dev/davinci/msg.dvs
The gates of heaven remain closed
Program terminated with signal SIGKILL, Killed.
The program no longer exists.
(gdb)
```

If an attacker wants to bypass the anti-`ptrace` code, there are several techniques that are commonly used.

1. `LD_PRELOAD` can be used to preload a library. This loads the specified library before any others, and any of its symbols will take precedence over subsequently loaded libraries. Attackers have used this to preload a custom shared library with a dummy `ptrace` that simply returns success and does nothing. In our stub executable we do not use dynamic linking, and therefore no shared libraries can even be loaded. We also use a syscall wrapper for `ptrace`, so that even if our stub did use dynamic linking, our calls to `ptrace` would not go through the PLT/GOT and therefore could not be hijacked with another shared library call. Always use syscall wrappers in binary hardening code, and stay away from glibc.

2. An attacker could modify the stub's binary code so that the `enable_anti_debug()` code is never called, or simply jumped over. An attacker could also overwrite the code in `enable_anti_debug()` so that it doesn't actually do anything to prevent debugging. We use a simple form of code watermarking to try to prevent this, which we will discuss in Section 9.3.4.

/proc/<pid>/mem Dump Protection It is a common practice for reverse engineers/attackers to dump a hardened binary from memory. This can be done by attaching to the process and reading `/proc/<pid>/mem`. If the process is already stopped, then attaching to the process isn't necessary, and a simple `read()` suffices. Fortunately, Linux has a neat syscall called `prctl()`, which allows us to change the characteristics of our running programs, but must be issued by the program itself.

```
        int prctl(int option, unsigned long arg2, unsigned long arg3,
2               unsigned long arg4, unsigned long arg5);

4   OPTION:  PR_SET_DUMPABLE (since Linux 2.3.20)
             Setting arg2 to 0
6       prevents process from dumping a CORE file,
        prevents process from being attached to with ptrace, and
8       prevents process from being dumped from /proc/<pid>/mem.
```

The `PR_SET_DUMPABLE` option applies several very neat and useful anti-debugging features. We use this to add even more resistance to `ptrace`, while also preventing core dumps and memory dumps of our process.

```
/*
2 * Always implement a syscall wrapper when using syscalls for anti-debugging
 */
4 int _prctl(long option, unsigned long arg2, unsigned long arg3,
            unsigned long arg4, unsigned long arg5) {
6       long ret;

8       __asm__ volatile(
                        "mov %0, %%rdi\n"
10                      "mov %1, %%rsi\n"
                        "mov %2, %%rdx\n"
12                      "mov %3, %%r10\n"
                        "mov $157, %%rax\n"
14                      "syscall\n" :: "g"(option), "g"(arg2), "g"(arg3),
                                       "g"(arg4), "g"(arg5));
16      asm("mov %%rax, %0" : "=r"(ret));
        return (int)ret;
18 }

20 /*
 * Simply call _prctl(PR_SET_DUMPABLE, 0, 0, 0, 0) from your code.
22 * (Ideally from a glibc constructor)
 */
24
void anti_debug_dump(void) __attribute__ ((constructor));
26
void anti_debug_dump(void) {
28   _prctl(PR_SET_DUMPABLE, 0, 0, 0, 0);
}
```

SIGTRAP Detection When breaking binaries, the attacker generally will set breakpoints in specific areas of the code. With `SIGTRAP` detection we can detect breakpoints, as they generate a `SIGTRAP` signal. Upon detection we can do whatever we like, ideally bail out and kill the program.

This can be done by creating a signal handler for `SIGTRAP`. If our signal handler catches the signal, then it means there is no debugger attached. Since our stub is not linked to `libc` in any way, we must use our own syscall wrapper for `sigaction`. Thanks to Jpanic for pointing out important caveats that must be considered when doing this.

```
#define SA_RESTORER 0x04000000

/* struct sigaction act.sa_restorer must point to a handler
 * that performs an rt_sigreturn(0)-- normally this is done
 * by glibc.
 */
int _sigreturn(unsigned long unused) {
        unsigned long ret;
        __asm__ volatile(
                        "mov %0, %%rdi\n"
                        "mov $15, %%rax\n"
                        "syscall" : : "g"(unused));
        __asm__("mov %%rax, %0" : "=r"(ret));
        return (int)ret;
}

/* We increment trap_count if we caught the signal */
int trap_count = 0;

void sigcatch(int sig) {
        trap_count++;
}

/* This function sets up a signal handler for SIGTRAP
 * if a debugger caught it.
 */

void install_trap_handler(void) {
        struct sigaction act, oldact;
        act.sa_handler = sigcatch;
        act.sa_flags = SA_RESTORER;
        act.sa_restorer = restore;
        sigemptyset(&act.sa_mask);
        sigaddset(&act.sa_mask, SIGTRAP);
        // must pass sizeof(long) or kernel returns -EINVAL
        _sigaction (SIGTRAP, &act, NULL, sizeof(long));

}

void detect_debugger(void) {
  __asm__ ("int3\n"
     "nop");
  if (trap_count == 0)
    bail_out(); // debugger caught the trap, bail out!
  trap_count = 0;
}
```

There exist other anti-debugging techniques not used in this example. `/proc/self/status` can check if a `ptrace` attachment exists. Junk or misaligned assembly code could be used to obfuscate the application against a disassembler while keeping it functionally equivalent.

Advanced reverse engineers will go well beyond the use of `ptrace()`-based debuggers when attempting dynamic analysis. Such engineers might use the Pin instrumentation framework, an emulator, or ERESI's `e2dbg`.

Detection of Pin hooking can be done by checking `/proc/self/maps` to see whether the mapping called `[vvar]` exists after `[vdso]`. This happens when `vdso` has been partially remapped by Pin.

Emulation detection can also be performed by `rtdsc` timestamp checking.

9.3.4 Code and data watermarking

To enforce our anti-debugging code so that it is not easily circumvented, we have some simple code and data watermarking in-place. As mentioned earlier, if someone were to modify the `enable_anti_debug()` code, or simply jump over it, it would be rendered useless. We must therefore be prepared to detect when this happens and act accordingly by exiting or killing the program before it is successfully cracked.

Data Watermarking For the data watermarking, we have a static initialized variable that is set to 0 and only incremented after the `enable_anti_debug()` function successfully completes. Later on, we check the value of this variable. If it has not been incremented, then we can assume that an attacker either jumped over the anti-debug code or NOP'd it out.

```
void denied(void) {
        bail_out();
}

void accepted(void) {
        __asm__ __volatile__("nop\n");
}

_start() {
  uint64_t a[2], x;
        void (*f)();
  int ret;

  ... <code> ...

  a[0] = (uint64_t)&denied;    // a[0] points to denied() address
        a[1] = (uint64_t)&accepted;    // a[1] points to accepted() address
        x = a[!(!(data_watermark))];  // convert data_watermark to a boolean, 0 or 1
        f = (void *)x;           // assign function pointer to either accepted() or denied()
        f();         // call accepted() or denied()

  ... <code> ...
}
```

As we can see by the code snippet above, if `data_watermark` was not incremented it will still be 0, so we can assume that an attacker jumped over the `enable_anti_debug()` code. So `denied()` would be called, which calls `bail_out()` to kill the process. Otherwise, `accepted()` will be called, which does nothing, and our binary goes on running untampered.

Code Watermarking For the code watermarking, we want to validate that the `enable_anti_debug()` function has not been modified in any way. We do this by simply fingerprinting it.

```
/* From davinci.h */
typedef struct code_watermark {
        uint32_t code_size;
        uint8_t code_signature[CODE_CHUNK_SIZE];
} code_watermark_t;


/* From davinci.c
 * NOTE: 'uint8_t *mem is a mapping of the stub executable'
 * This code will create the fingerprint of enable_anti_debug() and store
 * it within the stub executable
 */
 ... <code> ...

  symval = resolve_symbol("enable_anti_debug", mem);
```

```
    symsize = resolve_symbol_size("enable_anti_debug", mem);
    offset = textOffset + (symval - textVaddr);
    code_watermark = (code_watermark_t *)alloca(sizeof(code_watermark_t));
    memcpy((uint8_t *)code_watermark->code_signature, (uint8_t *)&mem[offset], symsize);
    code_watermark->code_size = symsize;
    symval = resolve_symbol("code_watermark", mem);
    symsize = resolve_symbol_size("code_watermark", mem);
    offset = dataOffset + (symval - dataVaddr);
    memcpy((void *)&mem[offset], (void *)code_watermark, sizeof(code_watermark_t));
  ... <code> ...

/* From stub.c
 * We memcmp the enable_anti_debug() function with code_watermark.code_signature.
 * If there are any discrepancies, we call denied(), which bails out and prints the message
 * "The gates of heaven remain closed"
 */
  ... <code> ...

          a[0] = (uint64_t)&accepted;
          a[1] = (uint64_t)&denied;
          ret = _memcmp((uint8_t *)code_watermark.code_signature, (uint8_t *)enable_anti_debug
      , code_watermark.code_size);
          x = a[!(!(ret))];
          f = (void *)x;
          f();
  ... <code> ...
```

9.4 Getting Davinci

The Davinci source code tarball is stored in a davinci seal itself :)

```
chmod +x davinci.tgz.dvs
./davinci.tgz.dvs d4v1nc1 > davinci.tgz
tar zxvf davinci.tgz
```

"For the last time, Brian," said Barbie, "`$4C` is absolute jump and `$6C` is indirect jump. It's like this: `$4C` is me telling you that you're an idiot; `$6C` is me pointing you to a piece of paper that says, 'You're an idiot.' And what the hell are you smiling at, Steven? You've got code here that overwrites the ROM monitor. Unless your last name is Wozniak, STFO out of `$F000` block."

10 Observable Metrics

fiction by Don A. Bailey
from a concept developed with Tamara L. Rhoads and Jaime Cochran
for J. O., A. S., and S. G. S.

Gold from the late November sun washed an otherwise porcelain hallway, as the door to the Vice President of Engineering's office opened. Stepping into this naturally lit office, out of the antiseptic hall, was a reminder of the perks of a hard earned career rolling out next generation Internet of Things technology.

He stood in the center of the room, smiling an inviting smile, while rays of light seemed to flow from the tips of his outstretched arm. He beckoned the engineer to sit. His raised standing-desk was elegantly constructed in a nod to George Nakashima's signature style. Its varnished surface accentuated the tree rings underneath through a translucent hue. The sides of the desktop were kept natural, almost raw. Some of the tree's original bark still proudly masked the unfinished growth hidden below.

To the left of the desk stood a large American flag, whose pole rose to centimeters below the ceiling. Its fabric moved slightly to the rhythm of the office air, which was coaxed around the room by an unseen and unheard ventilation system. The flag seemed to be placed purposefully on this side of the room, at the edge of the wall of windows that faced south San Diego bay, where a battleship sat in the distance. Tiny figures in white were noticeably scurrying around the flat, grey deck, in what seemed to be a concerted effort to clean the behemoth.

She smiled as she sat down. The chair's leather creaked under her slim figure, as her body adjusted to the boxy and industrial shape of the Le Corbusier-style object.

"Thank you for joining me for a quick discussion! I know how busy you are with the final security audit of the new 768 product line," the VP smiled, one arm relaxing on the edge of his standing desk, the other casually half-hanging from his designer jeans pocket.

Before the engineer could comment on the progress of the current audit, the VP questioned her. "How do you feel about the security of the new low-power mesh module? It's pretty robust for being able to fit on the new product line, isn't it?"

She paused before answering, expecting the silence was only a dramatic pause before he continued on with the wireless module he designed himself. Even though it was yet another low-power wireless module, it was designed using transparent silicon, and is able to integrate seamlessly into their new eye-contact heads-up-display line. What was even more impressive was the fact that he designed the module to use a new energy harvesting method that relied on the human eye's restlessness, its constant micro-movements, its tremors, to generate the small bursts of power required to drive the transceiver. It was all very impressive, and very heavily patented.

A new mesh protocol had to be designed, in order for the extremely low-power transceiver to work effectively. The protocol was heavily vetted from a security perspective prior to filing the patents. Even the company lawyers had to get involved by assisting with the high level threat modeling process, especially since weaknesses in this protocol could allow attackers to hijack a victim's imaging data, let alone their vital statistics. She knew this was all done prior to her arrival at the organization, just over a year and a half ago. Obviously, he was looking for a little praise.

"The security architecture is excellent. I don't think there is anywhere that I could add value to the project," she smiled. She wasn't going to drip saccharine words from her mouth. The truth was good enough as a compliment.

"Excellent," he regurgitated with his chin in the air. "Excellent."

He continued, "But you did find the security flaw in our cryptographic key storage chip. That was excellent work. We needed someone with your expertise to help find out how we'd end up hacked."

"Yeah, but to be honest, I'm just following the recommendations of other researchers that have done prior work in this area. Tarnovsky, Nohl, and even Nedospasov have given presentations on strong attacks in this area. It's really just a matter of bypassing the chip's security mesh with existing technology that was designed for complex hardware analysis. Not to mention, you can use similar attacks against Physically Unclonable Functions..." She realized his eyes had glazed over, and looked sheepishly at her feet, which were tapping nervously against the cold, cylindrical legs of the Le Corbusier replica.

Her moment of emotional self-doubt aroused him from his entranced state. He scoffed "Yeah, I'm sure everybody can hack hardware like that, these days." Realizing his eagerness to exploit her humility was

obvious, he regained his composure and ran his hand through one side of his hair and smiled. "You did excellent work, there. I was impressed."

She couldn't help herself from narrowing her eyes. She thought this was just a check-in on the status of the mesh security architecture. But, now, she knew he needed something else. What was bothering her was that this typically direct, type-A male was seemingly taking the round-about in arriving at the real topic.

"So, how can I help you? I'm sure you didn't ask me to your office to discuss research. What's up?" she offered, her right foot still tapping against the chair leg.

"I just got word this morning, entities overseas have recreated your work. I guess I should say they've independently discovered the security flaw." The VP leaned forward, putting the weight of his abs on the standing desk, his thick chest pointed directly toward her. His knuckles whitened, his hands gripped the sides of the desk, as he leaned even further over the desk like a reverend poised at a pulpit, ready to spit out a sermon.

"Those sons of bitches not only have broken this device, but they've broken every one of our products! How are they doing it?!" His oddly calm voice was chilling in contrast to the hulking position his body took behind the pulpit-like desk. "I don't even care how anymore. I really don't."

"The clones they've been building of our products have been flooding the foreign markets for several years." he continued. "Our quarterly earnings are hundreds of millions of dollars short on revenue because of these cheap knock-off items. I don't even want to look some of our investors in the eye because we can't keep these people out of our market."

The man moved out from behind his pulpit and stood in the center of the room, with the rays of the sun behind him. As he leaned in, the angle of the sunlight caused his face to become engulfed in shadow. He spoke so softly now that she had to lean in, making his aggressive posture even more uncomfortable. "It's weak. It's pathetic. I want it stopped".

The young engineer was barely able to contain her sigh of relief. "For a second there, I thought you were going to fire me," she half-joked.

He raised his body into a polite, standing posture and laughed whole-heartedly, "No, no! My apologies! You're imperative to this organization, now! I know how hard you've worked, you should have absolutely no concerns about your performance. The fact is, I need your advice."

She put her hand to her chest. Her foot moved away from the metal chair leg, where it had already began to tarnish the gleaming silver. Her eyes widened as she humbly replied "Thank you, I really appreciate that. Sometimes it's a bit hard, you know, still being 'the new guy' even after a year and a half of effort."

He picked up a white mug half filled with black tea and emblazoned with the company logo from his desk, and took a sip. His eyes affixed somewhere past her, as if he were caught up in another distant conversation she couldn't hear. "Don't be ridiculous, he replied. You're excellent..."

"Unfortunately, sir, I have to tell you what you already know. Unbreakable security is simply impossible. It's just never going to happen. We build effective models so that arbitrary people can't affect the products of millions of people. But, anyone with adequate funding can attack and learn about any given system. No proprietary technology will stop someone from cloning or reproducing someone else's work. Security just can't achieve a goal like that."

Her eyes were light, but serious. She understood his frustration, and even sympathized with him. He had worked so relentlessly for so many years building new and innovative things that leeches just flippantly dressed in cheap 3D plastics and silk screened logos. They had no respect for the artist behind the engineering degree. They only saw a Giovanni Bellini that was finally forgeable, because no one decaps an integrated circuit to see if the eye-contact wearable device was sculpted by the real artist, or by a second-rate hack. They only want to flaunt the logo most recently approved by the hip kids, and the ability to Tweet photos of Bae with a champagne glass balanced on her ass.

"Yeah." He sighed. "Yeah, you're right. I know that better than most. We've lost billions in revenue over the past few years of success. People call us a success. We rang that bell in New York City, and it looked like a success. The world looks at us as if we are a success. They want to use our devices regardless of who actually made it."

He took a long, slow sip of his black tea. When his lips parted from the porcelain, and the mug turned slightly, she could see a single black bead of tea drip lazily down its side. His disposition darkened, seemingly descending as quickly as that tiny drip of tea through the manufactured air and onto the office floor.

"But fuck them. We aren't a success. We can't even keep those people out of our security chips."

He placed an elbow on his standing desk, resting his hair in his hand. "I'm done caring about how to solve security. It's just a god damned cat and mouse cycle of nonsense." He looked her straight in the eyes. "Nonsense!" he loudly snarled. He looked downward, his other hand still attached to the vessel holding the blackened liquid. He continued more calmly.

"They forge our logos. They recreate our software. They steal our customers. We have a right to protect ourselves. Technically, if they use our trademarks, their devices are ours. We just didn't make them. If they're ours, we have a right. We have a god damned right to do with them as we please."

His eyes tightened as he stood up as straight as the flagpole next to him. "We have a god damned duty to our employees, our investors, and our country, to protect what's ours. If they're going to produce technology that they claim is ours, we have the right to take that technology. We have a right to destroy that technology."

He looked over at his standing desk, and hit a key on his laptop's keyboard. He glanced at the screen for a brief moment, then continued.

"I need a way to stop this nonsense. I'm sick of worrying about someone hacking into this or hacking into that. We need this game finished. No more cold war bullshit with fake engineers and shell companies overseas. I'm done. I'm fucking done. I need a way to brick every single device that claims it's one of ours. If it connects to the Internet and sends a message saying it's owned by Fit'd, Inc., I want it bricked. If it connects to a computer and identifies itself as Fit'd, Inc., I want it bricked. If it peers with another mesh device and claims it's Fit'd, Inc., I want it bricked. They're done. These people are fucking done. And you? You're going to write the exploit."

Her eyes widened again, this time in discomfort. She understood why he seemed so unable to hold back these worsening emotions. He was on the edge, if not slightly beyond it.

"But, we have absolutely no way of knowing how this will affect the end users!" Her right foot began tapping madly again, as she leaned forward in her chair. Her body barely hung on to the edge of her seat, practically mirroring how his mind must be teetering on its ethical edge, half ready to give itself to the wind, leaping recklessly into the abyss. "We can't possibly put people's lives at risk like that! You realize how many infinite scenarios there are for people using our technology! Think of how people are using wearables to monitor and control their pacemakers, their insulin pumps, their seizure reducers... There are people who could die if their products are suddenly unable to function!"

The VP briskly walked the few steps toward the shaken woman, with a pointed finger and furrowed eyebrows, "These people are putting themselves at risk by knowingly purchasing cloned technology! You said it yourself in your security review of a third-party clone: there was no guarantee that reproduced work could even come close to ensuring the confidentiality, integrity, or availability of a consumer's data! No guarantee!" he barked.

"But, sir!" her body was pinned against the back of the chair, as if forced there by a sudden atmospheric microburst. "The impoverished buy these knock-offs because they can't afford the real thing. There is a user base of millions in foreign countries that depend on this technology for their basic communication needs. It isn't about protecting our product, our trademark, or even our corporate persona." She calmed down as she heard the sensible words starting to emanate from her mouth.

"It's about a worldwide phenomenon that this company has created. That you've helped create! People want to participate, they want to be in this brave new world, but it's just a fact that not everyone can afford what we sell."

"By arbitrarily disabling these devices you're widening the communication gap between the have's and have-not's. Think about how clones of this company's technology are used to connect millions of people to the world. People in oppressive governments, people in religiously strict societies, people without access to broadband in their region. It's their only method for keeping up with worldwide evolution in culture. You're risking sending a large portion of the Internet back into the technological stone age. If you destroy these people's tools, they're going to have to essentially uplink other modern mesh devices, dependent on clones of our technology, to the Internet using the equivalent of ancient serial-port speeds. For what? Ten percent of what this company makes in revenue per quarter?"

The VP sat his mug down on the desk, his brow still furrowed. Half of his hair, where one hand had been nervously running its fingers, was sticking out sideways, in some laughable nod to a Hollywood mad man. The other side was eerily plastic, like some bizarre executive Ken doll. As he turned to the side, the rustled hair disappeared, and the words that came out of his mouth seemed even more despicable while rolling out of what seemed like a perfectly coiffed, button-downed executive.

"If we don't hit these companies where they hurt the most, the end users, we won't ever hurt them. We need to show them that it's their fault people are dying. We need to prove to them that what they are doing can hurt actual people." He turned to face her, his unkempt hair appearing as he further proclaimed his righteousness. Again, he glanced back at his laptop, gauging something, then quickly looked away.

"These companies are risking lives as it is. They make an inferior product that lacks the guarantees that we can make. People will get hurt eventually, and what if it's in the millions? We can put a stop to it now, and maybe only a couple thousand get hurt. If we act today, we can potentially save millions later. You can help me put an end to this. You can help me save those millions of lives. You can help save this company, if we can build the perfect remote exploit."

His disregard for human life was somehow not shocking to her. She wasn't sure why. Maybe it was always there, under the surface of his skin, hidden behind that natural hippy-turned-professional vibe. Maybe it was the fact that he claimed to care about the ecosystem, posturing with the Boulder, Colorado mindset, while driving a gas guzzling Porsche, and flying in a private jet whose pollution costs were offset by carbon credits. She didn't know why it made sense. It just did.

It wasn't shocking, but it was terrifying to her. Even if she quit, if he was this far gone, how could she trust him not to hurt her? Did anyone else even know about this? Was she the only one he told? Would he hurt her to keep this psychotic rant from going beyond these walls? Was this a test? It sure as hell didn't feel like a test. It felt real. It felt dangerous.

Suddenly, a pop-up appeared in her line of vision. Her own eye-contact heads-up-display was notifying her that she was perspiring and had an elevated heart rate, but didn't seem to be moving in any particular direction. "Are you feeling okay?" the artificial intelligence asked in a little text pop-up box, as her fitness statistics hovered in little graphic-user-interface clouds throughout her field of vision. "I can sense that you seem to be running, but our movement mesh shows you aren't moving. Would you like to recalibrate?"

The intrusion of these observable metrics into this ridiculously cartoonish scenario simply furthered her disbelief that any of this was actually happening. This began to seem more and more like a bizarre and belated Halloween prank. As her heart thumped louder and louder, she couldn't help but break into a humiliatingly inappropriate grin. Was he crazy? Was she? Was any of this happening?

The eye-contact queried again: "Would you like to recalibrate?"

"Yes, this is real." he stated with an absurd calm that sent chills down her spine. He instantly seemed more in control than ever. He was almost gloating! Whatever he kept glancing at on his laptop screen was reassuring him. "This is very real."

"How did you know that's what I was thinking?! You're putting me through some kind of fucked up joke, right? Some kind of loyalty test? This isn't funny. I don't think it's funny." She tried to gather herself. She stood up, but seemed frozen by his lack of reaction. "I quit. I have to quit. Even if this is a joke or a test, it's too fucked up. I can't..."

"You can't?" he said. He grabbed his standing desk and twisted it back, flattening the desktop surface before hitting a switch with his foot that enabled the surface to be lowered, then loudly slammed the desk down into its sitting position. The shotgun-like boom of the thick, flat, cherry wood smacking more thick flat wood was unbearable! He slowly wheeled the desk over to the center of the room, in front of a setting San Diego sun. "You can't what? Change the world? You're afraid of the cost of change. I get it. It takes a lot of bravery to do what we do here, to make real, tangible change. Sometimes, that cost is unthinkable. But, we do it, because we can aff...."

"Because you fucking can!" she exclaimed, infuriated by his sudden calm. "Say it! Because you fucking can! Knock it off with the perpetual rhetoric nonsense! You do it because you fucking can!" Tears began to well up in her eyes, still waiting for the rest of the executive team to burst through the doorway exclaiming this horrible test of will and ethics was over.

The sun finally lowered over the late afternoon horizon, sending a green flash, and pink hues barreling into the suddenly quiet office room. The flat gray surface of the battleship was devoid of little men in

white. The barrel of the turret they were polishing earlier now seemed to be pointed in her direction. Was it pointing this way earlier? She couldn't remember. It must have been.

She felt her temperature rising, even with the sun disappearing. Her HUD popped up another little text box into her field of vision exclaiming that her core temperature has elevated to 99 degrees Fahrenheit. She wanted desperately to run out of the office. But where would she go? And would the guards at the building exits stop her? Or would there be little men in white to cleanse this building of her presence?

"If you run, that will be a big problem for you," he smirked. "Please, sit back down. We have much to discuss."

"How the fuck?" Suddenly, she saw it. He wasn't glancing at instant messages. It wasn't stock prices he had been monitoring throughout the discussion. As the sun set, the world outside darkened almost in parallel with the tone in the office. And it was there, a clear reflection in the wall of windows in front of her. As her vital statistics updated in real time on her HUD, she could see the updates slightly delayed on the screen of his laptop. He had been playing with her emotions the entire time! He was watching how she would react, how she would process what he told her, whether she was a threat to him... He could predict what she was thinking by analyzing all the sensors in their wearable mesh network: the heart rate sensor, the perspiration sensor, 3D body positioning, mouth dryness, blink-rate analysis, muscle tension monitoring... He couldn't read her mind, but his machine learning software was analyzing what she was most likely thinking, and it was god damned close...

She recklessly shoved a black painted fingernail into her eye, nearly scratching her retina as she dug out the wireless-enabled contact. Her teeth clenched as she tried to stop herself from reacting from the pain. "Mother fucker!!! Fuck you!"

He laughed casually, motioning again to the chair. "Please, take a seat."

"Why should I! You're fucking insane!"

"Why? Because everyone you know and love wears these sensors now. Not the cheap knock offs. The real ones. And we can access them all remotely thanks to the security architecture that you signed off on. Not to mention, someone told those people how to break these security chips, and that report was for internal use only. Someone will get blamed. We both know it wasn't you, but how can you prove it wasn't?"

She almost spoke the obvious...

"Yes, you could tell them all about the so-called evil we can do here. Blah, fucking blah. You'll just sound like another pressured paranoid security engineer that finally snapped, gone schizophrenic, thinking trojan horses are communicating to the devices in your SCIF using sound waves projected through your own body. You'll be another fucking psychotic loser that no one gives a shit about because no one is strong enough to be comfortable around your Enemy Of The State, Three Days of the Condor, stereotypical bullshit."

"They will listen to me..."

"Listen to a blue haired ex-punk rock wannabe corporate security fuck? The door is right behind you. There are lots of people in the building right now. Want to give it a shot? Go for it." his smile was almost razor-thin. "Go ahead. See what they think."

Her eyes were blood red from anger, humiliation, her fingertip, and a feeling of complete loss of control. As she stood in the center of the room, her foot began to twitch, tapping out some unheard, emotionally exhausting, industrial-rock song.

"Now, then. Why don't you sit down. We have much to discuss."

Her body shook as she sat back down in the L3 reproduction. She could feel the noiseless ventilation system come back on. As her hands touched the cold metal frame of the chair underneath her, the frigid air slid like unwanted fingers down the back of her neck. In silence, she watched the American flag in the corner wave hypnotically to the oscillation of the hidden fans, as the fluorescent lights flickered above the darkened crescent skin under the man's machinated, inanimate eyes.

The world outside had fully relinquished what was left of its grip on the evening sun, as if it had given up its fight against the incessant hum of the digitally controlled fluorescent lighting. A pulsing, flickering, buzzing, manufactured light which bullied its way through these office windows and outside, into the uncertain San Diego streets. A reflection in the windows shone a familiar pop-up flashing on the man's laptop's screen.

"Would you like to recalibrate?"

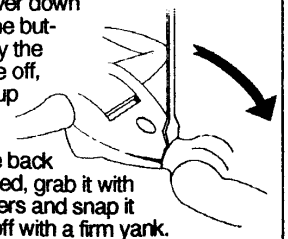
OFFICIAL Barbie Liberation Organization

Barbie/G.I.JOE HOME SURGERY INSTRUCTIONS

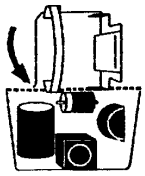
You Will Need

- *Teen Talk* Barbie Doll
- 12" talking G.I. Joe
- soldering iron
- electric solder
- Epoxy (not fast drying)
- 2 sharp screwdrivers
- 1 coping or hack saw
- 12" electrical wire
- hot glue (or similar)
- switch (see step 12)

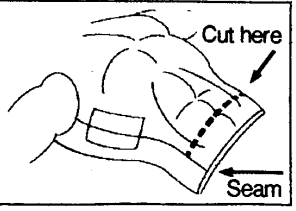
1. To open Barbie, insert a screwdriver firmly into the joint at the base of the spine. With a quick jerk, snap the screwdriver down toward the buttocks. Pry the backplate off, working up from the waist. Once the back is loosened, grab it with your fingers and snap it straight off with a firm yank. Do not twist. Remove head, arms, and legs. Gently loosen circuit board. Break off tab holding speaker in place. Remove speaker/circuit board.



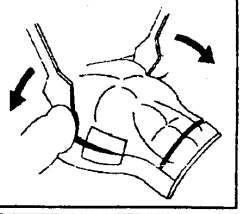
2. Using saw, sever battery contacts from rest of circuit board as shown. Battery contacts go back into doll.



3. To open G.I. Joe, remove batteries and pop off head. Using saw, make incision across abdomen from seam to seam. **Be careful not to cut wires underneath.**

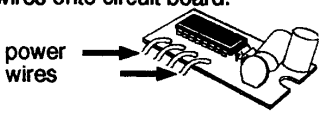


4. Start prying front/back plates apart at neck and work down towards shoulders. **Careful - neck is fragile.** Once shoulders are split, insert screwdrivers into joints where arms meet torso. Pry torso apart from both arms simultaneously.

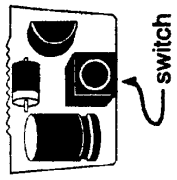


5. Cut bracket holding Joe's circuit board in place and loosen board, speaker, and switch.

6. Locate power wires (red & black) running from Joe to contacts on circuit board. Heat contacts with soldering iron. Remove wires from board but leave them attached to Joe. Solder two similar replacement wires onto circuit board.

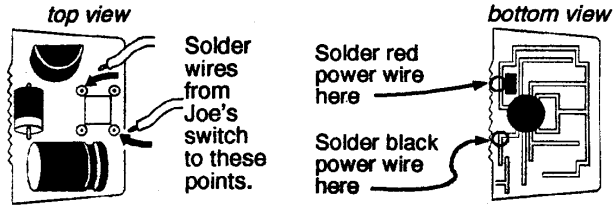


7. Locate the switch on Barbie's circuit board. Heat the four solder points and remove. A solder-removing bulb may help.



8. When removing Joe's switch, make a note of where the switch wires meet the circuit board. Heat contacts and remove switch.

9. Wire Joe's power and switch to Barbie's circuit board as shown. Install board, speaker, and switch back into Joe. Hot glue works well to anchor everything in place. Speaker should be firmly glued to breastplate for maximum volume.

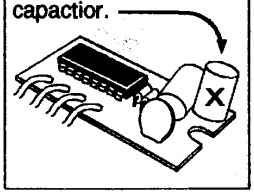


10. **IMPORTANT**: When running the Barbie circuit board in Joe, use only three batteries. You may want to re-wire the battery contacts, or substitute something to take up the extra space. A filed-down conductive nail wrapped in tape works well as a pseudo-battery.

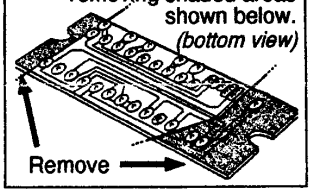
11. There are two options for re-installing Barbie's switch. The first (and more difficult) is to use a small, stiff, non-conductive scrap of circuit board, plastic or similar material. Mount the switch on the board, and sandwich it between the board and the button on Barbie's back. Glue the board to the posts on Barbie's back. If done carefully, Barbie need never know she's been under the knife.

12. The second option is to use a small momentary contact switch. (Radio Shack Cat. No. 275-1571B) Mount it in place of the button in Barbie's back. It's easier and more permanent, although Barbie no longer looks like everyone else.

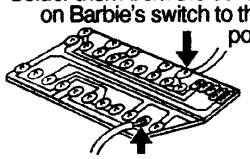
13. Unfortunately, Joe's circuit board will not fit properly into Barbie without modification. First, desolder and remove this capactior.



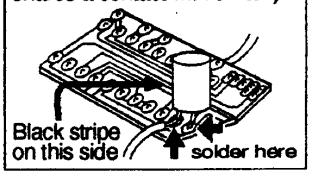
14. Next, cut down board by removing shaded areas shown below. *(bottom view)*



15. Cut two 2" pieces of wire. Solder them from the contacts on Barbie's switch to these points.

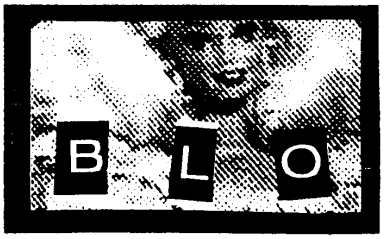


16. Re-solder capacitor as shown. (Note: capacitor shares a contact with switch)



17. Cut any additional unused space off the board. Solder the two wires from step 6 to Barbie's battery contacts.

18. Fitting the board into Barbie is tricky. You may need to bend the capacitors or shave the posts in her chestplate. Before re-sealing Barbie or Joe, first make sure body parts fit together properly. Apply epoxy around rim of front and back plate. Quick-drying epoxy is not reccomended, as it leaves little room for error. First insert both neck sections into the head, insert the arms and legs, then clamp the doll together. To touch up any scars or mistakes, use plumber's epoxy putty and model paint.



11 A Call for PoC

by Pastor Manul Laphroaig, Proselytizer of Weird Machines

Howdy, neighbor! Is that a fresh new PoC you are hugging so close? Don't stifle it, neighbor, it's time for it to see the world, and what better place to do it than from the pages of the famed International Journal of PoC or GTFO? It will be in a merry company of other PoCs big and small, bit-level and byte-level, raw binary or otherwise, C, Python, Assembly, hexdump or any other language. But wait, there's more—our editors will groom it for you, and dress it in the best Sunday clothes of proper church English. And when it looks proudly back at you from these pages, in the company of its new friends, won't that make you proud? So set that little PoC free, neighbor, and let it come to me, pastor@phrack.org!

– – – –   – – –   – – – –   – – –   – – –   – – –   –   – – – –   – – –

Do this: write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, or German. If you don't speak those languages, we'll dig up a translator.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do. Don't try to make it thorough or broad.

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me how to make music that also parses as PSK31, RTTY, or WeFax. Show me how to reverse engineer SoftStrip barcodes. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

NEW! from ads

6809 SINGLE-BOARD COMPUTER
S-100 bus

- IEEE S-100 Proposed Standard
- 2K RAM
- 4K/8K/16K ROM
- PIA, ACIA Ports
- adsMON; 6809 Monitor Available

P.C. Board & Manual Presently Available

ALL PC BOARDS FROM ADS ARE SOLDER MASKED, WITH GOLD CONTACTS, & PARTS LAYOUT SILK SCREENED ON BOARD.
Add 50¢ postage & handling per item.
Ill. residents add sales tax.

Sound Effects . . . Sound Effects . . . !!!
© NOISEMAKER * & © NOISEMAKER II **
S-100 bus   Apple IItm bus

ADD "SPACESHIP" SOUNDS, PHASERS, GUNSHOTS, TRAINS, MUSIC, SIRENS, ETC.! UNDER SOFTWARE CONTROL!!!

- Soundboards Use GI AY 3-8910 I.C.'s to Generate Programmable Sound Effects.
- On Board Audio Amp. Breadboard Area With + 5 & GND.
- Noise Sources • Envelope Generators • I/O Ports

PCB & Manual: *39.95 (NM); **34.95 (NM II)

!!!!!!ATTENTION APPLE II USERS!!!!!!
Assembled and Tested NM II Units Now Available!!!

Call or Write for Details. ads

Ackerman Digital Systems, Inc., 110 N. York Road, Suite 208, Elmhurst, Illinois 60126   (312) 530-8992