PoC||GTFO

PASTOR LAPHROAIG'S MERCY SHIP HOLDS STONES FROM THE IVORY TOWER, BUT ONLY AS BALLAST!

Üres hasnak elég a szép szó; это самиздат. `pocorgtfo13.pdf`. October 18, 2016.
€0, $0 USD, 10s 6d GBP, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő).

Legal Note: In solidarity with ℋ, the Author Formerly Known as Homer Hickam, we place no restrictions of any kind upon our authors. They are quite welcome to do whatever the hell they like with their own work, in any medium they like, including but not limited to endeavors of theater and interpretive dance.

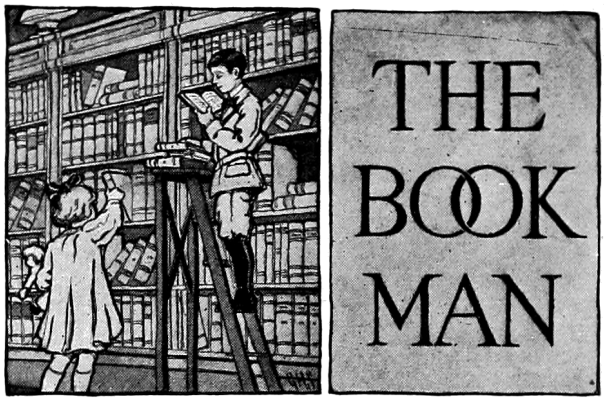
Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo13.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. We like the following mirrors.
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`http://www.sultanik.com/pocorgtfo/`

Technical Note: As described in PoC‖GTFO 13:10, `pocorgtfo13.pdf` is a polyglot that may be interpreted as *both* a PDF *and* a PostScript file. As a PDF, this file is mostly harmless, but we warn you that the Postscript will render differently each time, including both a randomly generated maze and—if Tavis Ormandy hasn't killed such a lovely bug yet—a copy of your `/etc/passwd` file.

Cover Art: The cover artwork from this issue is by Harry Clarke, first used to illustrate the poem Sea Fever by John Masefield in the collection The Year's at the Spring, 1920.

Printing Instructions: Pirate print runs of this journal are most welcome! 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, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. 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 --vanilla --paper a3paper pocorgtfo13.pdf -o pocorgtfo13-book.pdf
```



Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen

and sundry others

1 Read me if you want to live!

Neighbors, please join me in reading this fourteenth 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 reverse engineering and worshippers of weird machines. This fourteenth release is given on paper to the fine neighbors of São Paulo, San Diego, and Budapest.

If you are missing the first thirteen 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, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, or the thirteenth in Montréal.

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo13.pdf`. It is valid as PDF, ZIP, and PostScript; please read it with Adobe Reader, `unzip`, and `gv`.

We begin on page 5 with the story of how **STAR RAIDERS** by Doug Neubauer for the Atari 400 was taken apart by Lorenz Weist, from a mere ROM cartridge dump to annotated and literate 6502 disassembly. By a stroke of luck, Lorenz was able to read Doug's original source code for the game after completing his reverse engineering project, giving him the rare opportunity to confirm his understanding of the game's design and behavior.

On page 24, James Forshaw introduces us to a nifty little trick for simplifying reliable exploitation of race condition vulnerabilities. Rather than spin up a dozen attempts to improve racetrack odds, he instead induces situations with pathological performance penalties to Windows NT system calls, stunning the threads of execution that might interfere with his exploit for twenty minutes or more!

Micah Elizabeth Scott continues to send us brilliant articles that refuse to be described by a single abstract, so let's just say that on page 30 she explains a USB magic trick in which her FaceWhisperer board—combining the Facedancer and the Chip Whisperer—is able to reliably glitch the USB stack of an embedded device to dump its firmware. Or, we could say that on page 30 she explains how to use undocumented commands from that firmware dump to program the Harvard device by ROP. Or, we could say that on page 30 she shows you to read RFID tags with a Wacom tablet. These tricks are all the same article, and you'd be a fool not to read it.

In PoC||GTFO 10:8, Travis Goodspeed jailbroke the Tytera MD380 radio to allow for firmware extraction and patching. Since then, a lively open source project has sprung up, with fancy new features and fixes to old bugs. On page 38, he describes how to rip the AMBE audio codec out of the radio firmware, transforming it into a command line audio processing tool that runs on any Linux workstation. Similar tricks can be used to quickly toss together emulators for many ARM and PowerPC embedded systems, re-using their library functions, or fuzzing their parsers in the familiar environment of an everyday laptop.

Evan Sultanik is back with a safe cracking adventure that could only be expressed as a play in three acts, narrated by our own Pastor Manul Laphroaig. Speaking parts are available for Alice Feynman, Bob Schrute, Havva al-Kindi, and the ghost of Paul Erdős. You'll find Evan's script on page 43.

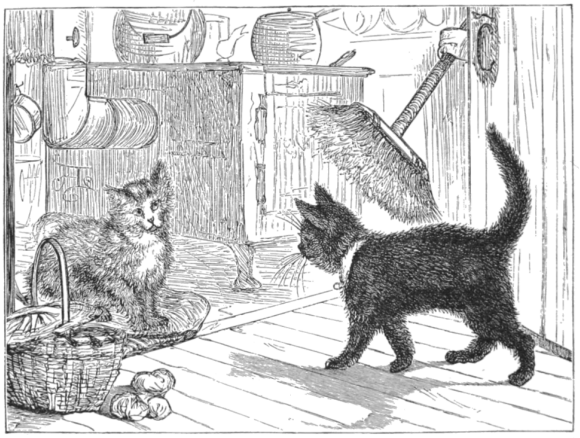
Matt Knight has been reverse engineering the PHY of LoRa, a low-power protocol for sub-GHz wireless networking over long distances. On page 48 you will find not just the protocol details that allowed him to write an open source receiver, but, far more importantly, you will also find the methods by which he reverse engineered this information from captured packets, vague application notes, and the outright lies of the patent application.

Pastor Manul Laphroaig, your friendly neighborhood evangelist of the gospel of the weird machines, has a sermon for you on page 60. He reminds us that science takes place neither on stage in front of a live studio audience nor in committees and government offices, but over a glass of fine scotch that's accompanied by finer conversation of practitioners. In the same way that we oughtn't put Tim the "Tool Man" Taylor in charge of vocational education, we ought to leave the teaching of science to those who do it, not those who talk about it on TV.

Geoff Chappell is an old-school reverse engineer, an x86 archaeologist who has spent the past twenty-four years reading Windows binaries to identify all the forgotten features and corner cases that the rest of us might take for granted.[1] On page 63, he introduces us to the mystery of Microsoft's Shim Database Compiler, an unpublished tool for compiling driver shims that doesn't seem to be available to the outside world. Geoff shows us that, in fact, the tool is available, wrapped up inside of a GUI as `QFixApp.exe` or `CompatAdmin.exe`. By patching the program to expose its intact `winmain()`, he can recover the long-lost `ShimDBC.exe` for compiling Windows driver compatibility shims from XML!

Evan Sultanik and Philippe Teuwen have teamed up on page 71, to explain the inner workings of `pocorgtfo13.pdf`, which you can rename to read as `pocorgtfo13.zip` or `pocorgtfo13.ps`.

On page 72, the last page, we pass around the collection plate. Our church has no interest in cash or cheques, but we'd love your donation of a nifty reverse engineering story. Please send one our way.

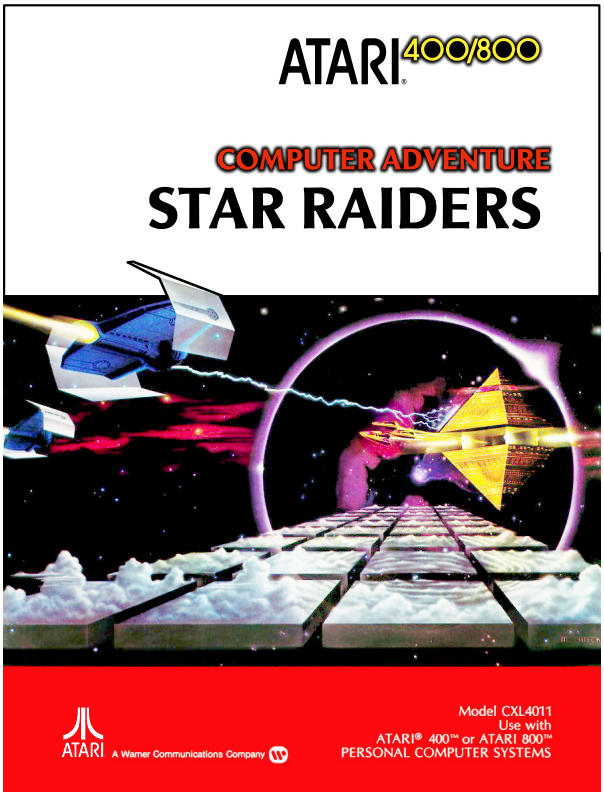


[1] Geoff was the first to discover Aaron R. Reynolds' "AARD" code from the beta release of Windows 3.1 that intentionally broke compatibility with DR-DOS. He also has a delightful article on exactly how AOL exploited a buffer overflow in their own AOL Instant Messenger client to distinguish it from Microsoft's clone, MSN Messenger.

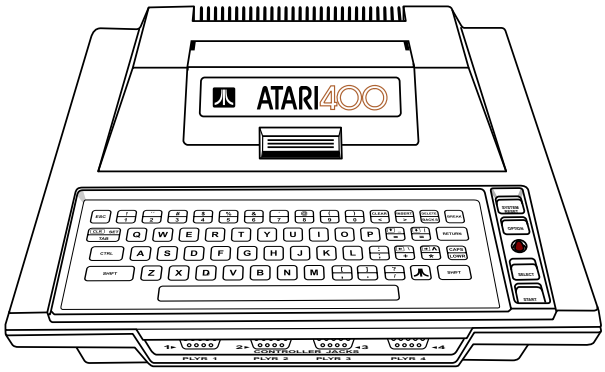
2 Reverse Engineering Star Raiders

by Lorenz Wiest

2.1 Introduction



STAR RAIDERS is a seminal computer game published by Atari Inc. in 1979 as one of the first titles for the original Atari 8-bit Home Computer System (Atari 400 and Atari 800). It was written by Atari engineer Doug Neubauer, who also created the system's POKEY sound chip. **STAR RAIDERS** is considered to be one of the ten most important computer games of all time.[2].



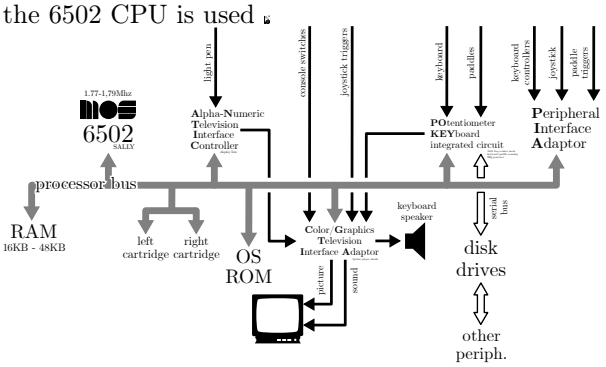
The game is a 3D space combat flight simulation where you fly your starship through space, shooting at attacking Zylon spaceships. The game's universe is made up of a 16 × 8 grid of sectors. Some of them contain enemy Zylon units, some a friendly starbase. The Zylon units converge toward the starbases and try to destroy them. The starbases serve as repair and refueling points for your starship. You move your starship between sectors with your hyperwarp drive. The game is over if you have destroyed all Zylon ships, have ran out of energy, or if the Zylons have destroyed all starbases.

At a time when home computer games were pretty static – think SPACE INVADERS (1978) and PAC MAN (1980) – **STAR RAIDERS** was a huge hit because the game play centered on the very dynamic 3D first-person view out of your starship's cockpit window.

The original Atari 8-bit Home Computer System

[2]"Is That Just Some Game? No, It's a Cultural Artifact." Heather Chaplin, The New York Times, March 12, 2007.

has up to 48 KB RAM and uses a Motorola 6502 CPU. The same CPU is also used in the Apple II, the Commodore C64 (a 6502 variant), and the T-800 Terminator.[3] Several proprietary Atari custom chips provide additional capabilities to the system. **STAR RAIDERS** shows off many of them: 5 Players (sprites), mixed text and pixel graphics modes, dynamic Display Lists, a custom character set, 4-channel sound, Vertical Blank Interrupt and Display List Interrupt code – even the BCD mode of the 6502 CPU is used.



I have been always wondering what made **STAR RAIDERS** tick. I was especially curious how that 3D first-person view star field worked, in particular the rotations of the stars when you fly a turn. So I decided to reverse engineer the game, aiming at a complete, fully documented assembly language source code of **STAR RAIDERS**.

```
S T A R   R A I D E R S
for the Atari 8-bit Home Computer System
Reverse-engineered and documented assembly language source code
by
Lorenz Wiest
(lo.wiest(at)web.de)
First Release
22-SEP-2015
Last Update
10-AUG-2016
STAR RAIDERS was created by Douglas Neubauer
STAR RAIDERS was published by Atari Inc.
```

In the following sections I'll show you how I approached the reverse engineering effort, introduce my favorite piece of code in **STAR RAIDERS**, talk about how the tight memory limits influenced the implementation, reveal some bugs, point at some mysterious code, and explain how I got a grip on documenting **STAR RAIDERS**. From time to time, to provide some context to you, I will reference memory locations of the game, which you can look up in the reverse engineered, complete, and fully documented assembly language source code of **STAR RAIDERS** available on GitHub.[4]

2.2 Getting Started

STAR RAIDERS is distributed as an 8 KB ROM cartridge, occupying memory locations `$A000` to `$BFFF`.

The obvious first step was to prod a ROM dump with a disassembler and to apply Atari's published hardware and OS symbols to the disassembly. To my surprise this soon revealed that code and data were clearly separated into three parts:

`$A000`	–	`$A149`	Data (Part 1 of 2)
`$A14A`	–	`$B8DE`	Code (6502 instructions)
`$B8DF`	–	`$BFFF`	Data (Part 2 of 2)

This clear separation helped me instantly to get an overview of the code part, as I could create a disassembly of the code in one go and not having to sift slowly through the bytes of the ROM, deciding which ones are instructions and which ones are data.

Closer inspection of the code part revealed that it was composed of neatly separated subroutines. Each subroutine handles a specific task. The largest subroutine is the main game loop `GAMELOOP` (`$A1F3`), shown in Figure 1. What I expected to be spaghetti code – given the development tools of 1979 and the substantial amount of game features crammed into the 8K ROM – turned out to be surprisingly structured code. Table 1 lists all subroutines of **STAR RAIDERS**, as their function emerged during the reverse engineering effort, giving a good overview how the **STAR RAIDERS** code is organized.

Figure 2 shows the "genome sequence" of the **STAR RAIDERS** 8 KB ROM: The 8192 bytes of the game are stacked vertically, with each byte represented by a tiny, solid horizontal line of 8 pixels. This stack is split into strips of 192 bytes, arranged side-by-side. Alternating light and dark blue areas represent bytes of distinct subroutines. Alternating light and dark green and purple areas represent bytes of distinct sections of data (lookup tables, graphical shapes, etc.). When data bytes represent graphical shapes, the solid line of a byte is replaced by its actual bit pattern (in purple color).

There are a couple of interesting things to see:

- The figure reflects the ROM's separation into a data part (green and purple), a code part (blue), and one more data part (green and purple).

- The first data part contains mostly the custom

[3] In the movie TERMINATOR (1984) there are scenes showing the Terminator's point of view in shades of red. In these scenes lines of source code are listed onscreen. Close inspection of still frames of the movie reveal this to be 6502 assembly language source code.

[4] `git clone https://github.com/lwiest/StarRaiders` or `unzip pocorgtfo13.pdf StarRaiders.zip`

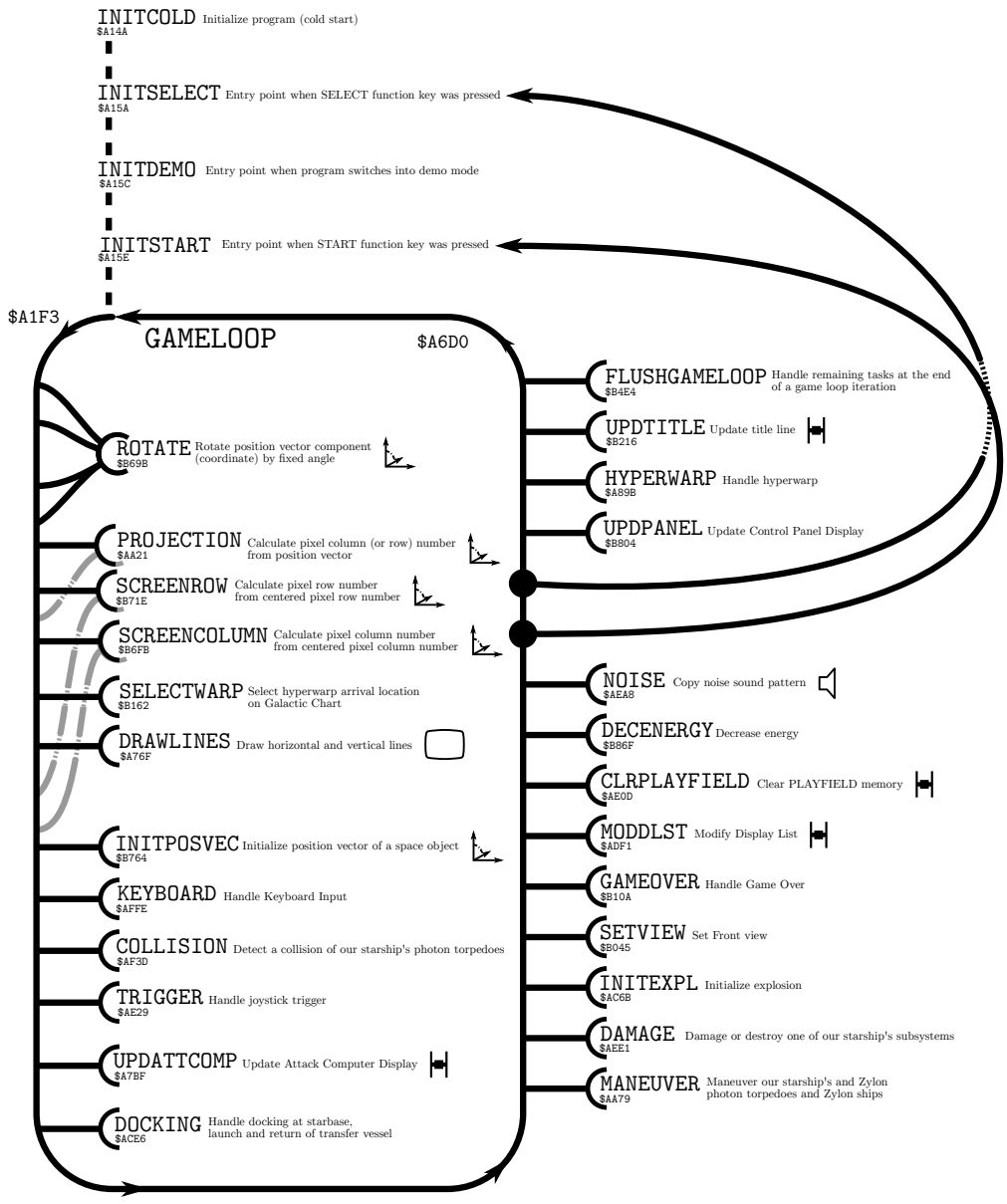


Figure 1. Simplified Call Graph of Start Up and Game Loop

1	$A14A	INITCOLD	Initialize program (Cold start)
	$A15A	INITSELECT	Entry point when SELECT function key was pressed
3	$A15C	INITDEMO	Entry point when program switches into demo mode
	$A15E	INITSTART	Entry point when START function key was pressed
5	$A1F3	GAMELOOP	Game loop
	$A6D1	VBIHNDLR	Vertical Blank Interrupt Handler
7	$A718	DLSTHNDLR	Display List Interrupt Handler
	$A751	IRQHNDLR	Interrupt Request (IRQ) Handler
9	$A76F	DRAWLINES	Draw horizontal and vertical lines
	$A782	DRAWLINE	Draw a single horizontal or vertical line
11	$A784	DRAWLINE2	Draw blip in Attack Computer
	$A7BF	UPDATTCOMP	Update Attack Computer Display
13	$A89B	HYPERWARP	Handle hyperwarp
	$A980	ABORTWARP	Abort hyperwarp
15	$A987	ENDWARP	End hyperwarp
	$A98D	CLEANUPWARP	Clean up hyperwarp variables
17	$A9B4	INITTRAIL	Initialize star trail during STAR TRAIL PHASE of hyperwarp
	$AA21	PROJECTION	Calculate pixel column (or row) number from position vector
19	$AA79	MANEUVER	Maneuver our starship's and Zylon photon torpedoes and Zylon ships
	$AC6B	INITEXPL	Initialize explosion
21	$ACAF	COPYPOSVEC	Copy a position vector
	$ACC1	COPYPOSXY	Copy x and y components (coordinates) of position vector
23	$ACE6	DOCKING	Handle docking at starbase, launch and return of transfer vessel
	$ADF1	MODDLST	Modify Display List
25	$AE0D	CLRPLAYFIELD	Clear PLAYFIELD memory
	$AE0F	CLRMEM	Clear memory
27	$AE29	TRIGGER	Handle joystick trigger
	$AEA8	NOISE	Copy noise sound pattern
29	$AECA	HOMINGVEL	Calculate homing velocity of our starship's photon torpedo 0 or 1
	$AEE1	DAMAGE	Damage or destroy one of our starship's subsystems
31	$AF3D	COLLISION	Detect a collision of our starship's photon torpedoes
	$AFFE	KEYBOARD	Handle Keyboard Input
33	$B045	SETVIEW	Set Front view
	$B07B	UPDSCREEN	Clear PLAYFIELD, draw Attack
35	$B10A	GAMEOVER	Handle game over
	$B121	GAMEOVER2	Game over (Mission successful)
37	$B162	SELECTWARP	Select hyperwarp arrival location on Galactic Chart
	$B1A7	CALCWARP	Calculate and display hyperwarp energy
39	$B216	UPDTITLE	Update title line
	$B223	SETTITLE	Set title phrase in title line
41	$B2AB	SOUND	Handle sound effects
	$B3A6	BEEP	Copy beeper sound pattern
43	$B3BA	INITIALIZE	More game initialization
	$B4B9	DRAWGC	Draw Galactic Chart
45	$B4E4	FLUSHGAMELOOP	Handle remaining tasks at the end of a game loop iteration
	$B69B	ROTATE	Rotate position vector component (coordinate) by fixed angle
47	$B6FB	SCREENCOLUMN	Calculate pixel column number from centered pixel column number
	$B71E	SCREENROW	Calculate pixel row number from centered pixel row number
49	$B764	INITPOSVEC	Initialize position vector of a space object
	$B7BE	RNDINVXY	Randomly invert the x and y components of a position vector
51	$B7F1	ISSURROUNDED	Check **if** a sector is surrounded by Zylon units
	$B804	UPDPANEL	Control Panel Display
53	$B86F	DECENERGY	Decrease energy
	$B8A7	SHOWCOORD	Display a position vector component (coordinate) in
55			Control Panel Display
	$B8CD	SHOWDIGITS	Display a value by a readout of the Control Panel Display

Table 1. Star Raiders Subroutines

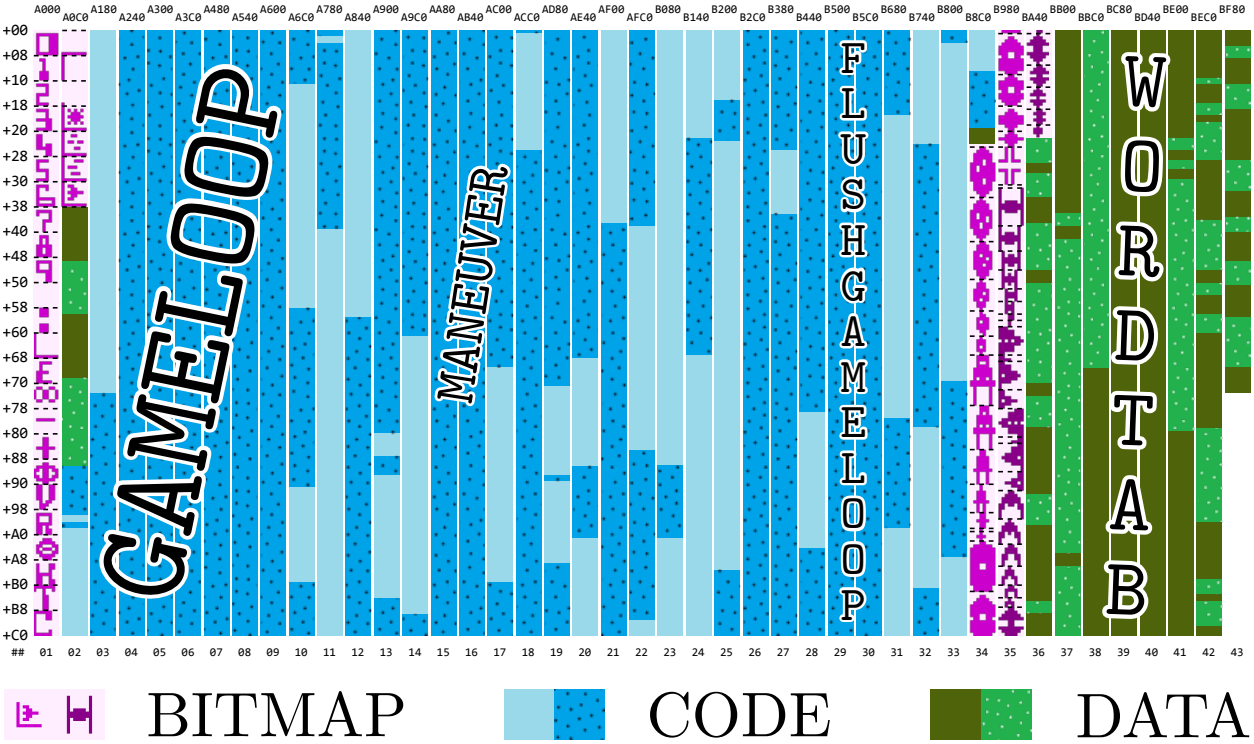


Figure 2. Genome Sequence of the STAR RAIDERS ROM

font (in strips 1-2).

- The largest contiguous (dark) blue chunk represents the 1246 bytes of the main game loop `GAMELOOP` (`$A1F3`) (in strips 3-10).
- At the beginning of the second data part are the shapes for the Players (sprites) (in strips 34-36).
- The largest contiguous (light) green chunk represents the 503 bytes of the game's word table `WORDTAB` (`$BC2B`) (in strips 38-41).

A good reverse engineering strategy was to start working from code locations that used Atari's published symbols, the equivalent of piecing together the border of a jigsaw puzzle first before starting to tackle the puzzle's center. Then, however, came the inevitable and very long stretch of reconstructing the game's logic and variables with a combination of educated guesses, trial-and-error, and lots of patience. At this stage, the tools I used mostly were nothing but a text editor (Notepad) and a word processor (Microsoft Word) to fill the gaps in the documentation of the code and the data. I also created a memory map text file to list the used memory locations and their purpose. These entries were continually updated – and more than often discarded after it turned out that I had taken a wrong turn.

2.3 A Programming Gem: Rotating 3D Vectors

What is the most interesting, fascinating, and unexpected piece of code in **STAR RAIDERS**? My pick would be the very code that started me to reverse engineer **STAR RAIDERS** in the first place: subroutine `ROTATE` (`$B69B`), which rotates objects in the game's 3D coordinate space (shown in Figure 3). And here is why: Rotation calculations usually involve trigonometry, matrices, and so on – at least some multiplications. But the 6502 CPU has only 8-bit addition and subtraction operations. It does not provide either a multiplication or a division operation – and certainly no trig operation! So how do the rotation calculations work, then?

Let's start with the basics: The game uses a 3D coordinate system with the position of our starship at the center of the coordinate system. The locations of all space objects (Zylon ships, meteors, pho-

ton torpedoes, starbase, transfer vessel, Hyperwarp Target Marker, stars, and explosion fragments) are described by a position vector relative to our starship.

A position vector is composed of an x, y, and z component, whose values I call the x, y, and z coordinates with the arbitrary unit <KM>. The range of a coordinate is −65536 to +65535 <KM>.

Each coordinate is a signed 17-bit integer number, which fits into three bytes. Bit 16 contains the sign bit, which is 1 for positive and 0 for negative sign. Bits 15 to 0 are the mantissa as a two's-complement integer.

```
   Sign       Mantissa
2       B16 B15...B8 B7....B0
          |  |       |  |       |
4 0000000* ******** ********
```

Some example bit patterns for coordinates:

```
   00000001 11111111 11111111 = +65535 <KM>
2  00000001 00000001 00000000 =   +256 <KM>
   00000001 00000000 11111111 =   +255 <KM>
4  00000001 00000000 00000001 =     +1 <KM>
   00000001 00000000 00000000 =     +0 <KM>
6  00000000 11111111 11111111 =     −1 <KM>
   00000000 11111111 11111110 =     −2 <KM>
8  00000000 11111111 00000001 =   −255 <KM>
   00000000 11111111 00000000 =   −256 <KM>
10 00000000 00000000 00000000 = −65536 <KM>
```

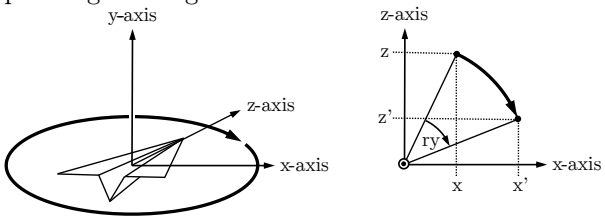
The position vector for each space object is stored in nine tables (3 coordinates × 3 bytes for each coordinate). There are up to 49 space objects used in the game simultaneously, so each table is 49 bytes long:

XPOSSIGN	XPOSHI	XPOSLO
($09DE..$0A0E)	($0A71..$0AA1)	($0B04..$0B34)
YPOSSIGN	YPOSHI	YPOSLO
($0A0F..$0A3F)	($0AA2..$0AD2)	($0B35..$0B65)
ZPOSSIGN	ZPOSHI	ZPOSLO
($09AD..$09DD)	($0A40..$0A70)	($0AD3..$0B03)

With that explained, let's have a look at subroutine ROTATE ($B69B). This subroutine rotates a position vector component (coordinate) of a space object by a fixed angle around the center of the 3D coordinate system, the location of our starship. This operation is used in 3 out of 4 of the game's view modes (Front view, Aft view, Long-Range Scan view) to rotate space objects in and out of the view.

2.3.1 Rotation Mathematics

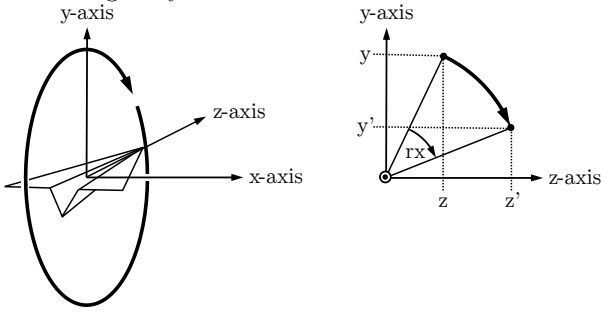
The game uses a left-handed 3D coordinate system with the positive x-axis pointing to the right, the positive y-axis pointing up, and the positive z-axis pointing into flight direction.



A rotation in this coordinate system around the y-axis (horizontal rotation) can be expressed as

$$x' = \cos(r_y)x + \sin(r_y)z \qquad (1)$$
$$z' = -\sin(r_y)x + \cos(r_y)z$$

where r_y is the clockwise rotation angle around the y-axis, x and z are the coordinates before this rotation, and the primed coordinates x' and z' the coordinates after this rotation. The y-coordinate is not changed by this rotation.



A rotation in this coordinate system around the x-axis (vertical rotation) can be expressed as

$$z' = \cos(r_x)z + \sin(r_x)y \qquad (2)$$
$$y' = -\sin(r_x)z + \cos(r_x)y$$

where r_x is the clockwise rotation angle around the x-axis, z and y are the coordinates before this rotation, and the primed coordinates z' and y' the coordinates after this rotation. The x-coordinate is not changed by this rotation.

2.3.2 Subroutine Implementation Overview

A single call of subroutine ROTATE ($B69B) is able to compute one of the four expressions in Equations 1 and 2. To compute all four expressions to

get the new set of coordinates, this subroutine has to be called four times. This is done twice in pairs in `GAMELOOP` (`$A1F3`) at `$A391` and `$A398`, and at `$A3AE` and `$A3B5`, respectively.

The first pair of calls calculates the new x and z coordinates of a space object due to a horizontal (left/right) rotation of our starship around the y-axis following the expressions of Equation 1.

The second pair of calls calculates the new y and z coordinates of the same space object due to a vertical (up/down) rotation of our starship around the x-axis following the expressions of Equation 2.

If you look at the code of `ROTATE` (`$B69B`), you may be wondering how this calculation is actually executed, as there is neither a sine nor cosine function call. What you'll actually find implemented, however, are the following calculations:

Joystick Left

$$\begin{aligned} x &:= x + z/64 \\ z &:= -x/64 + z \end{aligned} \tag{3}$$

Joystick Right

$$\begin{aligned} x &:= x - z/64 \\ z &:= x/64 + z \end{aligned} \tag{4}$$

Joystick Down

$$\begin{aligned} y &:= y + z/64 \\ z &:= -y/64 + z \end{aligned} \tag{5}$$

Joystick Up

$$\begin{aligned} y &:= y - z/64 \\ z &:= y/64 + z \end{aligned} \tag{6}$$

2.3.3 CORDIC Algorithm

When you compare the expressions of Equations 1–2 with expressions of Equations 3–6, notice the similarity between the expressions if you substitute[5]

$$\begin{aligned} \sin(r_y) &\to 1/64 \\ \cos(r_y) &\to 1 \\ \sin(r_x) &\to 1/64 \\ \cos(r_x) &\to 1 \end{aligned}$$

From $\sin(r_y) = 1/64$ and $\sin(r_x) = 1/64$ you can derive that the rotation angles r_y and r_x by which the space object is rotated (per game loop iteration) have a constant value of 0.89°, as $\arcsin(1/64) = 0.89°$.

What about $\cos(r_y)$ and $\cos(r_x)$? The substitution does not match our derived angle exactly, because $\cos(0.89°) = 0.99988$ and is not exactly 1. However, this value is so close that substituting $\cos(0.89°)$ with 1 is a very good approximation, simplifying calculations significantly.

Another significant simplification results from the division by 64, as the actual division operation can be replaced with a much faster bit shift operation.

This calculation-friendly way of computing rotations is also known as the "CORDIC (COordinate Rotation DIgital Computer)" algorithm.

2.3.4 Minsky Rotation

There is one more interesting mathematical subtlety: Did you notice that expressions of Equations 1 and 2 use a new (primed) pair of variables to store the resulting coordinates, whereas in the implemented Equations 3–6, the value of the first coordinate of a coordinate pair is overwritten with its new value and this value is used in the subsequent calculation of the second coordinate? For example, when the joystick is pushed left, the first call of this subroutine calculates the new value of x according to first expression of Equation 3, overwriting the old value of x. During the second call to calculate z according to the second expression of Equation 3, the new value of x is used instead of the old one. Is this to save the memory needed to temporarily store the old value of x? Is this a bug? If so, why does the rotation calculation actually work?

Have a look at the expressions of Equation 3 (the other Equations 4–6 work in a similar fashion):

$$\begin{aligned} x &:= x + z/64 \\ z &:= -x/64 + z \end{aligned}$$

If we substitute $1/64$ with e, we get

$$\begin{aligned} x &:= x + ez \\ z &:= -ex + z \end{aligned}$$

[5] *This substitution gave a friendly mathematician who happened to see it a nasty shock. She yelled at us that $cos^2x+sin^2x = 1$ for all real x and forever, and therefore this could not possibly be a rotation; it's a rotation with a stretch! We reminded her of the old joke that in wartime the value of the cosine has been known to reach 4. —PML*

Note that x is calculated first and then used in the second expression. When using primed coordinates for the resulting coordinates after calculating the two expressions we get

$$\begin{aligned} x' &:= x + ez \\ z' &:= -ex' + z \\ &= -e(x + ez) + z \\ &= -ex + (1 - e^2)z \end{aligned}$$

or in matrix form

$$\begin{pmatrix} x' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & e \\ -e & 1 - e^2 \end{pmatrix} \begin{pmatrix} x \\ z \end{pmatrix}$$

Surprisingly, this turns out to be a rotation matrix, because its determinant is $(1 \times (1 - e^2) - (-e \times e)) = 1$. (Incidentally, the column vectors of this matrix do not form an orthogonal basis, as their scalar product is $1 \times e + (-e \times (1 - e^2)) = -e^2$. Orthogonality holds for $e = 0$ only.)

This kind of rotation calculation is described by Marvin Minsky in AIM 239 HAKMEM[6] and is called "Minsky Rotation."

2.3.5 Subroutine Implementation Details

To better understand how the implementation of this subroutine works, we must again look at Equations 3–6. If you rearrange the expressions a little, their structure is always of the form:

$$\texttt{TERM1} := \texttt{TERM1 SIGN TERM2}/64$$

or shorter

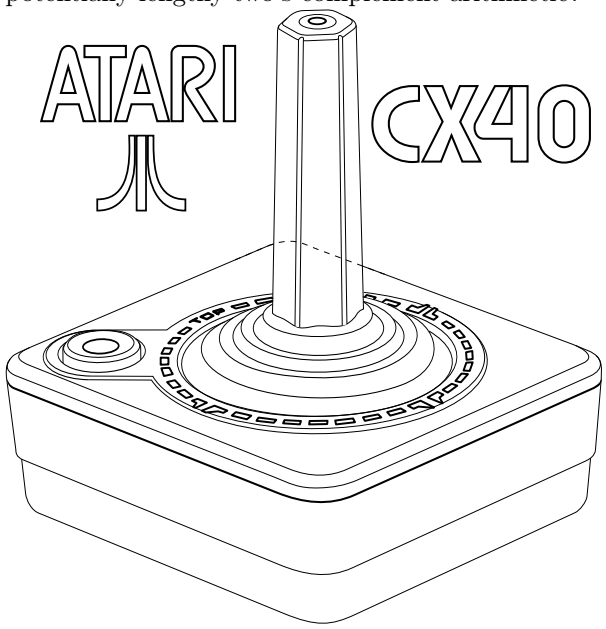
$$\texttt{TERM1} := \texttt{TERM1 SIGN TERM3}$$

where `TERM3` := `TERM2`/64 and `SIGN` := + or − and where `TERM1` and `TERM2` are coordinates. In fact, this is all this subroutine actually does: It simply adds `TERM2` divided by 64 to `TERM1` or subtracts `TERM2` divided by 64 from `TERM1`.

When calling this subroutine the correct table indices for the appropriate coordinates `TERM1` and `TERM2` are passed in the CPU's `Y` and `X` registers, respectively.

What about `SIGN` between `TERM1` and `TERM3`? Again, have a look at Equations 3–6. To compute the two new coordinates after a rotation, the `SIGN` toggles from plus to minus and vice versa. The `SIGN` is initialized with the value of `JOYSTICKDELTA` (`$6D`) before calling subroutine `ROTATE` (`$B69B`, Figure 3) and is toggled in every call of this subroutine. The initial value of `SIGN` should be positive (+, byte value `$01`) if the rotation is clockwise (the joystick is pushed right or up) and negative (−, byte value `$FF`) if the rotation is counter-clockwise (the joystick is pushed left or down), respectively. Because `SIGN` is always toggled in `ROTATE` (`$B69B`) before the adding or subtraction operation of `TERM1` and `TERM3` takes place, you have to pass the already toggled value with the first call.

Unclear still are three instructions starting at address `$B6AD`. They seem to set the two least significant bits of `TERM3` in a random fashion. Could this be some quick hack to avoid messing with exact but potentially lengthy two's-complement arithmetic?



2.4 Dodging Memory Limitations

It is impressing how much functionality was squeezed into **STAR RAIDERS**. Not surprisingly, the bytes of the 8 KB ROM are used up almost completely. Only a single byte is left unused at the very end of the code. When counting four more bytes from three orphaned entries in the game's lookup tables, only five bytes in total out of 8,192 bytes are actually not used. ROM memory was extremely precious. Here are some techniques that demonstrate

[6] `unzip pocorgtfo13.pdf AIM-239.pdf` #Item 149, page 73.

```
                ; INPUT
2               ;
                ; X = Position vector component index of TERM2. Used values are:
4               ; $00..$30 -> z-component (z-coordinate) of position vector 0..48
                ; $31..$61 -> x-component (x-coordinate) of position vector 0..48
6               ; $62..$92 -> y-component (y-coordinate) of position vector 0..48
                ;
8               ; Y = Position vector component index of TERM1. Used values are:
                ; $00..$30 -> z-component (z-coordinate) of position vector 0..48
10              ; $31..$61 -> x-component (x-coordinate) of position vector 0..48
                ; $62..$92 -> y-component (y-coordinate) of position vector 0..48
12              ;
                ; JOYSTICKDELTA ($6D) = Initial value of SIGN. Used values are:
14              ; $01 -> (= Positive) Rotate right or up
                ; $FF -> (= Negative) Rotate left or down
16
                                                ; TERM3 is a 24-bit value, represented by 3 bytes as
18                                              ; $(sign)(high byte)(low byte)
     =006A      L.TERM3LO    = $6A              ; TERM3 (high byte), where TERM3 := TERM2 / 64
20   =006B      L.TERM3HI    = $6B              ; TERM3 (low byte),  where TERM3 := TERM2 / 64
     =006C      L.TERM3SIGN = $6C               ; TERM3 (sign),      where TERM3 := TERM2 / 64
22
 B69B BDAD09    ROTATE       LDA ZPOSSIGN,X     ;
24 B69E 4901                 EOR #$01           ;
 B6A0 F002                   BEQ SKIP224        ; Skip if sign of TERM2 is positive
26 B6A2 A9FF                 LDA #$FF           ;

28 B6A4 856B    SKIP224      STA L.TERM3HI      ; If TERM2 pos. -> TERM3 := $0000xx (= TERM2 / 256)
 B6A6 856C                   STA L.TERM3SIGN    ; If TERM2 neg. -> TERM3 := $FFFFxx (= TERM2 / 256)
30 B6A8 BD400A               LDA ZPOSHI,X       ; where xx := TERM2 (high byte)
 B6AB 856A                   STA L.TERM3LO      ;
32
 B6AD AD0AD2                 LDA RANDOM         ; (?) Hack to avoid messing with two-complement's
34 B6B0 09BF                 ORA #$BF           ; (?) arithmetic? Provides two least significant
 B6B2 5DD30A                 EOR ZPOSLO,X       ; (?) bits B1..0 in TERM3.
36
 B6B5 0A                     ASL A              ; TERM3 := TERM3 * 4 (= TERM2 / 256 * 4 = TERM2 / 64)
38 B6B6 266A                 ROL L.TERM3LO      ;
 B6B8 266B                   ROL L.TERM3HI      ;
40 B6BA 0A                   ASL A              ;
 B6BB 266A                   ROL L.TERM3LO      ;
42 B6BD 266B                 ROL L.TERM3HI      ;

44 B6BF A56D                 LDA JOYSTICKDELTA  ; Toggle SIGN for next call of ROTATE
 B6C1 49FF                   EOR #$FF           ;
46 B6C3 856D                 STA JOYSTICKDELTA  ;
 B6C5 301A                   BMI SKIP225        ; If SIGN negative then subtract, else add TERM3
48
                ;*** Addition ********************************************************************
50 B6C7 18                   CLC                ; TERM1 := TERM1 + TERM3
 B6C8 B9D30A                 LDA ZPOSLO,Y       ; (24-bit addition)
52 B6CB 656A                 ADC L.TERM3LO      ;
 B6CD 99D30A                 STA ZPOSLO,Y       ;
54
 B6D0 B9400A                 LDA ZPOSHI,Y       ;
56 B6D3 656B                 ADC L.TERM3HI      ;
 B6D5 99400A                 STA ZPOSHI,Y       ;
58
 B6D8 B9AD09                 LDA ZPOSSIGN,Y     ;
60 B6DB 656C                 ADC L.TERM3SIGN    ;
 B6DD 99AD09                 STA ZPOSSIGN,Y     ;
62 B6E0 60                   RTS                ;

64              ;*** Subtraction *****************************************************************
 B6E1 38        SKIP225      SEC                ; TERM1 := TERM1 - TERM3
66 B6E2 B9D30A               LDA ZPOSLO,Y       ; (24-bit subtraction)
 B6E5 E56A                   SBC L.TERM3LO      ;
68 B6E7 99D30A               STA ZPOSLO,Y       ;

70 B6EA B9400A               LDA ZPOSHI,Y       ;
 B6ED E56B                   SBC L.TERM3HI      ;
72 B6EF 99400A               STA ZPOSHI,Y       ;

74 B6F2 B9AD09               LDA ZPOSSIGN,Y     ;
 B6F5 E56C                   SBC L.TERM3SIGN    ;
76 B6F7 99AD09               STA ZPOSSIGN,Y     ;
 B6FA 60                     RTS                ;
```

Figure 3. `ROTATE` Subroutine at `$B69B`

the fierce fight for each spare ROM byte.

2.4.1 Loop Jamming

Loop jamming is the technique of combining two loops into one, reusing the loop index and optionally skipping operations of one loop when the loop index overshoots.

How much bytes are saved by loop jamming? As an example, Figure 4 shows an original 19-byte fragment of subroutine `INITIALIZE` (`$B3BA`) using loop jamming. The same fragment without loop jamming, shown in Figure 5, is 20 bytes long. So loop jamming saved one single byte.

Another example is the loop that is set up at `$A165` in `INITCOLD` (`$A14A`). A third example is the loop set up at `$B413` in `INITIALIZE` (`$B3BA`). This loop does not explicitly skip loop indices, thus saving four more bytes (the `CMP` and `BCS` instructions) on top of the one byte saved by regular loop jamming. Thus, seven bytes are saved in total by loop jamming.

2.4.2 Sharing Blank Characters

One more technique to save bytes is to let strings share their leading and trailing blank characters. In the game there is a header text line of twenty characters that displays one of the strings "`LONG RANGE SCAN`," "`AFT VIEW`," or "`GALACTIC CHART`." The display hardware directly points to their location in the ROM. They are enclosed in blank characters (bytes of value `$00`) so that they appear horizontally centered.

A naive implementation would use $3 \times 20 = 60$ bytes to store these strings in ROM. In the actual implementation, however, the trailing blanks of one header string are reused as leading blanks of the following header, as shown in Figure 6. By sharing blank characters the required memory is reduced from 60 bytes to 54 bytes, saving six bytes.

2.4.3 Reusing Interrupt Exit Code

Yet another, rather traditional technique is to reuse code, of course. Figure 7 shows the exit code of the Vertical Blank Interrupt handler `VBIHNDLR` (`$A6D1`) at `$A715`, which jumps into the exit code of the Display List Interrupt handler `DLSTHNDLR` (`$A718`) at `$A74B`, reusing the code that restores the registers that were put on the CPU stack before entering the Vertical Blank Interrupt handler.

This saves another six bytes (`PLA`, `TAY`, `PLA`, `TAX`, `PLA`, `RTI`), but spends three bytes (`JMP JUMP004`), in total saving three bytes.

2.5 Bugs

There are a few bugs, or let's call them glitches, in **STAR RAIDERS**. This is quite astonishing, given the complex game and the development tools of 1979, and is a testament to thorough play testing. The interesting thing is that the often intense game play distracts the players' attention from noticing these glitches, just like what a skilled parlor magician would do.

2.5.1 A Starbase Without Wings

When a starbase reaches the lower edge of the graphics screen and overlaps with the Control Panel Display below (Figure 8 (left), screenshot) and you nudge the starbase a little bit more downward, its wings suddenly vanish (Figure 8 (right), screenshot).

The reason is shown in the insert on the right side of the figure: The starbase is a composite of three Players (sprites). Their bounding boxes are indicated by three white rectangles. If the vertical position of the top border of a Player is larger than a vertical position limit, indicated by the tip of the white arrow, the Player is not displayed. The relevant location of the comparison is at `$A534` in `GAMELOOP` (`$A1F3`). While the Player of the central part of the starbase does not exceed this vertical limit, the Players that form the starbase's wings do so, and are thus not rendered.

This glitch is rarely noticed because players do their best to keep the starbase centered on the screen, a prerequisite for a successful docking.

2.5.2 Shuffling Priorities

There are two glitches that are almost impossible to notice (and I admit some twisted kind of pleasure to expose them, ;-):

- During regular gameplay, the Zylon ships and the photon torpedoes appear *in front of* the cross hairs (Figure 9 (left)), as if the cross hairs were light years away.
- During docking, the starbase not only appears *behind* the stars (Figure 9 (right)) as if the starbase is light years away, but the transfer vessel moves *in front of* the cross hairs!

```
1 B3BA A259      INITIALIZE      LDX #89                 ; Set 89(+1) GRAPHICS7 rows from DSPLST+5 on
  B3BC A90D      LOOP060         LDA #$0D                ; Prep DL instruction $0D (one row of GRAPHICS7)
3 B3BE 9D8502                    STA DSPLST+5,X          ; DSPLST+5,X := one row of GRAPHICS7
  B3C1 E00A                      CPX #10                 ;
5 B3C3 B005                      BCS SKIP195             ;
  B3C5 BDA9BF                    LDA PFCOLORTAB,X        ; Copy PLAYFIELD color table to zero-page table
7 B3C8 95F2                      STA PF0COLOR,X          ; (loop jamming)
  B3CA CA        SKIP195         DEX                     ;
9 B3CB 10EF                      BPL LOOP060             ;
```

Figure 4. `INITIALIZE` Subroutine at `$B3BA` (Excerpt)

```
1 B3BA A259      INITIALIZE      LDX #89                 ; Set 89(+1) GRAPHICS7 rows from DSPLST+5 on
  B3BC A90D      LOOP060         LDA #$0D                ; Prep DL instruction $0D (one row of GRAPHICS7)
3 B3BE 9D8502                    STA DSPLST+5,X          ; DSPLST+5,X := one row of GRAPHICS7
  B3C1 CA                        DEX                     ;
5 B3C2 10F8                      BPL LOOP060             ;
  B3C4 A209                      LDX #9                  ;
7 B3C6 BDAABF    LOOP060B        LDA PFCOLORTAB,X        ; Copy PLAYFIELD color table to zero-page table
  B3C9 95F2                      STA PF0COLOR,X          ;
9 B3CB CA                        DEX                     ;
  B3CC 10F8                      BPL LOOP060B            ;
```

Figure 5. `INITIALIZE` Subroutine Without Loop Jamming (Excerpt)

The reason is the drawing order or "graphics priority" of the bit-mapped graphics and the Players (sprites). It is controlled by the `PRIOR` (`$D01B`) hardware register.

During regular flight, see Figure 9 (left), `PRIOR` (`$D01B`) has a value of `$11`. This arranges the displayed elements in the following order, from front to back:

- Players 0-4 (photon torpedoes, Zylon ships, ...)
- Bit-mapped graphics (stars, cross hairs)
- Background.

This arrangement is fine for the stars as they are bit-mapped graphics and need to appear behind the photon torpedoes and the Zylon ships, but this arrangement applies also to the cross hairs – causing the glitch.

During docking, see Figure 9 (right), `PRIOR` (`$D01B`) has a value of `$14`. This arranges the displayed elements the following order, from front to back:

- Player 4 (transfer vessel)
- Bit-mapped graphics (stars, cross hairs)
- Players 0-3 (starbase, ...)
- Background.

This time the arrangement is fine for the cross hairs as they are bit-mapped graphics and need to appear in front of the starbase, but this arrangement also applies to the stars. In addition, the Player of the white transfer vessel correctly appears in front of the bit-mapped stars, but also in front of the bit-mapped cross hairs.

Fixing these glitches is hardly possible, as the display hardware does not allow for a finer control of graphics priorities for individual Players.

2.6 A Mysterious Finding

A simple instruction at location `$A175` contained the most mysterious finding in the game's code. The disassembler reported the following instruction, which is equivalent to `STA $0067,X`. (`ISVBISYNC` has a value of `$67`.)

```
A175 9D6700     STA ISVBISYNC,X
```

The object code assembled from this instruction is unusual as its address operand was assembled as a 16-bit address and not as an 8-bit zero-page address. Standard 6502 assemblers would always generate shorter object code, producing `9567` (`STA $67,X`) instead of `9D6700` and saving a byte.

In my reverse engineered source code, the only way to reproduce the original object code was the following:

```
                ;*** Header text of Long-Range Scan view (shares spaces with following header) *
A0F8 00006C6F LRSHEADER       .BYTE $00,$00,$6C,$6F,$6E,$67,$00,$72 ; ''  LONG RANGE SCAN''
A0FC 6E670072
A100 616E6765                 .BYTE $61,$6E,$67,$65,$00,$73,$63,$61
A104 00736361
A108 6E                       .BYTE $6E

                ;*** Header text of Aft view (shares spaces with following header) *************
A109 00000000 AFTHEADER       .BYTE $00,$00,$00,$00,$00,$00,$61,$66 ; ''      AFT VIEW    ''
A10D 00006166
A111 74007669                 .BYTE $74,$00,$76,$69,$65,$77,$00,$00
A115 65770000
A119 00                       .BYTE $00

                ;*** Header text of Galactic Chart view *************************************
A11A 00000067 GCHEADER        .BYTE $00,$00,$00,$67,$61,$6C,$61,$63 ; ''   GALACTIC CHART   ''
A11E 616C6163
A122 74696300                 .BYTE $74,$69,$63,$00,$63,$68,$61,$72
A126 63686172
A12A 74000000                 .BYTE $74,$00,$00,$00
```

Figure 6. Header Texts at `$A0F8`

```
A6D1 A9FF     VBIHNDLR        LDA #$FF                ; Start of Vertical Blank Interrupt handler
                ...
A715 4C4BA7   SKIP046         JMP JUMP004             ; End of Vertical Blank Interrupt handler
                ...
A718 48       DLSTHNDLR       PHA                     ; Start of Display List Interrupt handler
                ...
A74B 68       JUMP004         PLA                     ; Restore registers
A74C A8                       TAY                     ;
A74D 68                       PLA                     ;
A74E AA                       TAX                     ;
A74F 68                       PLA                     ;
A750 40                       RTI                     ; End of Display List Interrupt Handler
```

Figure 7. `VBIHNDLR` and `DLSTHNDLR` Handlers Share Exit Code

```
; HACK: Fake STA ISVBISYNC,X with 16b addr
A175 9D         .BYTE $9D
A176 6700       .WORD ISVBISYNC
```

I speculated for a long time whether this strange assembler output indicated that the object code of the original ROM cartridge was produced with a non-standard 6502 assembler. I have heard that Atari's in-house development systems ran on PDP-11 hardware. Luckily, the month after I finished my reverse engineering effort, the original **STAR RAIDERS** source code re-surfaced.[7] To my astonishment it uses exactly the same "hack" to reproduce the three-byte form of the `STA ISVBISYNC,X` instruction:

```
A175 9D         .BYTE $9D    ; STA ABS,X
A176 67 00      .WORD PAGE0 ; STA PAGE0,X (ABSOLUTE)
```

Unfortunately the comments do not give a clue why this pattern was chosen. After quite some time it made click: The instruction `STA ISVBISYNC,X` is used in a loop which iterates the CPU's X register from 0 to 255 to clear memory. By using this instruction with a 16-bit address ("indexed" mode operand) memory from `$0067` to `$0166` is cleared. Had the code been using the same operation with an 8-bit address ("indexed, zero-page" mode operand), memory from `$0067` to `$00FF` would have been cleared, then the indexed address would have wrapped back to `$0000` clearing memory `$0000` to `$0066`, effectively overwriting already initialized memory locations.

2.7 Documenting Star Raiders

Right from the start of reverse engineering **STAR RAIDERS** I not only wanted to understand how the game worked, but I also wanted to document the result of my effort. But what would be an appropriate form?

First, I combined the emerging memory map file with the fledgling assembly language source code in

[7] `https://archive.org/details/AtariStarRaidersSourceCode`
`unzip pocorgtfo13.pdf StarRaidersOrig.pdf`

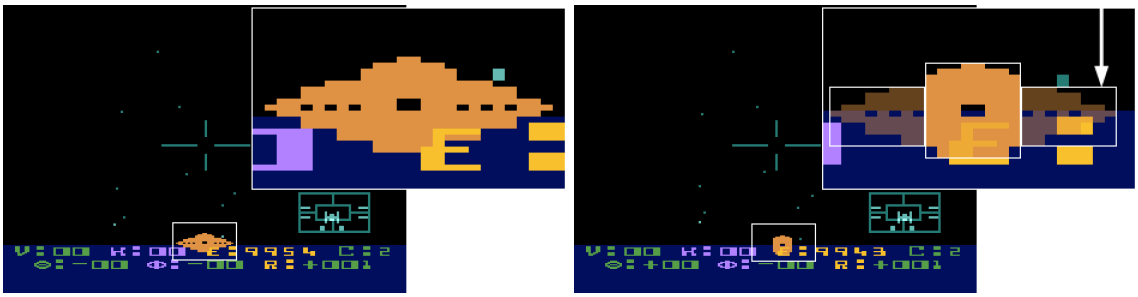


Figure 8. A Starbase's Wings Vanish

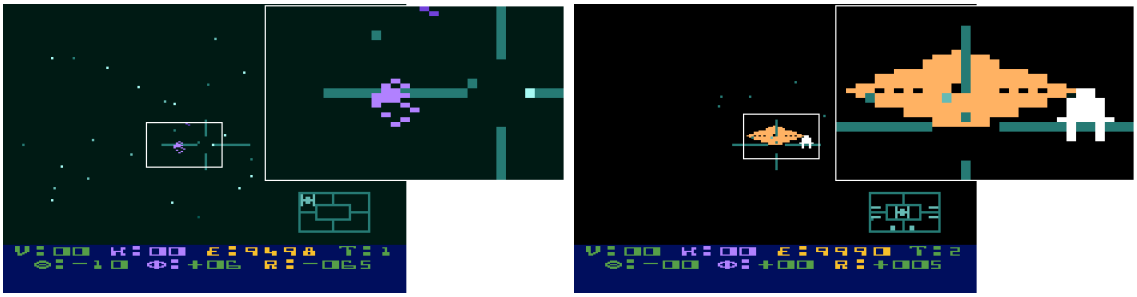


Figure 9. Photon torpedo in front of cross hairs and a starbase behind the stars!

order to work with just one file. Then, I switched the source code format to that of MAC/65, a well-known and powerful macro assembler for the Atari 8-bit Home Computer System. I also planned, at some then distant point in the future, to assemble the finished source code with this assembler on an 8-bit Atari.

Another major influence on the emerging documentation was the Atari BASIC Source Book, which I came across by accident[8]. It reproduced the complete, commented assembly language source code of the 8 KB Atari BASIC interpreter cartridge, a truly non-trivial piece of software. But what was more: The source code was accompanied by several chapters of text that explained in increasing detail its concepts and architecture, that is, how Atari BASIC actually worked. Deeply impressed, I decided on the spot that my reverse engineered **STAR RAIDERS** source code should be documented at the same level of detail.

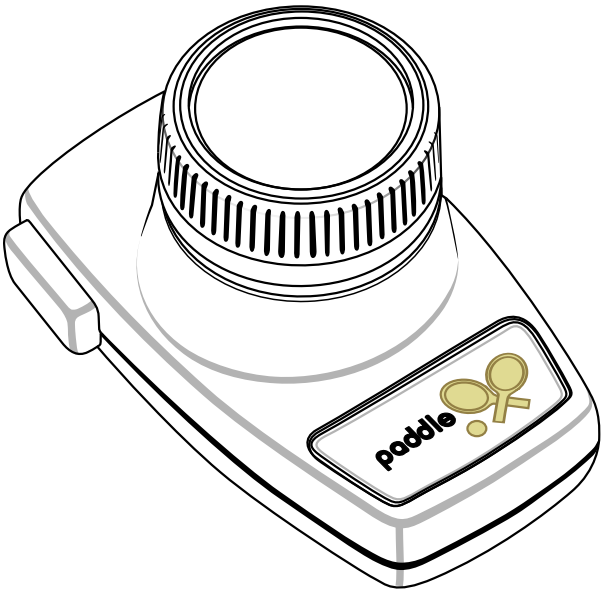
The overall documentation structure for the source code, which I ended up with was fourfold: On the lowest level, end-of-line comments documented the functionality of individual instructions. On the next level, line comments explained groups of instructions. One level higher still, comments composed of several paragraphs introduced each subroutine. These paragraphs provided a summary of the subroutine's implementation and a description of all input and output parameters, including the valid value ranges, if possible. On the highest level, I added the memory map to the source code as a handy reference. I also planned to add some chapters on the game's general concepts and overall architecture, just like the Atari BASIC Source Book had done. Unfortunately, I had to drop that idea due to lack of time. I also felt that the detailed subroutine documentation was quite sufficient. However, I did add sections on the 3D coordinate system and the position and velocity vectors to the source code as a tip of the hat to the Atari BASIC Source Book.

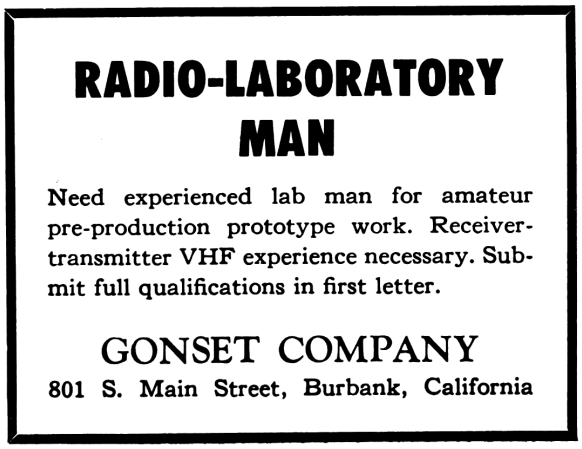
After I was well into reverse engineering **STAR RAIDERS**, slowly adding bits and pieces of information to the raw disassembly of the **STAR RAIDERS** ROM and fleshing out the ever growing documentation, I started to struggle with establishing a consistent and uniform terminology for the documentation (Is it "asteroid," "meteorite," or "meteor"? "Explosion bits," "explosion debris," or "explosion fragments"? "Gun sights" or "cross hairs"?) A look into the **STAR RAIDERS** instruction manual clarified only

[8]The Atari BASIC Source Book by Wilkinson, O'Brien, and Laughton. A COMPUTE! publication.

a painfully small amount of cases. Incidentally, it also contradicted itself as it called the enemies "Cylons" while the game called them "Zylons," such as in the message "`SHIP DESTROYED BY ZYLON FIRE`."

But I was not only after uniform documentation, I also wanted to unify the symbol names of the source code. For example, I had created a hodge-podge of color-related symbol names, which contained fragments such as "`COL`," "`CLR`," "`COLR`," and "`COLOR`." To make matters worse, color-related symbol names containing "`COL`" could be confused with symbol names related to (pixel) columns. The same occurred with symbol names related to Players (sprites), which contained fragments such as "`PL`," "`PLY`," "`PLYR`," "`PLAY`," and "`PLAYER`," or with symbol names of lookup tables, which ended in "`TB`," "`TBL`," "`TAB`," and "`TABLE`," and so on. In addition to inventing uniform symbol names I also did not want to exceed a self-imposed symbol name limit of 15 characters. So I refactored the source code with the search-and-replace functionality of the text editor over and over again.





I noticed that I spent more and more time on refactoring the documentation and the symbol names and less time on adding actual content. In addition, the actual formatting of the emerging documented source code had to be re-adjusted after every refactoring step. Handling the source code became very unwieldy. And worst of all: How could I be sure that the source code still represented the exact binary image of the ROM cartridge?

The solution I found to this problem eventually was to create an automated build pipeline, which dealt with the monotonous chores of formatting and assembling the source code, as well as comparing the produced ROM cartridge image with a reference image. This freed time for me to concentrate on the actual source code content. Yet another incarnation of "separation of form and content," the automated build pipeline was always a pleasure to watch working its magic. (Mental note: I should have created this pipeline much earlier in the reverse engineering effort.) These are the steps of the automated build pipeline:

1. The pipeline starts with a raw, documented assembly language source code file. It is already roughly formatted and uses a little proprietary markup, just enough to mark up sections of meta-comments that are to be removed in the output as well as subroutine documentation containing multiple paragraphs, numbered, and unnumbered lists. This source code file is fed to a pre-formatter program, which I implemented in Java. The pre-formatter removes the meta-comments. It also formats the entries of the memory map and the subroutine documentation by wrapping multi-line text at a preset right margin, out- and indenting list items, numbering lists, and vertically aligning parameter descriptions. It also corrects the number of trailing asterisks in line comments, and adjusts the number of asterisks of the box headers that introduce subroutine comments, centering their text content inside the asterisk boxes.

2. The output of the pre-formatter from step 1 is fed into an Atari 6502 assembler, which I also wrote in Java. It is available as open-source on GitHub.[9] Why write an Atari 6502 assembler? There are other 6502 assemblers readily available, but not all produce object code for the Atari 8-bit Home Computer System, not all use the MAC/65 source code format, and not all of them can be easily tweaked when necessary. The output of this step is both an assembler output listing and an object file.

3. The assembler output listing from step 2 is the finished, formatted, reverse engineered **STAR RAIDERS** source code, containing the documentation, the source code, and the object code listing.

4. The assembler output listing from step 2 is fed into a symbol checker program, which I again wrote in Java. It searches the documentation parts of the assembler output listing and checks if every symbol, such as "`GAMELOOP`," is followed by its correct hex value, "(`$A1F3`)." It reports any symbol with missing or incorrect hex values. This ensures further consistency of the documentation.

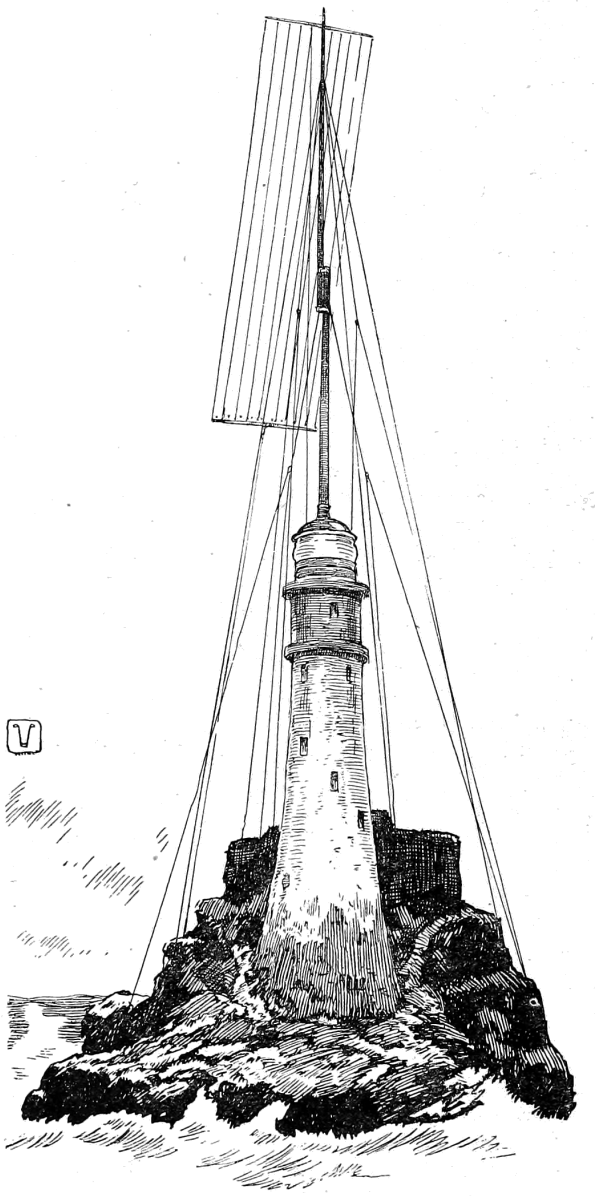
5. The object file of step 2 is converted by yet another program I wrote in Java from the Atari executable format into the final Atari ROM cartridge format.

6. The output from step 5 is compared with a reference binary image of the original **STAR RAIDERS** 8 KB ROM cartridge. If both images are the same, then the entire build was successful: The raw assembly language source code really represents the exact image of the **STAR RAIDERS** 8 KB ROM cartridge

[9] `git clone https://github.com/lwiest/Atari6502Assembler`
`unzip pocorgtfo13.pdf Atari6502Assembler.zip`

Typical build times on my not-so-recent Windows XP box (512 MB) were 15 seconds.

For some finishing touches, I ran a spell-checker over the documented assembly language source code file from time to time, which also helped to improve documentation quality.



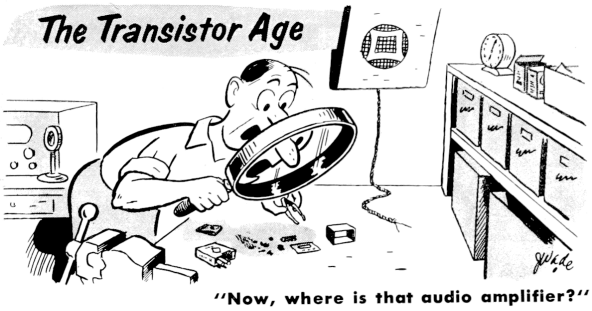
FASTNET LIGHT AS IT WOULD APPEAR IF CONVERTED INTO A "BLIND LIGHTHOUSE."

2.8 Conclusion

After quite some time, I achieved my goal to create a reverse engineered, complete, and fully documented assembly language source code of **STAR RAIDERS**. For final verification, I successfully assembled it with MAC/65 on an Atari 800 XL with 64 KB RAM (emulated with Atari800Win Plus). MAC/65 is able to assemble source code larger than the available RAM by reading the source code as several chained files. So I split the source code (560 KB) into chunks of 32 KB and simply had the emulator point to a hard disk folder containing these files. The resulting assembler output listing and the object file were written back to the same hard disk folder. The object file, after being transformed into the Atari cartridge format, exactly reproduced the original **STAR RAIDERS** 8 KB ROM cartridge.

2.9 Postscript

I finished my reverse engineering effort in September 2015. I was absolutely thrilled to learn that in October 2015 scans of the original **STAR RAIDERS** source code re-surfaced. To my delight, inspection of the original source code confirmed the findings of my reverse engineered version and caused only a few trivial corrections. Even more, the documentation of my reverse engineered version added a substantial amount of information – from overall theory of operation down to some tricky details – to the understanding of the often sparsely commented original (quite expected for source code never meant for publication).



"Now, where is that audio amplifier?"

Cut Here if Printing on A4

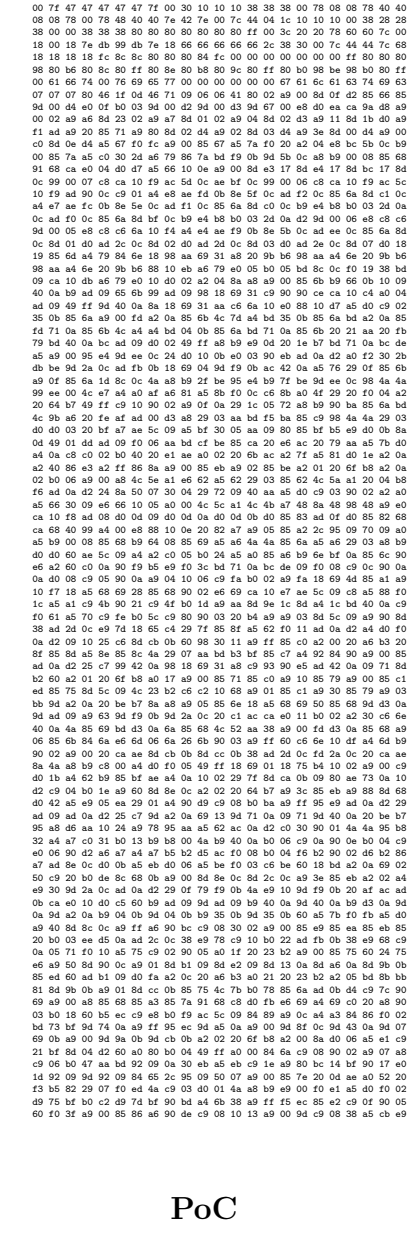
Cut Here if Printing on A4

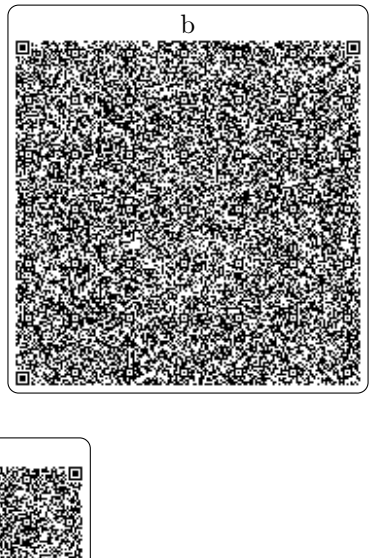
1

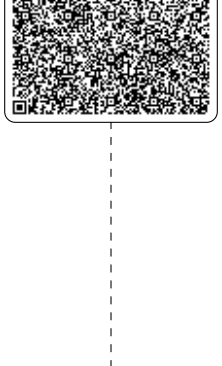
2

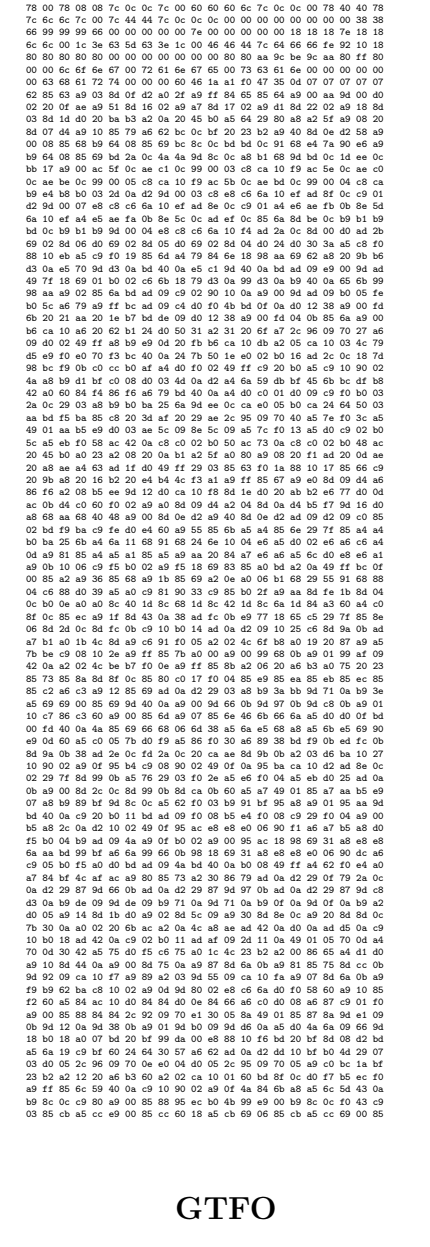
3

Самиздат









PoC

GTFO

COMPUTER ADVENTURE

STAR RAIDERS

ATARI 400/800

Manhattan Punch Line Theatre

Steve Kaplan Mitch McGuire Richard Erickson Jerry Heymann

Producing Directors

presents

a new play

by

Mike Eisenberg

HACKERS

Directed by Jerry Heymann

3 How Slow Can You Go?

by James Forshaw

While doing my research into Windows, I tend to find quite a few race condition vulnerabilities. Although these vulnerabilities can be exploited, you typically only get a tiny window of time in which to do it. A fairly typical sequence of actions looks something like this:

1. Do some security check.
2. Access some resource.
3. Perform secure action.

In this case the race condition is between the security check and the action. If we can modify the state of the system in between those actions, it might be possible to elevate privileges or do unexpected things. The time window is typically very small, but if the code is accessing some controllable resource in between the check and the action, we might still be able to create a very reliable exploit.

I wanted to find a way of increasing the time window to win the race in cases where the code accesses a resource we control. The following is an overview of the thought process I went through to come up with a working solution.

3.1 Investigating Object Manager Lookup Performance

Hidden under the hood of Windows NT is the Object Manager Namespace (OMN). You wouldn't typically interact with it directly as the Win32 API for the most part hides it away. The NT kernel defines a set of objects, such as Files, Events, Registry Keys, that can all have a name associated with them. The OMN provides the means to lookup these named objects. It acts like a file system; for example, you can specify a path to an NT system call such as `\BaseNamedObjects\MyEvent`, and an event can be thus looked up.

There are two special object types for use in the OMN: Object Directories and Symbolic Links. Object Directories act as named containers for other objects, whereas Symbolic Links allow a name to be redirected to another OMN path. Symbolic Links are used quite a lot; for example, the Windows drive letters are really symbolic links to the real storage device. When we call an NT system call, the kernel must lookup the entire path, following any symbolic links until it either reaches the named object or fails to find a match.

In this exploit we want to make the process of looking up a resource we control as slow as possible. For example, if we could make it take 1 or 2 seconds, then we've got a massive window of opportunity to win the race condition. Therefore I want to find a way of manipulating the Object Manager lookup process in such a way that we achieve this goal. I am going to present my approach to achieving the required result.

A note about my setup: for my testing I am going to open a named Event object. All testing is done on my 2.8GHz Xeon Workstation. Although it has 20 physical cores, the lookup process won't be parallelized, and therefore that shouldn't be an issue. Xeons tend to have more L2/L3 cache than consumer processors, but if anything this should only make our timings faster. If I can get a long lookup time on my Workstation, it should be possible on pretty much anything else running Windows. Finally, this is all tested on an up-to-date Windows 10; however, not much has changed since Windows 7 that might affect the results.

First let's just measure the time it takes to do

a normal lookup. We'll repeat the lookup a 1,000 times and take the average. The results are probably what we'd expect: the lookup process for a simple named Event is roughly 3μs. That includes the system call transition, lookup process, and the access check on the Event object. Although in theory you could win a race, it seems pretty unlikely, even on a multi-core processor. So let's think about a way of improving the lookup time (and when I say "improve", I mean making the lookup time slower).

An Object Manager path is limited to the maximum string size afforded by the UNICODE_STRING structure.

```
struct UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
}
```

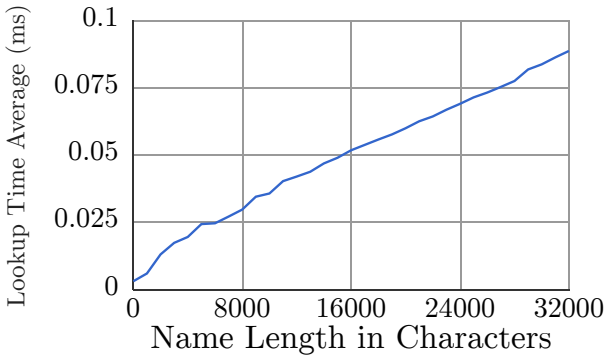
We can see that the Length member is an unsigned 16 bit integer, limiting the maximum length to $2^{16} - 1$. This, however, is a byte count, so in fact this limits us to $2^{15} - 1$ or 32767 characters. From this result, there are two obvious possible approaches we can take:

1. Make a path that contains one very long name. The lookup process would have to compare the entire name using a typical string comparison operation to verify it's accessing the correct object. This should take linear time relative to the length of the string.

2. Make multiple small named directories and repeat. E.g., `\A\A\A\A\...\EventName`. The assumption here is that each lookup takes a fixed amount of time to complete. The operation will again be linear time relative to the depth of recursion of the directories.

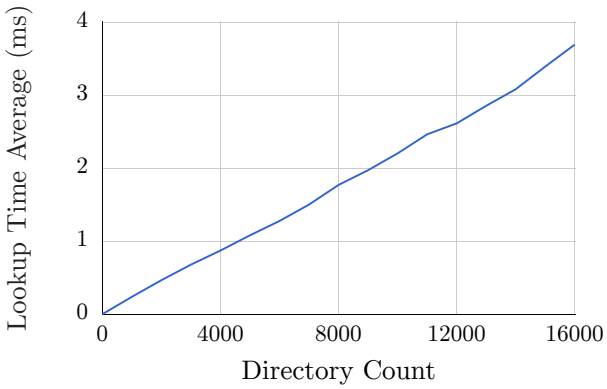
Now it would seem likely that the cost of the entire operation of a single lookup will be worse than a string comparison, a primitive that is typically optimized quite heavily. At this point we have not had to look at any actual kernel code, and we won't start quite yet, so instead empirical testing seems the way to go.

Let's start with the first approach, making a long string and performing a lookup on it. Our name limit is around 32767, although we'll need to be able to make the object in a writable directory such as `\BaseNamedObject`, which reduces the length slightly, but not enough to make significant impact. Therefore, we'll perform the Event opening on names between 1 character and 32,000 characters in length. The results are shown below:



Although this is a little noisy, our assumption of a linear lookup time seems correct. The longer the string, the longer it takes to look it up. For a 32,000 character long string, this seems to top out at roughly 90μs – still not enough in my opinion for a useful primitive, but certainly a start.

Now let's instead look at the recursive directory approach. In this case the upper bound is around 16,000 directories. This is because each path component must contain a backslash and a single character name (i.e. `\A\A\A...`). Therefore our maximum path limit is halved. Of course we'd make the assumption that the time to go through the lookup process is going to be greater than the time it takes to compare 4 Unicode characters, but let's test to make sure. The results are shown below:



Well, I think that's unequivocal. For 16,000 recursive depth, the average lookup time is around 3700μs, or around 40 times larger than the long path name lookup result. Now, of course, this comes with downsides. For a start, you need to create 16,000 or so directory objects in the kernel. At least on a mod-

ern 64 bit Windows this isn't likely to be too taxing, however it's still worth bearing in mind. Also the process must maintain a handle to each of those directories, because otherwise they'd be deleted (as a normal user cannot make kernel objects permanent). Fortunately our handle limit for a single process is of the order of 16 million, so we're a couple of orders of magnitude below the limit of that.

Now, is 3700μs going to be enough for us? Maybe, it's certainly orders of magnitude greater than 3μs. But can we do better? We've now run out of path space, we've filled the absolute maximum allowed string length with recursive directory names. What we could do with is a method of multiplying that effect without requiring a longer path. We can do this by using Object Manager symbolic links. By placing the symbolic link as the last component of the long path we can force the kernel to reparse, and start the lookup all over again. On the final lookup we'll just point the symbolic link to the target.

Ultimately though we can only do this 64 times. Why, can't we do this indefinitely? Well, no—for a fairly obvious reason: each time a symbolic link is encountered the kernel restarts the parsing processes; if you pointed a symbolic link at itself, you'd end up in an infinite loop. The reparse limit of 64 prevents that from becoming a problem. The results are as we expected, the time taken to lookup our event is proportional to both the number of symbolic links and the number of recursive directories. For 64 symbolic links and 16,000 directories it takes approximately 200ms (note I've had to change the order of the result now to milliseconds). At around $\frac{1}{5}$ of a second that should be enough, right? Sure, but I'm greedy; I want more. How can we make the lookup time even worse?

At this point it's time to break out the disassembler and see how the lookup process works under the hood in the kernel. First off, let's see what an object directory structure looks like. We can dump it from a kernel debugging session using WinDBG with the

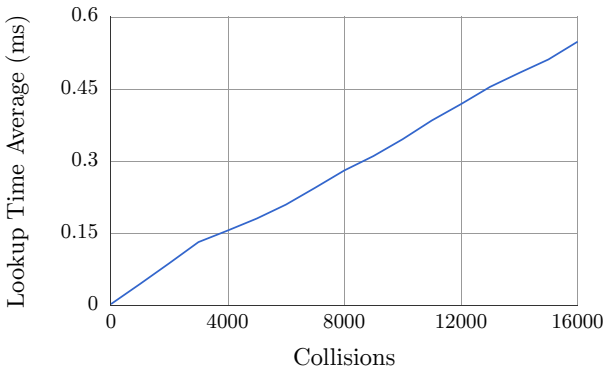
command `dt nt!_OBJECT_DIRECTORY`. Converted back to a C-style structure, it looks something like the following:

```
struct OBJECT_DIRECTORY
{
    POBJECT_DIRECTORY_ENTRY HashBuckets[37];
    EX_PUSH_LOCK Lock;
    PDEVICE_MAP DeviceMap;
    ULONG SessionId;
    PVOID NamespaceEntry;
    ULONG Flags;
    POBJECT_DIRECTORY ShadowDirectory;
}
```

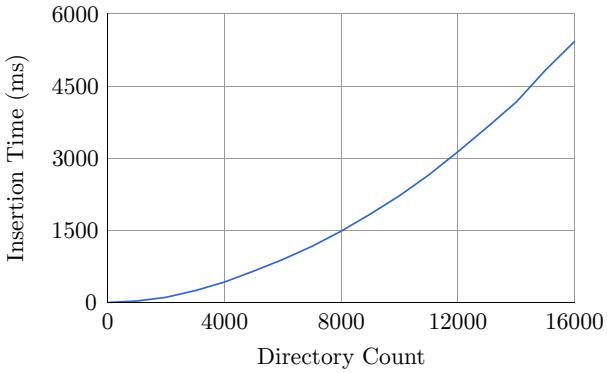
Based on the presence of the HashBucket field, it's safe to assume that the kernel is using a hash table to store directory entries. This makes some sense, because if the kernel just maintained a list of directory entries, this would be pretty poor for performance. With a hash table the lookup time is much reduced as long as the hashing algorithm does a good job of reducing collisions. This is only the case though if the algorithm isn't being actively exploited. As we're trying to increase the cost of lookups, we can intentionally add entries with collisions to make the lookup process take the worst case time, which is linear relative to the number of entries in a directory. This again provides us with another scaling factor, and in this case the number of entries is only going to be limited by available memory, as we are never going to need to put the name into the path.

So what's the algorithm for the hash? The main function of interest is `ObpLookupObjectName`, which is referenced by functions such as `ObReferenceObjectByName`. The directory entry logic is buried somewhere in this large function; however, fortunately there's a helper function `ObpLookupDirectoryEntryEx`, which has the same logic (it isn't actually called by `ObpLookupObjectName`, but it doesn't matter) that is smaller and easier to reverse (Figure 10).

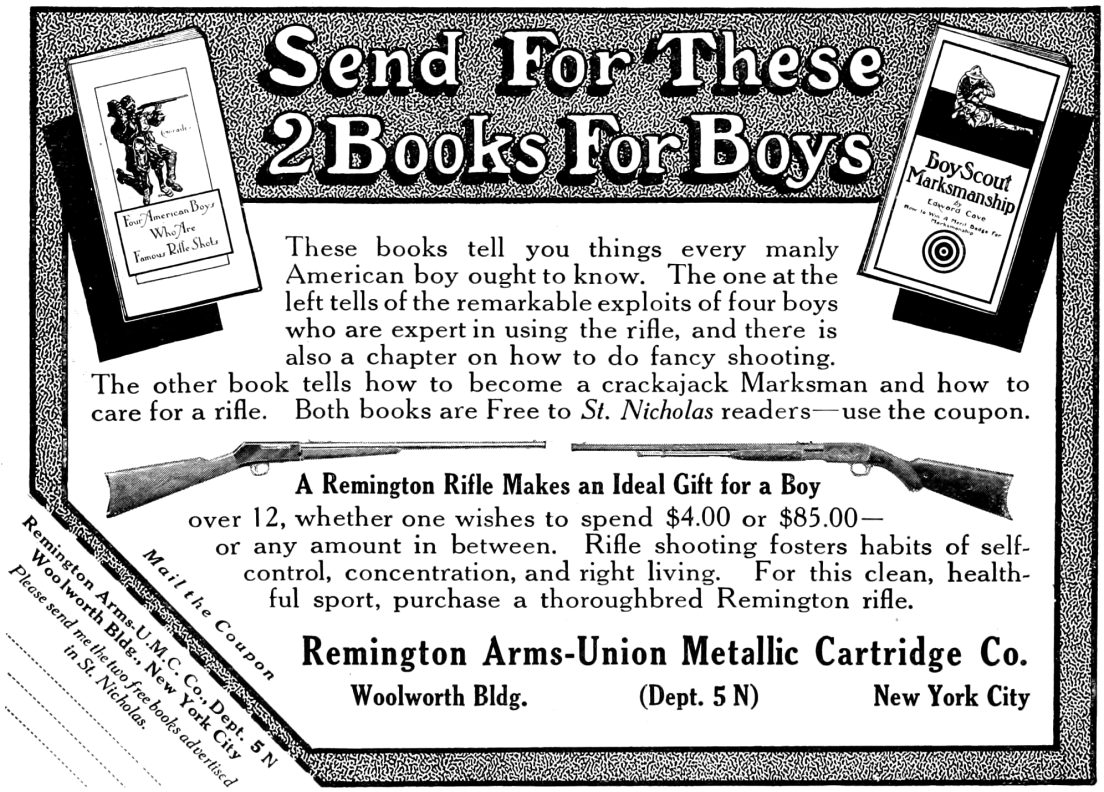
So the hashing algorithm is pretty simple; it repeatedly mixes the bits of the current hash value and then adds the uppercase Unicode character to the hash. We could work out a clever way of getting hash collisions from this, but actually it's pretty simple. The object manager allows us to specify names containing NULL characters, therefore if we take our target name, say 'A', and prefix it with increasing length strings containing only NULL, we get both Hash and Bucket collisions. This does limit us to creating only 32,000 or so colliding entries before we run out of strings to create them, but, as we'll see in a minute, that's not a problem. Let's look at the results of doing this for a single directory:



Yet again, a nice linear graph. For a given collision count it's nowhere near as good as the recursive directory approach, but it is a multiplicative factor in the lookup time, which we can abuse. So you'd think we can now easily apply this to all our 16,000 recursive directories, add in symbolic links, and probably get an insane lookup time. Yes, we would, however there's a problem, insertion time. Every time we add a new entry to a directory, the kernel must do a lookup to check that the entry doesn't already exist. This means that, for every entry we add, we must do $(n-1)^2$ checks in the hash bucket just to find that we don't have the entry before we insert it. This means that the time to add a new entry is approximately proportional to the square of the number of entries. Sure it's not a cubic or exponential increase, but that's hardly a consolation. To prove that this is the case we can just measure the insertion time:



That graph shows a pretty clear n^2 trend for the insertion time. If, say, we wanted to create a directory entry with 16,000 collisions, it takes close to 5.5 seconds. If we wanted to then do that for all 16,000



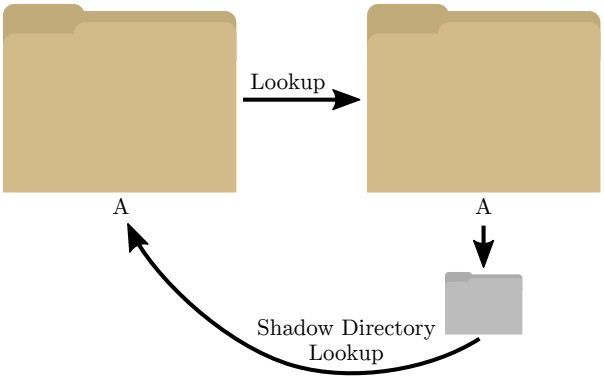

```
POBJECT_DIRECTORY ObpLookupDirectoryEntryEx(POBJECT_DIRECTORY Directory,
                                            PUNICODE_STRING Name,
                                            ULONG AttributeFlags) {
 BOOLEAN CaseInSensitive = (AttributeFlags & OBJ_CASE_INSENSITIVE) != 0;
 SIZE_T CharCount = Name->Length / sizeof(WCHAR);
 WCHAR* Buffer = Name->Buffer;
 ULONG Hash = 0;
 while (CharCount) {
   Hash = (Hash / 2) + 3 * Hash;
   Hash += RtlUpcaseUnicodeChar(*Buffer);
   Buffer++;
   CharCount--;
 }

 OBJECT_DIRECTORY_ENTRY* Entry = Directory->HashBuckets[Hash % 37];
 while(Entry) {
   if (Entry->HashValue == Hash) {
     if (RtlEqualUnicodeString(Name,
           ObpGetObjectName(Entry->Object), CaseInSensitive)) {
       ObReferenceObject(Entry->Object);
       return Entry->Object;
     }
   }
   Entry = Entry->ChainLink;
 }

 return NULL;
}
```

Figure 10. `ObpLookupDirectoryEntryEx()`

recursive directory entries, it would take around 24 hours! Now, I think we're going a bit over the top here, and by fiddling with the values we can get something that doesn't take too long to set up and gives us a long lookup time. But I'm still greedy; I want to see how far I can push the lookup time. Is there any way we can get the best of all worlds?

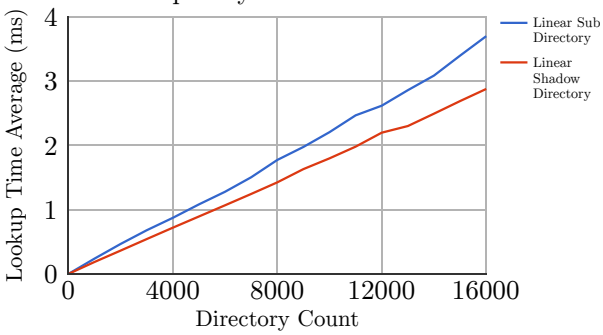
The final piece of the puzzle is to bring in Shadow directories, which allow the Object Manager a fallback path if it can't find an entry in a directory. You can use almost any other Object Manager directory as a shadow, which will allow us to control the lookup behavior. A Shadow Directory has a crucial difference from symbolic links, as it doesn't cause a reparse to occur in the lookup process. This means they're not restricted to the 64 reparse limit. As each lookup consumes a path component, eventually there will be no more paths to lookup. If we put together two directories in the following arrangement, we can pass a similar path to our recursive directory lookup, without actually creating all the directories.

Path: \A\A\A\A\A ...



So how does this actually work? If we open a path of the form `\A\A\A\A\A...`, the kernel will first lookup the initial 'A' directory. This is the directory on the left of the diagram. It will then try to open the next 'A' directory, which is on the right, which again it will find. Next the kernel again looks up 'A', but in this case it doesn't exist. As the directory has a shadow link to its parent, it looks there instead, finds the same 'A' directory, and repeats the process. This will continue until we run out of path elements to lookup.

So let's determine the performance of this approach. We'd perhaps expect it to be less performant relative to actually creating all those directories if only because of the cache effects of the processor. But hopefully it won't be too far behind.



Looks good. Yes, the performance is lower than actually creating the directories, but once we bring collisions into the mix, that's not really going to matter much. So the final result is that instead of creating 16,000 directories with 16,000 collisions we can do it with just 2 directories, which is far more manageable and only takes around 11 seconds on my workstation. So, to sign off, let's combine everything together.

1. 16,000 path components using 2 object directories in a shadow configuration
2. 16,000 collisions per directory
3. 64 symbolic link reparses

And the resulting time for a single lookup on my workstation is **drum roll please** 19 minutes! I think we might just be able to win the race condition with that.

Code examples can be found attached to this document.[10]

3.2 Conclusion

So after all that effort we can make the kernel take around 19 minutes to lookup a single controlled resource path. That's pretty impressive. We have many options to get the kernel to start the lookup process, allowing us to use not just files and registry keys but almost any named event. It's a typical tale of unexpected behavior when facing pathological input, and it's not really surprising Microsoft wouldn't optimize for this use case.

[10] `unzip pocorgtfo13.pdf object_manager_lookup_poc.cs`

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

by Micah Elizabeth Scott

Greetings, neighbors!

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

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

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

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

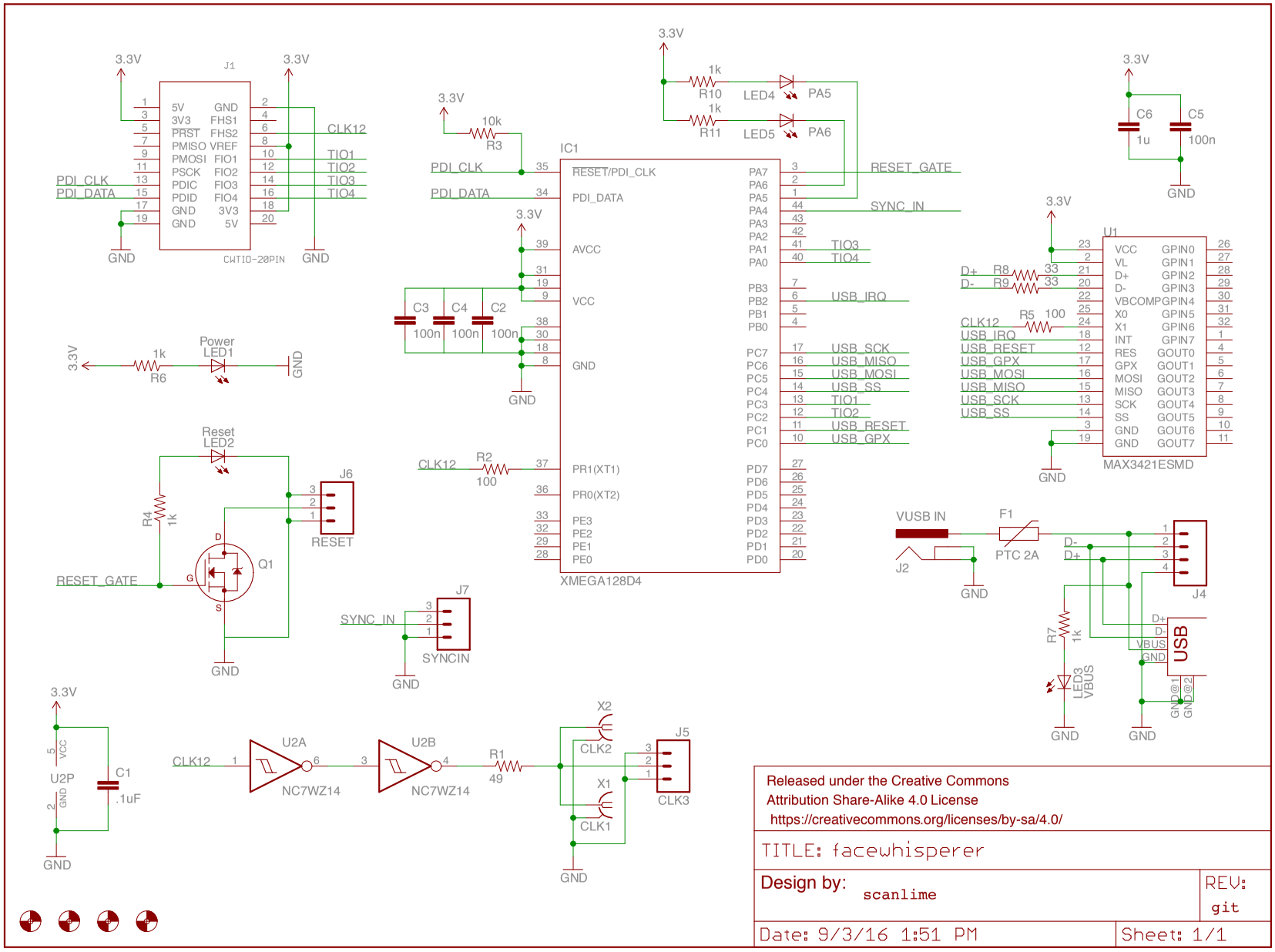
I recently made a video, a "Graphics Tablet Primer for Hackers," going into some detail on how a pen tablet input device actually works. I compared the electromagnetic power and data transfer to the low-frequency RFID cards still used by many door access control systems. At the time this was just a convenient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

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

4.1 A Very Descriptive Descriptor

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

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



TITLE: facewhisperer

Design by: scanlime

REV: git

Date: 9/3/16 1:51 PM

Sheet: 1/1

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

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

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

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

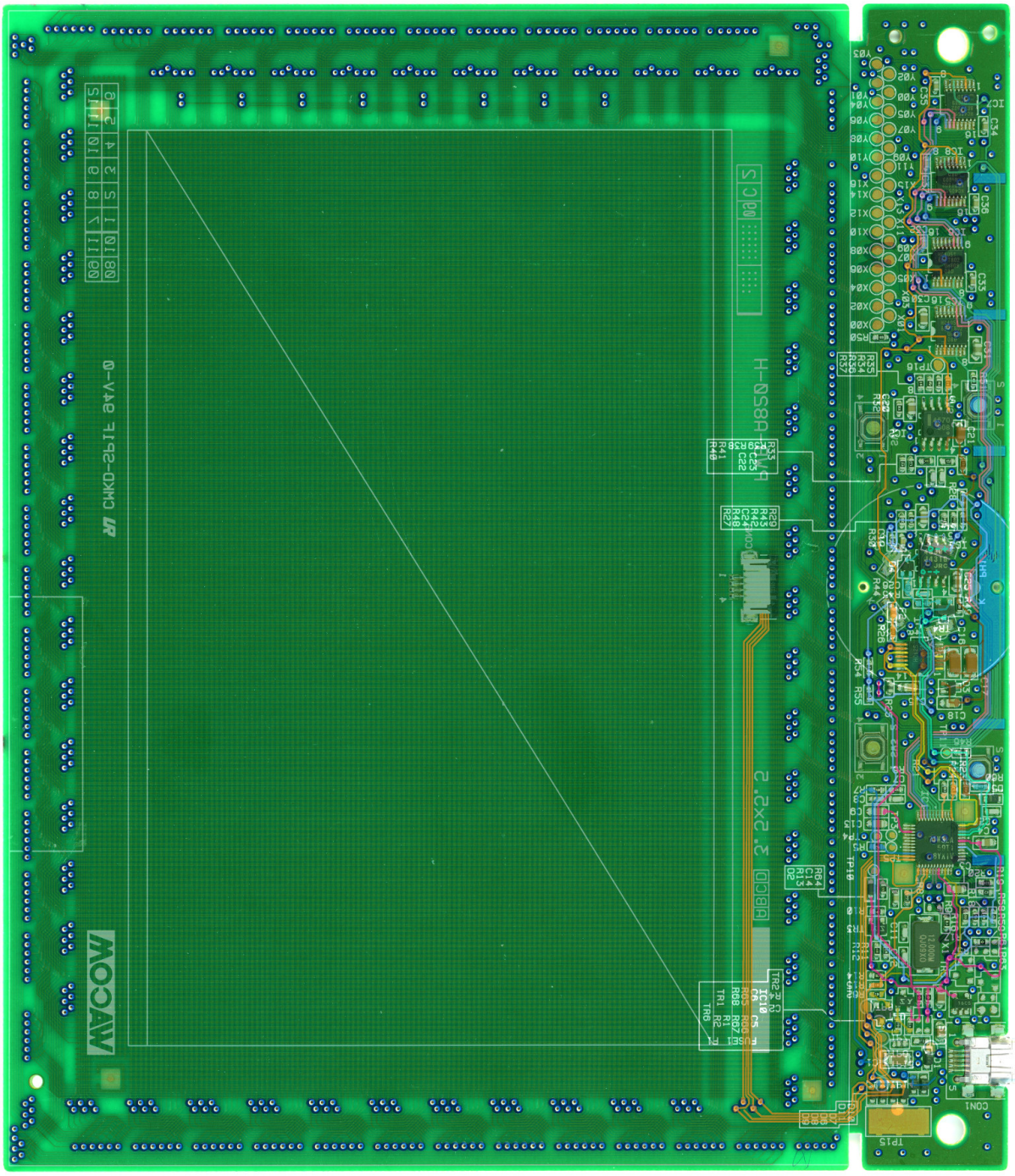
My humble addon is the FaceWhisperer, a USB host controller based on the MAX3421E chip, inspired of course by Travis Goodspeed's Facedancer21 tool. Whereas the USB host controller in your PC will be subject to many influences far outside your control, the USB host in the FaceWhisperer can be precisely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

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

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

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

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



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

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

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

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

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

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

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

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

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

4.2 Coding in Disappearing Ink

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

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

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

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

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

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

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

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

4.3 Amplitude Modulation Alchemy

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

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

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

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

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

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

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

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

4.4 More to do

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

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

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

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

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

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

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

5 Decoding AMBE+2 in MD380 Firmware in Linux

by Travis Goodspeed KK4VCZ
with kind thanks to DD4CR, DF8AV, and AB3TL

Howdy y'all,

In PoC||GTFO 10:8, I shared with you fine folks a method for extracting a cleartext firmware dump from the Tytera MD380. Since then, a rag-tag gang of neighbors has joined me in hacking this device, and hundreds of amateur radio operators around the world are using our enhanced firmware for DMR communications.

AMBE+2 is a fixed bit-rate audio compression codec under some rather strict patents, for which the anonymously-authored Digital Speech Decoder (DSD) project[14] is the only open source decoder. It doesn't do encoding, so if for example you'd like to convert your favorite Rick Astley tunes to AMBE frames, you'll have to resort to expensive hardware converters.

In this article, I'll show you how I threw together a quick and dirty AMBE audio decompressor for Linux by wrapping the firmware into a 32-bit ARM executable, then running that executable either natively or through Qemu. The same tricks could be used to make an AMBE encoder, or to convert nifty libraries from other firmware images into handy command-line tools.

This article will use an MD380 firmware image version 2.032 for specific examples, but in the spirit of building our own bird feeders, the techniques ought to apply just as well to your own firmware images from other devices.

– — — – — — — – — – —

Suppose that you are reverse engineering a firmware image, and you've begun to make good progress. You know where plenty of useful functions are, and you've begun to hook them, but now you are ready to start implementing unit tests and debugging chunks of code. Wouldn't it be nicer to do that in Unix than inside of an embedded system?

As luck would have it, I'm writing this article on an aarch64 Linux machine with eight cores and a few gigs of RAM, but any old Raspberry Pi or Android phone has more than enough power to run this code natively.

Be sure to build statically, targeting `arm-linux-gnueabi`. The resulting binary will run on armel and aarch64 devices, as well as damned near any Linux platform through Qemu's userland compatibility layer.

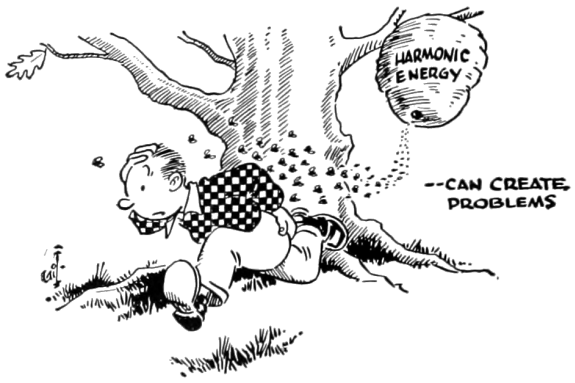
[14] `git clone https://github.com/szechyjs/dsd`

5.1 Dynamic Firmware Loading

First, we need to load the code into our process. While you can certainly link it into the executable, luck would have it that GCC puts its code sections very low in the executable, and we can politely ask `mmap(2)` to load the unpacked firmware image to the appropriate address. The first 48kB of Flash are used for a recovery bootloader, which we can conveniently skip without consequences, so the load address will be `0x0800c000`.

```
size_t length=994304;
int fd=open("experiment.img",0);
void *firmware=mmap(
     (void*) 0x0800c000, length,
     PROT_EXEC|PROT_READ|PROT_WRITE,
     MAP_PRIVATE,          //flags
     fd,                   //file
     0                     //offset
     );
```

Additionally, we need the 128kB of RAM at `0x20000000` and 64kB of TCRAM at `0x10000000` that the firmware expects on this platform. Since we'd like to have initialized variables, it's usually better go with dumps of live memory from a running system, but `/dev/zero` works for many functions if you're in a rush.



```
//Load an SRAM image.
int fdram=open("ram.bin",0);
void *sram=mmap(
      (void*) 0x20000000,
      (size_t) 0x20000,
      PROT_EXEC|PROT_READ|PROT_WRITE,
      MAP_PRIVATE,          //flags
      fdram,                //file
      0                     //offset
      );

//Create an empty TCRAM region.
int fdtcram=open("/dev/zero",0);
void *tcram=mmap(
      (void*) 0x10000000,
      (size_t) 0x10000,
      PROT_READ|PROT_WRITE, //protections
      MAP_PRIVATE,          //flags
      fdtcram,              //file
      0                     //offset
      );
```

5.2 Symbol Imports

Now that we've got the code loaded, calling it is as simple as calling any other function, except that our C program doesn't yet know the symbol addresses. There are two ways around this:

The quick but dirty solution is to simply cast a data or function pointer. For a concrete example, there is a null function at `0x08098e14` that simply returns without doing anything. Because it's a Thumb function and not an ARM function, we'll have to add one to that address before calling it at `0x08098e15`.

```
void (*nullsub)()=(void*) 0x08098e15;

printf("Trying to call nullsub().\n");
nullsub();
printf("Success!\n");
```

Similarly, you can access data that's in Flash or RAM.

```
printf("Manufacturer is: '%s'\n",
       0x080f9e4c);
```

Casting function pointers gets us part of the way, but it's rather tedious and wastes a bit of memory. Instead, it's more efficient to pass a textfile of symbols to the linker. Because this is just a textfile, you can easily export symbols by script from IDA Pro or Radare2.

The symbol file is just a collection of assignments of names to addresses in roughly C syntax.

```
/* Populates the audio buffer */
ambe_decode_wav = 0x08051249;
/* Just returns. */
nullsub = 0x08098e15;
```

You can include it in the executable by passing GCC parameters to the linker, or by calling `ld` directly.

```
CC=arm-linux-gnueabi-gcc-6 -static -g
$(CC) -o test test.c \
      -Xlinker --just-symbols=symbols
```

Now that we can load the firmware into process memory and call its functions, let's take a step back and see a second way to do the linking, by rewriting the firmware dump into an ELF object and then linking it. After that, we'll get along to decoding some audio.

5.3 Static Firmware Linking

While it's nice and easy to load firmware with `mmap(2)` at runtime, it would be nice and correct to convert the firmware dump into an object file for static linking, so that our resulting executable has no external dependencies at all. This requires both a bit of `objcopy` wizardry and a custom script for `ld`.

First, let's convert our firmware image dump to an ELF that loads at the proper address.

```
arm-linux-gnueabi-objcopy                \
  -I binary experiment.img               \
  --change-addresses=0x0800C000          \
  --rename-section .data=.experiment     \
  -O elf32-littlearm -B arm experiment.o
```

Sadly, `ld` will ignore our polite request to load this image at `0x08000C000`, because load addresses in Unix are just polite suggestions, to be thrown away by the linker. We can fix this by passing `-Xlinker -section-start=.experiment=0x0800C000` to `gcc` at compile time, so `ld` knows to place the section at the right address.

Similarly, the SRAM image can be embedded at its own load address.

Gratitude is a debt I intend to repay.

Uncledave

← Then

Now →

For the past 35 years radio amateurs throughout the world have been purchasing equipment and supplies from me. Their friendship and loyalty have been the determining factors in our success. For this we are grateful and it is time that we made an effort to express our appreciation in a material way.

Many amateur radio clubs need financial aid. Many others can use extra funds if these funds can be obtained without assessing their members. We have a plan which will greatly assist all amateur radio clubs.

For every order received until March 1, 1955, we will send our check for 15% of your order – to your radio club for deposit in their treasury. When you place your order, be sure to include the name and address of your club and treasurer.

My best wishes for a healthy, happy and prosperous New Year.

73 – CUL

Uncledave, W2APF

Fort Orange RADIO DISTRIBUTING COMPANY
904 BROADWAY, ALBANY, N. Y.
TELEPHONE ALBANY 5-1594

5.4 Decoding the Audio

To decode the audio, I decided to begin with the same `.amb` format that DSD uses. This way, I could work from their reference files and compare my decoding to theirs.

The `.amb` format consists of a four byte header (`2e 61 6d 62`) followed by eight-byte frames. Each frame begins with a zero byte and is followed by 49 bits of data, stored most significant bit first with the final bit in the least significant bit of its own byte.

To have as few surprises as possible, I take the eight packed bytes and extract them into an array of 49 shorts located at `0x20011c8e`, because this is the address that the firmware uses to store its buffer. Shorts are used for convenience in addressing during computation, even if they are a bit more verbose than they would be in a convenient calling convention.

```
1  //Re-use the firmware's own AMBE buffer.
   short *ambe=(short*) 0x20011c8e;
3
   int ambei=0;
5  for(int i=1;i<7;i++){//Skip first byte.
     for(int j=0;j<8;j++){
7      //MSBit first
       ambe[ambei++]=(packed[i]>>(7-j))&1;
9    }
   }
11 //Final bit in its own frame as LSBit.
   ambe[ambei++]=packed[7]&1;
```

Additionally, I re-use the output buffers to store the resulting WAV audio. In the MD380, there are two buffers of audio produced from each frame of AMBE.

```
   //80 samples for each audio buffer
2  short *outbuf0=(short*) 0x20011aa8;
   short *outbuf1=(short*) 0x20011b48;
```

The thread that does the decoding in firmware is tied into the MicroC/OS-II realtime operating system of the MD380. Since I don't have the timers and interrupts to call that thread, nor the I/O ports to support it, I'll instead just call the decoding routines that it calls.

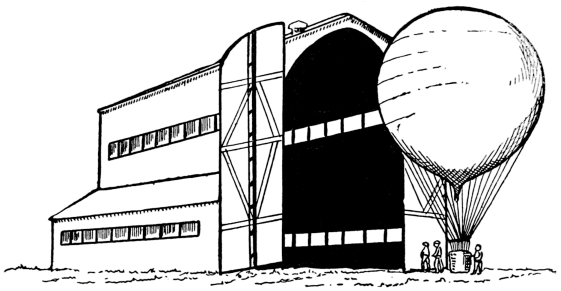
```
1  //Placed at 0x08051249
   int ambe_decode_wav(
3      signed short *wavbuffer,
       signed int eighty, //always 80
5      short *bitbuffer,  //0x20011c8e
       int a4,            //0
7      short a5,          //0
       short a6,          //timeslot, 0 or 1
9      int a7             //0x20011224
       );
```

For any parameter that I don't understand, I just copy the value that I've seen called through my hooks in the firmware running on real hardware. For example, `0x20011224` is some structure used by the AMBE code, but I can simply re-use it thanks to my handy RAM dump.

Since everything is now in the right position, we can decode a frame of AMBE to two audio frames in quick succession.

```
   //One AMBE frame becomes two audio frames.
2  ambe_decode_wav(
       outbuf0, 80, ambe,
4      0, 0, 0,
       0x20011224
6      );
   ambe_decode_wav(
8      outbuf1, 80, ambe,
       0, 0, 1,
10     0x20011224
       );
```

After dumping these to disk and converting to a `.wav` file with `sox -r 8000 -e signed-integer -L -b 16 -c 1 out.raw out.wav`, a proper audio file is produced that is easily played. We can now decode AMBE in Linux!



5.5 Runtime Hooks

So now we're able to decode audio frames, but this is firmware, and damned near everything of value except the audio routines will eventually call a function that deals with I/O—a function we'd better replace if we don't want to implement all of the STM32's I/O devices.

Luckily, hooking a function is nice and easy. We can simply scan through the entire image, replacing all `BX` (Branch and eXchange) instructions to the old functions with ones that direct to the new functions. False positives are a possibility, but we'll ignore them for now, as the alternative would be to list every branch that must be hooked.

The `BL` instruction in Thumb is actually two adjacent 16-bit instructions, which load a low and high half of the address difference into the link register, then `BX` against that register. (This clobbers the link register, but so does *any* `BL`, so the register use is effectively free.)

```
/* Calculates Thumb code to branch from
   one address to another. */
int calcbl(int adr, int target){
  /* Begin with the difference of the target
     and the PC, which points to just after
     the current instruction.*/
  int offset=target-adr-4;
  //LSBit doesn't count.
  offset=(offset>>1);

  /* The BL instruction is actually two
     Thumb instructions, with one setting
     the high part of the LR and the other
     setting the low part while swapping
     LR and PC. */
  int hi=0xF000 | ((offset&0xFFF800)>>11);
  int lo=0xF800 | (offset&0x7FF);

  //Return the pair as a single 32-bit word.
  return (lo<<16)|hi;
}
```

Now that we can calculate function call instructions, a simple loop can patch all calls from one address into calls to a second address. You can use this to hook the I/O functions live, rather than trapping them.

5.6 I/O Traps

What about those I/O functions that we've forgotten to hook, or ones that have been inlined to a dozen places that we'd rather not hook? Wouldn't it sometimes be easier to trap the access and fake the result, rather than hooking the same function?

You're in luck! Because this is Unix, we can simply create a handler for SIGSEGV, much as Jeffball did in PoC||GTFO 8:8. Your segfault handler can then fake the action of the I/O device and return.

Alternately, you might not bother with a proper handler. Instead, you can use GDB to debug the process, printing a backtrace when the I/O region at `0x40000000` is accessed. While GDB in Qemu doesn't support `ptrace(2)`, it has no trouble trapping out the segmentation fault and letting you know which function attempted to perform I/O.

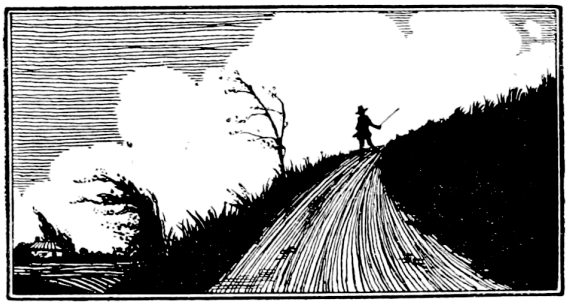
5.7 Conclusion

Thank you kindly for reading my ramblings about ARM firmware. I hope that you will find them handy in your own work, whenever you need to work with reverse engineered firmware away from its own hardware.

If you'd like to similarly instrument Linux applications, take a look at Jonathan Brossard's Witchcraft Compiler Collection,[15] an interactive ELF shell that makes it nice and easy to turn an executable into a linkable library.

The emulator from this article has now been incorporated into my md380tools[16] project, for use in Linux.

Cheers from Varaždin, Croatia,
–Travis 6A/KK4VCZ



[15] `git clone https://github.com/endrazine/wcc`
`unzip pocorgtfo13.pdf wcc.tar.bz2`

[16] `git clone https://github.com/travisgoodspeed/md380tools`

6 Password Weaknesses in Physical Security: Silliness in Three Acts

by Evan Sultanik

Dramatis Personæ

Disembodied Voice of Pastor Manul Laphroaig ..... Bard

Alice Feynman ..... Disciple of the Church of Weird Machines

Bob Schrute ..... Assistant to the Facility Security Officer

Havva al-Kindi ..... Alice's Old and Wise Officemate

The Ghost of Paul Erdős ..... Keeper of *The Book*

Act I: Memorize, Don't Compromise

Pastor: In the windowless bowels of a nondescript, Class A office building entrenched inside the Washington, D.C. beltway, we meet our heroine, Alice Feynman, lost on her way to a meeting with the Facility Security Officer.

Alice: Excuse me, which way is it to the security office?

Bob: You must be the new hire. Bob Schrute, assistant FSO. I can take you there right after I finish with this...

Alice: Alice. Nice to meet you. What're you doing?

Bob: Kaba Mas X-09 high security spin-lock. It's DSS-approved for use in our SCIFs. I'm resetting this one's passcode.

Alice: [*Blank Stare*]

Bob: U.S. Department of Defense (DoD) Defense Security Service (DSS). Sensitive Compartmented Information Facilities (SCIFs). The rooms where we are allowed to store and process classified information?

Alice: I see. I noticed those things all over this building.

Bob: They're ubiquitous. You'll see them anywhere in the country there's classified work going on. One on each door, and another on each safe. Super secure, too. Security in this office is no joke.

Alice: How do they work?

Bob: [*Throwing Alice the lock's manual.*] They run off of the electricity generated from spinning them, so you need to spin them a bit to get started. You see? The LCD on top shows you the current number. You enter three two-digit numbers. First one clockwise, second counter-clockwise, third clockwise, and then a final spin counter-clockwise to open. That's the passcode.

Alice: [*Flipping through the manual.*] Does each lock get a different passcode?

Bob: Yes. That's why we have this [*handing Alice a magnet stuck to the side of the door*].

Alice: Ah I see. It's a phone keypad. So you use a mnemonic to remember each passcode?

Bob: Exactly. [*Pointing to a poster on the wall with his own mugshot and memetic letters emblazoning "MEMORIZE, DON'T COMPROMISE", he sternly repeats that slogan:*] **Memorize, don't compromise**.

Alice: [*"Is this guy serious?" face.*]

Bob: You think you could crack it? FALSE. [*Flamboyantly produces a pocket calculator that had been hidden somewhere on his person.*] Three two-digit numbers. That's 100 times 100 times 100, so ... there are a million possible codes. I've set this to have a timeout of four minutes after each failed attempt. So, trying all possible combinations

would take . . . [*furiously punching at the calculator*] . . . almost eight years! We change each code once every couple months, so even if you could continuously try codes for eight hours a day, you'd have . . . [*more furious punching*] . . . about seven tenths of one percent chance of getting the code right.

Alice: [*Handing the manual back.*] I didn't see anything in here about an automatic lockout after too many failed attempts.

Bob: [*Pointing to his minuscule biceps.*] These provide the lockout.

Alice: Are you ready to take me to the security office now?

Bob: Fine.

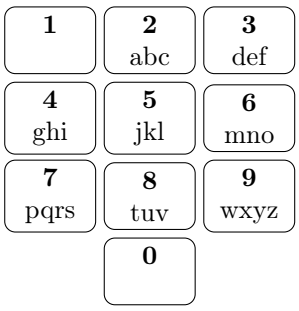
Act II: Surely You're Joking

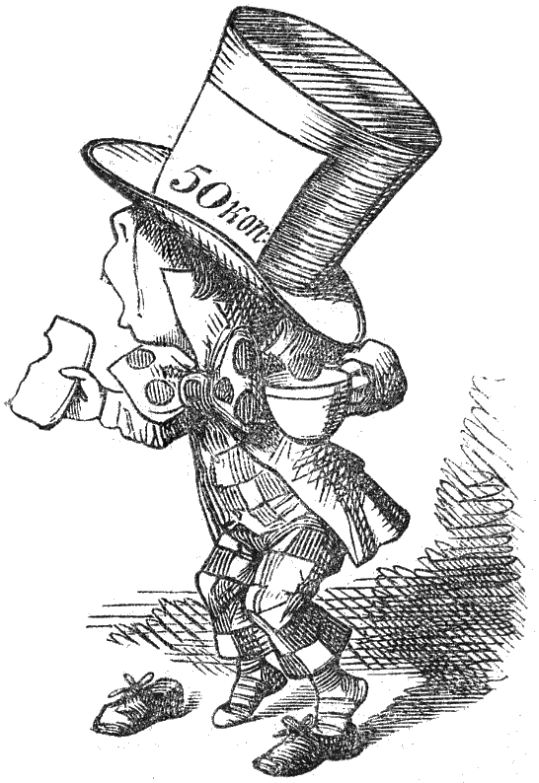
Pastor: Two weeks later, Alice has settled into her office, which she shares with Havva al-Kindi. She hasn't had a chance to play with those nifty locks at all yet; her clearance is still being processed. Most of her time is spent idling or doing busy-work while she waits to be approved to work on a real project.

Alice: [*On her desk phone*] Yes. Yes, no problem. By close of business today. No problem. Bye.

Pastor: As Alice hangs up the phone, she notices something odd about the keypad, and immediately remembers the magnet Bob had showed her.

Alice: [*Gets up and starts drawing on her whiteboard.*]





Havva: What are you doing?

Alice: Did you ever notice that the numbers zero and one don't have any letters on the phone?

Havva: Sure! You're probably too young to have ever used a rotary phone, right? Back when phone numbers were only seven digits long, the first two numbers represented the exchange, and a mnemonic was given to each exchange. [*Singing and tapping on her desk*] *Bum-dah-bum bah-duh-bum bahhh dummm! PEnnsylvania Six Five Thousand!* No? It was a big Glenn Miller hit! My parents used to play it all the time when I was a kid. That song is referring to the phone number for the Hotel Pennsylvania in New York, which to this day is still (212) PE6-5000.

Alice: Oh yeah! I went there once for HOPE.

Havva: Hope? Anyhow, for various reasons, the numbers zero and one were never used in exchanges, which meant they never occurred at the beginning of phone numbers, which meant they couldn't have letters associated with them.

ALICE: Interesting! [*Continuing on the whiteboard*] $8^6 = \ldots$ [*a pause to consult her computer*] 262144. $1 - 262144 \div 1000000 = \ldots$ 0.738. Wow! So, if there are only eight buttons with letters, that reduces the number of possible phone numbers associated with six-letter mnemonics by 74% compared to if all the buttons had letters!

HAVVA: I guess that's true. There are also certain phone numbers you'll never be able to have English mnemonics for, because the buttons for 5, 7, and 9 don't have any vowels. So you can't make a mnemonic for a phone number that only uses those three numbers.

ALICE: Wow, yeah, that's another $3^6 = \ldots$ [*quickly doing some math in her head this time*] 729 codes that don't have mnemonics.

HAVVA: Codes?

ALICE: Er, I mean "phone numbers."

HAVVA: I'll bet there are certain "codes" that don't have any English words associated with them. Plus, letters in English words don't all occur at the same frequency: It's much more likely that a word will have the letter "e" than it will have the letter "x."

ALICE: [*Opens up a terminal on her computer.*]

```
$ grep '^.\{6\}$' /usr/share/dict/words | wc -l
    17706
$ echo `!!` / 1000000 | bc -l
.01770600000000000000
```

PASTOR: And thus, Alice had discovered that fewer than 2% of the million possible codes actually map to English words.

ALICE: [*Once again at the whiteboard.*]

HA	CK	ER
42	25	34

[*Back at the computer.*]

```
$ grep -i '^.\{4\}er$' /usr/share/dict/words \
  | wc -l
    1562
```

About 10% of six-letter English words end with the letters "ER"!

[*Back at the board, with long pauses.*]

DO	SA	GE
36	72	43
EN	RA	GE
36	72	43
FO	RA	GE
36	72	43
FO	RB	ID
36	72	43

PASTOR: And many words share the same code. In fact, Alice quickly wrote a script to count the number of unique codes possible from six-letter English words[17].

ALICE: There are only 14684 possible codes to check! That would take ... only about 40 days to brute-force crack!

Act III: The Book

PASTOR: Later that day, Alice is at her favorite dive, decompressing with some of her side projects.

PAUL: [*Sits down next to Alice at the bar. Wheel of Fortune is playing on an ancient CRT.*] Television is something the Russians invented to destroy American education.

ALICE: [*Tippling a brown liquor, neat, while working on her laptop. Paul's comment draws her attention to the TV. Alice notices that some letters are given away "for free" and remembers what Havva had said about letter frequency. She quickly grabs her notebook and jots down the letters as a reminder.*] R, S, T, L, N, E.

PAUL: [*Noticing Alice's notebook.*] Yes, these are very common letters in English. My native language does not use "r" as much. But what do I know about English? I learned it from my father, who taught it to himself by reading English novels in one of Joe's Gulags. [*Awkward pause while Alice struggles with how to respond.*] Have you discovered anything beautiful? [*Pointing into her notebook.*]

ALICE: Oh that? I've been thinking about mnemonics for passcodes.

[17] `$ grep '^.\{6\}$' /usr/share/dict/words | tr '[:upper:]' '[:lower:]' | sed 's/[abc]/2/g; s/[def]/3/g; s/[ghi]/4/g; s/[jkl]/5/g; s/[mno]/6/g; s/[pqrs]/7/g; s/[tuv]/8/g; s/[wxyz]/9/g' | sort | uniq | wc -l`

PAUL: [*Pointing to the drink:*] That poison will not help you. [*Produces a small pill bottle out of his shirt pocket, raises it to eye level, drops it, and then catches it with the same hand before it hits the bar.*]

ALICE: Haven't you heard? The *Ballmer Peak* is real! Or at least that's what I read on Stack Exchange.

PAUL: Pál Erdős. My brain is open.

PASTOR: Alice introduces herself and proceeds to explain all of her findings to Paul.

ALICE: ...and I just finished sorting the 14684 distinct codes by the number of words associated with them. That way, if I try the codes in order of decreasing word associations, then it will maximize my chances of cracking the code sooner than later.

PAUL: Yes, if codewords are chosen uniformly from all six-letter English words. Can I see the distribution of word frequency? [*Grabbing a napkin, stealing Alice's pen, and scribbling some notes.*] Using your method, after fewer than 250 attempts, there is a 5% probability that you will have cracked the code. After about 5700 attempts, there will be a 50% probability of success.

ALICE: [*Typing on her computer.*] That's only about 16 days!

PASTOR: An adversary with intermittent access to the lock—for example, after hours—could quite conceivably crack the code in less than a month.

PAUL: If there exists a method that allows the code-breaker to detect whether each successive two-digit subcode is correct before entering the next two-digit subcode,...

PASTOR: ...otherwise known as a "vulnerability"...

PAUL: ...[*annoyed about having been interrupted, even if by the disembodied voice of a narrator*] then the expected value for the length of time required to crack the code is on the order of minutes. [*Mumbling toward the fourth wall:*] That Pastor is more annoying than the SF.

ALICE: What?

PAUL: SF means "Supreme Fascist." This would show that God is bad. I do not claim that this is correct, or that God exists. It is just a sort of half-joke. There is an anecdote I once heard. Suppose Israel Gelfand and his advisor, Andrei Kolmogorov, were to both arrive in a country with a lot of mountains. Kolmogorov would immediately try and climb the highest mountain. Gelfand would immediately start building roads. What would you do?

ALICE: I would learn to fly an airplane so I could discover new mountain ranges. What about you?

PAUL: Some might say that *is* what I do. My friends might add that they pay for the fuel. But really, I just try to keep the SF's score low. How can we create mnemonics that are not vulnerable to your attack?

ALICE: Well, I guess the first thing to do is create a keypad layout that uses zero and one.

PAUL: Yes, but my academic sibling Pólya would say that we first need to understand the *problem*. Ideally, we want a keypad layout that produces an injective mapping from the six-letter English words into the natural numbers from zero to one million.

ALICE: Injective?

PAUL: Such that no two words produce the same code number.

ALICE: Is that even possible?

PAUL: I do not know. I believe this is an instance of the *multiple subset sum* problem, related to the knapsack problem.

ALICE: Ah yeah, I remember that from my algorithms class. It's NP-Complete, right?

PAUL: Yes, and likely intractable for problems even as small as this one. The total number of possible keypad mappings is 100 million billion billion. But it is easy for us to check the pigeons.

ALICE: Huh?

PAUL: The *pigeonhole principle*. For any subset of m letters within a word, there can be at most 10^{6-m} words that have that pattern of

letters. If there are more, then there must be a collision, no matter the mapping we choose.

Alice: Ah, I see. That's easy enough to check! [*Typing.*]

```
for m in range(2,6):
    hits = {}
    for word in words:
        for indexes in itertools.
    combinations(range(len(word)), m):
            key = tuple((word[i], i)
    for i in indexes)
            if key not in hits:
                hits[key] = 1
            else:
                hits[key] += 1
    max_hits = 10**(6-m)
    for key, h in hits.iteritems():
        if h <= max_hits:
            continue
        k = ['.' for i in range(6)]
        for c, i in key:
            k[i] = c
        print "".join(k), h - max_hits
```

So, there are fourteen five-letter suffixes like "inder", "aggle", and "ingle" that will all produce at least one collision. I guess there's no way to make a perfect mapping.

Paul: Gelfand advised Endre Szemerédi. This problem is reminiscent of Szemerédi's use of *expander graphs* in pseudo-random number generation. What we want to do is take a relatively small set of inputs (being the six-letter English words) and use an expander graph as an embedding into the natural numbers between one and a million, such that the resulting distribution mimics uniformity.

Alice: That sounds ... difficult.

Paul: Constructing expander graphs is extremely difficult. But I think Szemerédi would agree that interesting things rarely happen in fewer than five dimensions.

Alice: I am a pragmatist. How about we use a genetic algorithm to evolve a near optimal mapping?

Paul: Such a solution would not be from *The Book*, but it would provide you with a mapping.

Alice: What book?

Paul: The Book in which the SF keeps all of the most beautiful solutions.

Alice: Well, I think I'll try my hand at a scruffy genetic algorithm. I need a decent mapping if I ever want to publish this in PoC||GTFO!

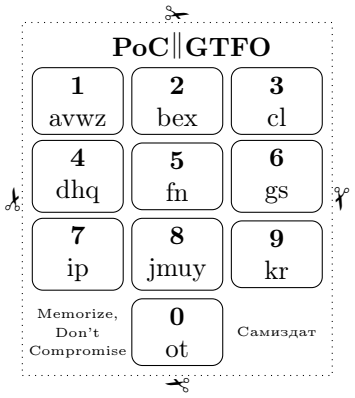
Paul: What is PoC||GTFO?

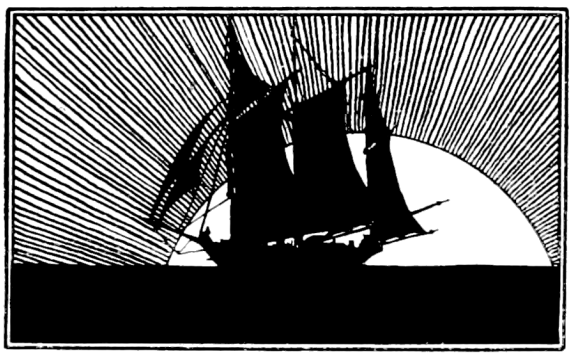
Alice: It's... I guess it's a sort of bible.

Paul: Then the only difference between your Book and mine are the fascists who created them. Maybe we will continue tomorrow ... if I live.

Alice: [*Looking up from her keyboard.*] Can I buy you a drink? [*Paul has vanished.*]

Pastor: The moral of the story, dear neighbors, is *not* that these locks are inherently vulnerable; if used properly, they are in fact incredibly secure. We must remember that these locks are only as secure as the codes humans choose to assign to them. Using a phone keypad mapping on six-letter English dictionary words is the physical security equivalent of a website arbitrarily limiting passwords to eight characters.





7 Reverse Engineering the LoRa PHY

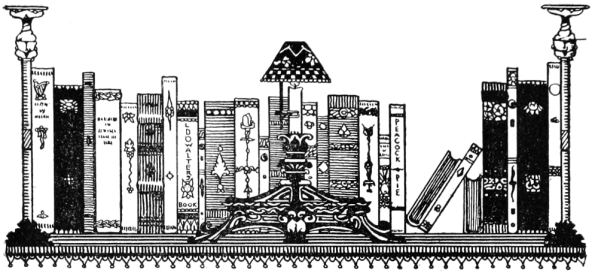
by Matt Knight

It's 2016, and everyone's favorite inescapable buzzword is IoT, or the "Internet of Things." The mere mention of this phrase draws myriad reactions, depending on who you ask. A marketing manager may wax philosophical about swarms of connected cars eradicating gridlock forever, or the inevitability of connected rat traps intelligently coordinating to eradicate vermin from midtown Manhattan,[18] while a security researcher may just grin and relish in the plethora of low-power stacks and new attack surfaces being applied to cyber-physical applications.

IoT is marketing speak for connected embedded devices. That is, inexpensive, low power, resource constrained computers that talk to each other, possibly on the capital-I Internet, to exchange data and command and control information. These devices are often installed in hard to reach places and can be expected to operate for years. Thus, easy to configure communication interfaces and extreme power efficiency are crucial design requirements. While 2G cellular has been a popular mechanism for connecting devices in scenarios where a PAN or wired technology will not cut it, AT&T's plans to sunset 2G on January 1, 2017 and LTE-M Rel 13's distance to widespread adoption presents an opportunity for new wireless specifications to seize market share.

LoRa is one such nascent wireless technology that is poised to capture this opportunity. It is a Low Power Wide Area Network (LPWAN), a class of wireless communication technology designed to connect low power embedded devices over long ranges. LoRa implements a proprietary PHY layer; therefore the details of its modulation are not published.

This paper presents a comprehensive blind signal analysis and resulting details of LoRa's PHY, chronicles the process and pitfalls encountered along the way, and aspires to offer insight that may assist security researchers as they approach their future unknowns.



7.1 Casing the Job

I first heard of LoRa in December 2015, when it and other LPWANs came up in conversation among neighbors. Collectively we were intrigued by its advertised performance and unusual modulation, thus I was motivated to track it down and learn more. In the following weeks, I occasionally scanned the 900 MHz ISM spectrum for signs of its distinctive waveform (more on that soon), however searches in the New York metropolitan area, Boston, and a colleague's search in San Francisco yielded no results.

Sometime later I found myself at an IoT security meetup in Cambridge, MA that featured representatives from Senet and SIGFOX, two major LPWAN players. Senet's foray into LoRa started when they sought to remotely monitor fluid levels in home heating oil tank measurement sensors to improve the existing process of sending a guy in a truck to read it manually. Senet soon realized that the value of this infrastructure extended far beyond the heating oil market and has expanded their scope to becoming a IoT cellular data carrier of sorts. While following up on the company I happened upon one of their marketing videos online. A brief segment featured a grainy shot of a coverage map, which revealed just enough to suggest the presence of active infrastructure in Portsmouth, NH. After quick drive with my Ettus B210 Software Defined Radio, I had my first LoRa captures.

7.2 First Observations and OSINT

LoRa's proprietary PHY uses a unique chirp spread spectrum (CSS) modulation scheme, which encodes information into RF features called chirps. A chirp

[18] LoRaWan in the IoT Industrial Panel, presentation by Jun Wen of Cisco.

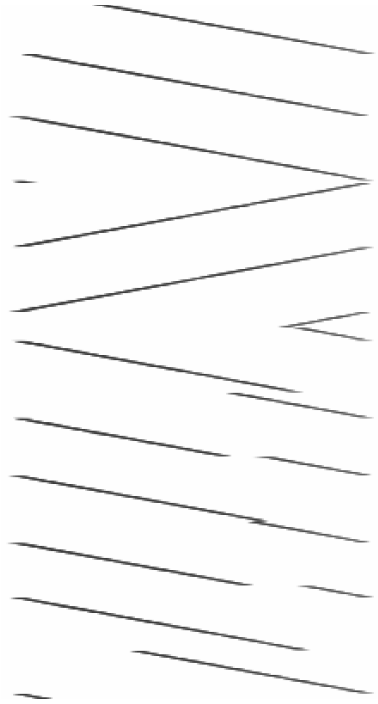


Figure 11. Spectrogram of a LoRa packet.

is a signal whose frequency is increasing or decreasing at a constant rate, and they are unmistakable within the waterfall. A chirp-based PHY is shown in Figure 11.

Contrasted with FSK or OFDM, two common PHYs, the differences are immediately apparent.

Modulation aside, visually inspecting a spectrogram of LoRa's distinct chirps reveals a PHY structure that is similar to essentially all other digital radio systems: the preamble, start of frame delimiter, and then the data or payload.

Since LoRa's PHY is proprietary, no PHY layer specifications or reference materials were available. However, thorough analysis of open source and readily available documentation can greatly abbreviate reverse engineering processes. When I conducted this investigation, a number of useful documents were available.

First, the Layer 2+ LoRaWAN stack is published, containing clues about the PHY.

Second, several application notes were available for Semtech's commercial LoRa modules.[19] These were not specs, but they did reference some PHY-layer components and definitions.

Third, a European patent filing from Semtech described a CSS modulation that could very well be LoRa.

Finally, neighbors who came before me had produced open-source prior art in the form of a partial `rtl-sdrangelove` implementation and a wiki page,[20] however in my experience the `rtl-sdrangelove` attempt was piecemeal and neglected and the wiki contained only high level observations. These were not enough to decode the packets that I had captured in New Hampshire.

7.3 Demodulation

OSINT gathering revealed a number of key definitions that informed the reverse engineering process. A crucial notion is that of the spreading factor (SF): the spreading factor represents the number of bits packed into each symbol. A symbol, for the unordained, is a discrete RF energy state that represents some quantity of modulated information (more on this later.) The LoRaWAN spec revealed that the chirp bandwidth, that is the width of the channel that the chirps traverse, is 125 kHz,

[19] Semtech AN1200.18, AN1200.22.

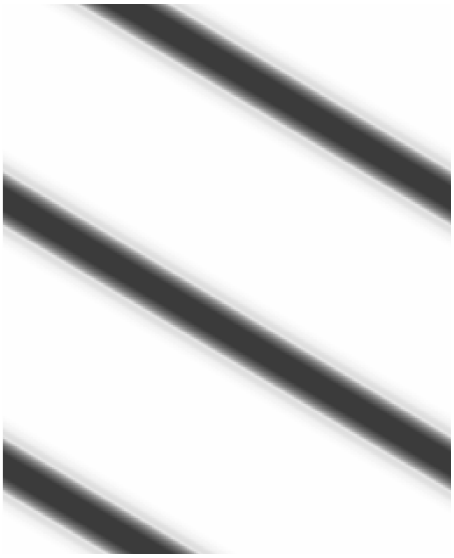
[20] Decoding LoRa on the RevSpace Wiki

250 kHz, or 500 kHz within American deployments. The chirp rate, which is intuitively the first derivative of the signal's frequency, is a function of the spreading factor and the bandwidth: it is defined as $\text{bandwidth}/2^{(\text{spreading\_factor})}$. Additionally, the absolute value of the downchirp rate is the same as the upchirp rate.[21]

Back to the crucial concept of symbols. In LoRa, symbols are modulated onto chirps by changing the instantaneous frequency of the signal – the first derivative of the frequency, the chirp rate, remains constant, while the signal itself "jumps" throughout its channel to represent data. The best way to intuitively think of this is that the modulation is frequency-modulating an underlying chirp. This is analogous to the signal alternating between two frequencies in a 2FSK system, where one frequency represents a `0` and the other represents a `1`. The underlying signal in that case is a signal of constant frequency, rather than a chirp, and the number of bits per symbol is 1. How many data bits are encoded into each frequency jump within LoRa? This is determined by the spreading factor.

The first step to extracting the symbols is to dechirp the received signal. This is done by channelizing the received signal to the chirp's bandwidth and multiplying the result against a locally-generated complex conjugate of whichever chirp is being extracted.

A locally generated chirp might look like this.

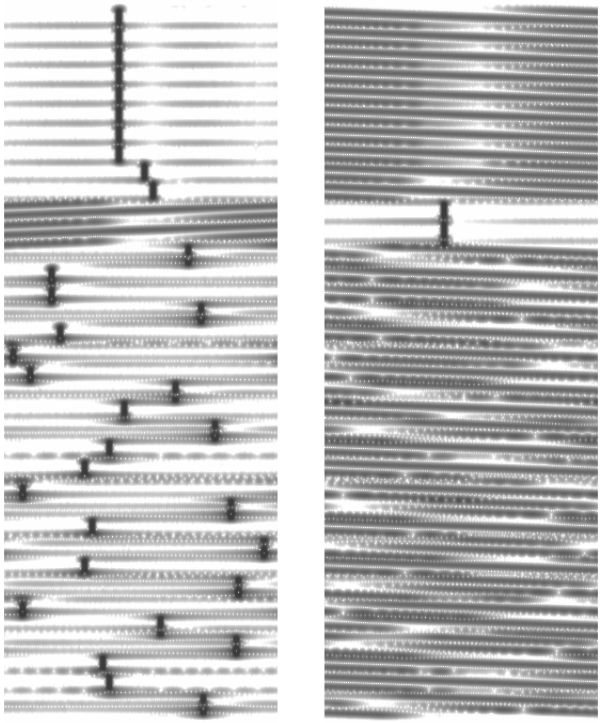


[21] See Semtech AN1200.22.

Since both upchirps and downchirps are present in the modulation, the signal should be multiplied against both a local upchirp and downchirp, which produces two separate IQ streams. Why this works can be reasoned intuitively, since waves obey superposition, multiplying a signal with frequency $f0$ against a signal with frequency $-f0$ results in a signal with frequency 0, or DC. If a chirp is multiplied against a copy of itself, it will result in a signal of $2 * f0$, which will spread its energy throughout the band. Thus, generating a local chirp at the negative chirp rate of whichever chirp is being processed

results in RF features with constant frequency that can be handled nicely.

In following examples, the left image shows de-chirped upchirps while the right shows de-chirped downchirps:



This de-chirped signal may be treated similarly to MFSK, where the number of possible frequencies is $M = 2^{(\text{spreading\_factor})}$. The Fast Fourier Transform (FFT) is the tool used to perform the actual symbol measurement. Fourier analysis shows that a signal can be modeled as a summed series of basic periodic functions (i.e., a sine wave) at various frequencies. A FFT decomposes a signal into the frequency components that comprise it, returning the power and phase of each component present. Each component to be extracted is colloquially called a "bin;" the number of bins is specified as the "FFT size" or "FFT width."

Thus, by taking an M-bin wide FFT of each IQ stream, the symbols may be resolved by finding the argmax, which is the bin with the most powerful component of each FFT. This works out nicely because a de-chirped CSS symbol turns into a signal with constant frequency; all of the symbol's energy should fall into a single bin.[22]

With the signal de-chirped, the remainder of the demodulation process can be described in three steps. These steps mimic the process required for essentially all digital radio receivers.

First, we'll identify the start of the packet by finding a preamble. Then, we'll synchronize with the start of the packet, so that we may conclude in demodulating the payload by measuring its aligned symbols.

7.3.1 Finding the Preamble

A preamble is a feature included in modulation schemes to announce that a packet is soon to follow. By visual inspection, we can infer that LoRa's preamble is represented by a series of continuous upchirps. Once de-chirped and passed through an FFT, all of the preamble's symbols wind up residing within the same FFT bin. Thus, a preamble is detected if enough consecutive FFTs have the same argmax.

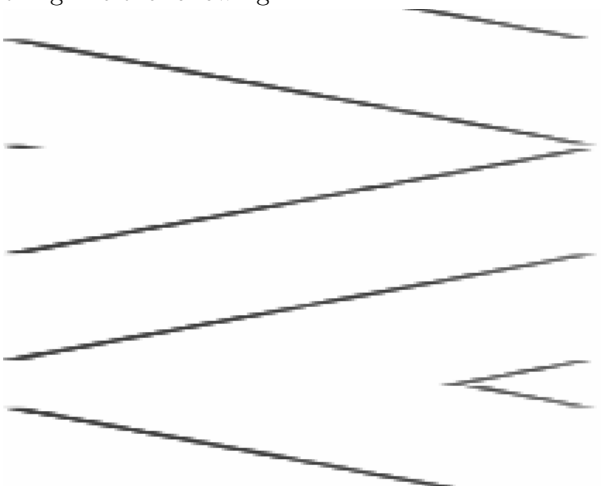
7.3.2 Synchronizing with the SFD

With our receiver aware that it's about to receive a packet, the next step is to accurately synchronize with it so that symbols can be resolved accurately. To facilitate this, modern radio systems often advertise the start of the packet's data unit with a Start of

[22]It may be possible to do this using FM demodulation rather than FFTs, however using FFTs preserves power information that is useful for framing the packet without knowing its definitive length.

Frame Delimiter, or SFD, which is a known symbol distinct from the preamble that receivers are programmed to look for. For LoRa, this is where the downchirps come in.

The SFD is composed of two and one quarter downchirps, while all the other symbols are represented by upchirps. With preamble having been found, our receiver should look for two consecutive downchirps to synchronize against. It looks something like the following:



Accurate synchronization is crucial to properly resolving symbols. If synchronization is off by enough samples, when FFTs are taken each symbol's energy will be divided between two adjacent FFTs. Until now, the FFT process used to resolve the symbols processed $2^{(\text{spreading\_factor})}$ samples per FFT with each sample being processed exactly once, however after a few trial runs it became evident that this coarse synchronization would not be sufficiently accurate to guarantee good fidelity.

Increasing the time-based FFT resolution was found to be a reliable method for achieving an accurate sync. This is done by shifting the stream of de-chirped samples through the FFT input buffer, processing each sample multiple times, to "overlap" adjacent FFTs. This increases the time-based resolution of the FFT process at the expense of being more computationally intensive. Thus, overlapping FFTs are only used to frame the SFD; non-overlapped FFTs with each sample being processed exactly once are taken otherwise to balance accuracy and computational requirements.

Technically there's also a sync word that precedes the SFD, but my demodulation process described in this article does not rely on it.

[23]European Patent #13154071.8/EP20130154071

7.3.3 Demodulating the Payload

Now synchronized against the SFD, we are able to efficiently demodulate the symbols in the payload by using the original non-overlapping FFT method. However, since our receiver's locally generated chirps are likely out of phase with the chirp used by the transmitter, the symbols appear offset within the set range $[0 : 2^{(\text{spreading\_factor})} - 1]$ by some constant. It was surmised that the preamble would be a reliable element to represent symbol `0`, especially given that the aforementioned sync word's value is always referenced from the preamble. A simple modulo operation to normalize the symbol value relative to the preamble's zero-valued bin produces the true value of the symbols, and the demodulation process is complete.

7.4 Decoding, and its Pitfalls

Overall, demodulation proved to not be too difficult, especially when you have someone like Balint Seeber feeding you advice and sagely wisdom. However, decoding is where the fun (and uncertainty) really began.

First, why encode data? In order to increase over the air resiliency, data is encoded before it is sent. Thus, the received symbols must be decoded in order to extract the data they represent.

The documentation I was able to gather on LoRa certainly suggested that figuring out the decoding would be a snap. The patent application describing a LoRa-like modulation described four decoding steps that were likely present. Between the patent and some of Semtech's reference designs, there were documented algorithms or detailed descriptions of every step. However, these documents slowly proved to be lies, and my optimism proved to be misplaced.

7.4.1 OSINT Revisited

Perhaps the richest source of overall hints was Semtech's European patent application.[23] The patent describes a CSS-based modulation with an uncanny resemblance to LoRa, and goes so far as to walk step-by-step through the encoding elements present in the PHY. From the encoder's perspective, the patent describes an encoding pipeline of forward error correction, a diagonal interleaver, data whitening, and gray indexing, followed by the just-described modulation process. The reverse process

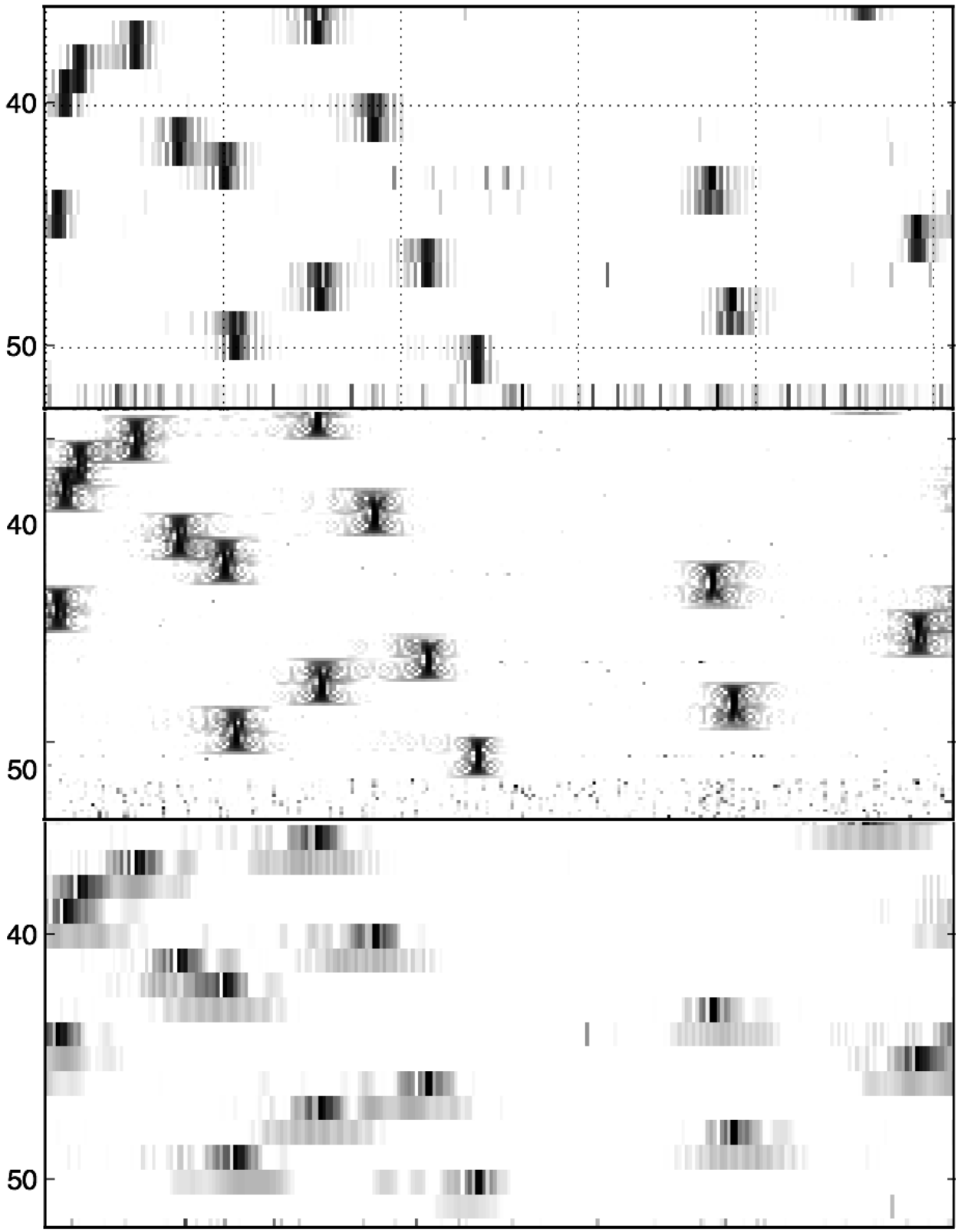


Figure 12. The top is pre-sync and non-overlapped, middle is pre-sync overlapped, bottom is synchronized and non-overlapped.

would be performed by the decoder. The patent even defines an interleaver algorithm, and Semtech documentation includes several candidate whitening algorithms.

The first thing to try, of course, was to implement a decoder exactly as described in the documentation. This involved, in order:

1. Undoing gray coding applied to the symbols.
2. Dewhitening using the algorithms defined in Semtech's documentation.
3. Deinterleaving using the algorithm defined in Semtech's patent.
4. Processing the Hamming forward error correction hinted at in Semtech's documentation.

First, let's review what we have learned about each step listed above based on open-source research, and what would be attempted as a result.

Gray Indexing Given the nomenclature ambiguity in the Semtech patent, I also decided to test no gray coding and reverse gray coding in addition to forward gray coding. These were done using standard algorithms.

Data Whitening Data whitening was a colossal question mark while looking at the system. An ideal whitening algorithm is pseudorandom, thus an effective obfuscator for all following components of the system. Luckily, Semtech appeared to have published the algorithm candidates in Application Note AN1200.18. Entitled "Implementing Data Whitening and CRC Calculation in Software on SX12xx Devices," it describes three different whitening algorithms that were relevant to the Semtech SX12xx-series wireless transceiver ICs, some of which support LoRa. The whitening document provided one CCITT whitening sequences and two IBM methods in C++. As with the gray indexing uncertainty, all three were implemented and permuted.

Interleaver Interleaving refers to methods of deterministically scrambling bits within a packet. It improves the effectiveness of Forward Error Correction, and will be elaborated on later in this text. The Semtech patent application defined a diagonal interleaver as LoRa's probable interleaver. It is a block-style non-additive diagonal interleaver that shuffles bits within a block of a fixed size. The interleaver is defined as: Symbol$(j, (i+j)\%\text{PPM}) =$ Codeword(i, j) where $0 <= i <$ PPM, $0 <= j < 4 +$ RDD In this case, PPM is set to the spreading factor (or $spreading\_factor - 2$ for the PHY header and when in low data rate modes), and RDD is set to the number of parity bits used by the Forward Error Correction scheme (ranging $[1:4]$).

There was only one candidate illustrated here, so no iteration was necessary.

Forward Error Correction The Semtech patent application suggests that Hamming FEC be used. Other documentation appeared to confirm this. A custom FEC decoder was implemented that originally just extracted the data bits from their standard positions within `Hamming(8,4)` codewords, but early results were negative, so this was extended to apply the parity bits to repair errors.

Using a Microchip RN2903 LoRa Mote, a transmitter that was understood to be able to produce raw frames, a known payload was sent and decoded using this process. However, the output that resulted bore no resemblance to the expected payload. The next step was to inspect and validate each of the algorithms derived from documentation.

After validating each component, attempting every permutation of supplied algorithms, and inspecting the produced binary data, I concluded that something in LoRa's described encoding sequence was not as advertised.

7.5 Taking Nothing for Granted

The nature of analyzing systems like this is that beneath a certain point they become a black box. Data goes in, some math gets done, RF happens, said math gets undone, and data comes out. Simple enough, but when encapsulated as a totality it becomes difficult to isolate and chase down bugs in each component. Thus, the place to start was at the top.

"Fred understands what they're saying since he converted to single sideband"

7.5.1 How to Bound a Problem

The Semtech patent describes the first stage of decoding as "gray indexing." Gray coding is a process that maps bits in such a way that makes it resilient to off-by-one errors. Thus, if a symbol were to be measured within ± 1 index of the correct bin, the gray coding would naturally correct the error. "Gray indexing," ambiguously referring to either gray coding or its inverse process, was initially understood to mean forward gray coding.

The whitening sequence was next in line. Data whitening is a process applied to transmitted data to induce randomness into it. To whiten data, the data is XORed against a pseudorandom string that is known to both the transmitter and the receiver. This does good things from an RF perspective, since it induces lots of features and transitions for a receiver to perform clock recovery against. This is functionally analogous to line coding schemes such as Manchester encoding, but whitening offers one pro and one con relative to line coding: data whitening does not impact the effective bit rate as Manchester encoding does,[24] but this comes at the expense of legibility due to the pseudorandom string.

At this point, it is important to address some of the assumptions and inferences that were made to frame the following approach. While the four decoding stages were thrown into question by virtue of the fact that at least one of the well-described algorithms was not correct, certain implied properties could be generalized for each class of algorithm, even if the implementation did not match exactly.

I made a number of assumptions at this point, which I'll describe in turn.

First, the interleaver in use is non-additive. This means that while it will reorder the bits within each interleaving block, it will not cause any additional bits to be set or unset. This was a reasonable assumption because many block-based interleavers are non-additive, and the interleaver defined in the patent is non-additive as well. Even if the interleaver used a different algorithm, such as a standard block interleaver or a different type of diagonal interleaver, it could still fit within this model.

Second, the forward error correction in use is Hamming FEC, with 4 data bits and 1-4 parity bits per codeword. FEC can be thought of as supercharged parity bits. A single parity bit can indicate the presence of an error, but if you use enough of them they can collectively identify and correct errors in place, without re-transmission. Hamming is specifically called out by the European patent, and the code rate parameter referenced throughout reference designs fits nicely within this model. The use of Hamming codes, as opposed to some other FEC or a cyclic coding scheme, was fortuitous because of a property of the Hamming code words. Hamming codeword mapping is deterministic based on the nybble that is being encoded. Four bits of data provide 16 possible codewords. When looking at `Hamming(8,4)` (which is the inferred FEC for LoRa code rate 4/8), 14 of the 16 codewords contain four set bits (`1`s) and four unset bits (`0`s). However, the code words for `0b0000` and `0b1111` are `0b00000000` and `0b11111111`, respectively.

Thus, following on these two assumptions, if a payload containing all `0x00`s or `0xFF`s were sent, then the interleaving and forward error correction should cancel out and not affect the output at all. This *reduces our unknown stages* in the decoding chain from four to just two, with the unknowns being gray indexing and whitening, and once those are resolved then the remaining two can be solved for!

Since "gray indexing" likely refers to gray coding, reverse gray coding, or no coding should it be omitted, this leaves only three permutations to try while solving for the data whitening sequence.

The first step was to take a critical look at the data whitening algorithms provided by Semtech AN1200.18. Given the detail and granularity in which they are described, plus the relevance of having come straight from a LoRa transceiver datasheet, it was almost a given that one of the three algorithms would be the solution. With the interleaver and FEC effectively zeroed out, and "gray indexing" reduced to three possible states, it became possible to test each of the whitening algorithms.

Testing each whitening algorithm was fairly

[24] Manchester's effective bit rate is 1/2 baud rate.

straightforward. A known payload of all `0x00`s or `0xFF`s (to cancel out interleaving and FEC) was transmitted from the Microchip LoRa Technology Mote and then decoded using each whitening algorithm and each of the possible "gray indexing" states. This resulted in 9 total permutations. A visual diff of the decoded data versus the expected payload resulted in no close matches. This was replaced with a diff script with a configurable tolerance for bits that did not match. This also resulted in no matches as well. One final thought was to forward compute the whitening algorithms in case there was a static offset or seed warm-up, as can be the case with other PRNG algorithms. Likewise, this did not reveal any close matches. This meant that either none of the given whitening algorithms in the documentation were utilized, or the assumptions that I made about the interleaver and FEC were not correct.

After writing off the provided whitening algorithms as fiction, the next course of action was to attempt to derive the real whitening algorithm from the LoRa transmitter itself. This approach was based on the previous observations about the FEC and interleaver and a fundamental understanding of how data whitening works. In essence, whitening is as simple as XORing a payload against a static pseudorandom string, with the same string used by both the transmitter and receiver. Since anything XORed with zero is itself, passing in a string of zeroes causes the transmitter to reveal a "gray indexed" version of its whitening sequence.

This payload was received, then transformed into three different versions of itself: one gray-coded, one unmodified, and one reverse gray-coded. All three were then tested by transmitting a set of `0xF` data nybbles and using each of the three "gray indexing" candidates and received whitening sequence to decode the payload. The gray coded and unmodified versions proved to be incorrect, but the reverse gray coding version successfully produced the transmitted nybbles, and thus in one fell swoop, I was able to both derive the whitening sequence and discern that "gray indexing" actually referred to the reverse gray coding operation. With "gray indexing" and whitening solved, I could turn my attention to the biggest challenge: the interleaver.

7.5.2 The Interleaver

At this point we've resolved two of the four signal processing stages, disproving their documentation in the process. Following on this, the validity of the interleaver definition provided in Semtech's patent was immediately called into question.

A quick test was conducted against a local implementation of said interleaver: a payload comprised of a repeated data byte that would produce a `Hamming(8,4)` codeword with four set and four unset bits was transmitted and the de-interleaved frame was inspected for signs of the expected codeword. A few other iterations were attempted, including reversing the diagonal offset mapping pattern described by the patent and using the inverse of the algorithm (i.e., interleaving the received payload rather than de-interleaving it). Indeed, I was able to conclude that the interleaver implemented by the protocol is not the one suggested by the patent. The next logical step is to attempt to reverse it.

Within a transmitter, interleaving is often applied after forward error correction in order to make the packet more resilient to burst interference. Interleaving scrambles the FEC-encoded bits throughout the packet so that if interference occurs it is more likely to damage one bit from many codewords rather than several bits from a single codeword. The former error scenario would be recoverable through FEC, the latter would result in unrecoverable data corruption.

Block-based interleavers, like the one described in the patent, are functionally straightforward. The interleaver itself can be thought of as a two-dimensional array, where each row is as wide as the number of bits in each FEC codeword and the number of columns corresponds to the number of FEC codewords in each interleaver block. The data is then written in row-wise and read out column-wise; thus the first output "codeword" is comprised of the LSB (or MSB) of each FEC codeword. A diagonal interleaver, as suggested in the patent, offsets the column of the bit being read out as rows are traversed.

Understanding the aforementioned fundamentals of what the interleaver was likely doing was essential to approaching this challenge. Ultimately, given that a row-column or row-diagonal relationship defines most block-based interleavers, I anticipated that patterns that could be revealed if approached appropriately. Payloads were therefore constructed to reveal the relationship of each row or codeword with a corresponding diagonal or column. In order to reveal said mapping, the `Hamming(8,4)` codeword for `0xF` was leveraged, since it would fill each row

0x0000000F	0x000000F0	0x00000F00	0x0000F000	0x000F0000	0x00F00000	0x0F000000	0xF0000000
00100011	11000000	00001001	11010000	00000011	01000100	01000001	00001000
00010011	00100101	00000111	00001001	00000011	00000011	10000010	01000101
00001001	00010001	00000011	00000101	01000001	00000000	00100001	10000011
00000111	00001101	00000011	00000110	10000010	01000101	00010010	00100011
00000000	00001100	01000010	00001000	00100010	10001001	00001010	00010011
00000100	00000000	10000001	01000010	00010001	00100010	00000111	00001011
01000011	00000001	00100001	10000000	00001001	00010000	00000011	00000111
10000101	01000111	00010000	00100101	00000000	00001111	00000101	00000111

Figure 13. Symbol Tests

with eight contiguous bits at a time. Payloads consisting of seven `0x0` codewords and one `0xF` codeword were generated, with the nybble position of `0xF` iterating through the payload. See Figure 13.

As one can see, by visualizing the results as they would be generated by the block, patterns associated with each codeword's diagonal mapping can be identified. The diagonals are arbitrarily offset from the corresponding row/codeword position. One important oddity to note is that the most significant bits of each diagonal are flipped.

While we now know how FEC codewords map into block diagonals, we do not know where each codeword starts and ends within the diagonals, or how its bits are mapped. The next step is to map the bit positions of each interleaver diagonal. This is done by transmitting a known payload comprised of FEC codewords with 4 set and 4 unset bits and looking for patterns within the expected diagonal.

```
Payload: 0xDEADBEEF
bit 76543210
     00110011
     10111110
     11111010
     11011101
     10000010
     10000111
     11000000
     10000010
```

Reading out the mapped diagonals results in the following table.

	T							Bot
D	1	0	1	0	0	0	0	1
E	0	1	1	1	0	1	0	0
A	0	1	0	1	1	0	0	0
D	1	0	1	1	0	0	0	0
B	1	1	0	0	0	0	1	0
E	0	1	1	1	0	1	0	0
E	0	1	1	1	0	1	0	0
F	1	1	1	1	1	1	1	1

While no matches immediately leap off the page, manipulating and shuffling through the data begins to reveal patterns. First, reverse the bit order of the extracted codewords:

	B							Top
D	1	0	0	0	0	1	0	1
E	0	0	1	0	1	1	1	0
A	0	0	0	1	1	0	1	0
D	0	0	0	0	1	1	0	1
B	0	1	0	0	0	0	1	1
E	0	0	1	0	1	1	1	0
E	0	0	1	0	1	1	1	0
F	1	1	1	1	1	1	1	1

And then have a look at the last nybble for each of the highlighted codewords:

	B							Top
D	1	0	0	0	0	1	0	1
E	0	0	1	0	**1**	**1**	**1**	**0**
A	0	0	0	1	**1**	**0**	**1**	**0**
D	0	0	0	0	**1**	**1**	**0**	**1**
B	0	1	0	0	0	0	1	1
E	0	0	1	0	**1**	**1**	**1**	**0**
E	0	0	1	0	**1**	**1**	**1**	**0**
F	1	1	1	1	**1**	**1**	**1**	**1**

Six of the eight diagonals resemble the data embedded into each of the expected FEC encoded codewords! As for the first and fifth codewords, it is possible they were damaged during transmission, or that the derived whitening sequence used for those positions is not exact. That is where FEC proves its mettle – applying `Hamming(8,4)` FEC would repair any single bit errors that occurred in transmission. The Hamming parity bits that are expected with each codeword are calculated using the Hamming FEC algorithm, or can be looked up for standard schemes like `Hamming(7,4)` or `Hamming(8,4)`.

```
     Data(8,4) Parity Bits
0xD  1101      1000
0xE  1110      0001
0xA  1010      1010
0xD  1101      1000
0xB  1011      0100
0xE  1110      0001
0xE  1110      0001
0xF  1111      1111
```

While the most standard `Hamming(8,4)` bit order is: p1, p2, d1, p3, d2, d3, d4, p4 (where p are parity bits and d are data bits), after recognizing the above data values we can infer that the parity bits are in a nonstandard order. Looking at the diagonal codeword table and the expected `Hamming(8,4)` encodings together, we can map the actual bit positions:

	Bot							Top
	p1	p2	p4	p3	d1	d2	d3	d4
D	1	0	0	0	0	1	0	1
E	0	0	1	0	1	1	1	0
A	0	0	0	1	1	0	1	0
D	0	0	0	0	1	1	0	1
B	0	1	0	0	0	0	1	1
E	0	0	1	0	1	1	1	0
E	0	0	1	0	1	1	1	0
F	1	1	1	1	1	1	1	1

Note that parity bits three and four are swapped. With that resolved, we can use the parity bits to decode the forward error correction, resulting in four bits being corrected, as shown in Figure 14.

That's LoRa!

– — — – — — — — – — –

Having reversed the protocol, it is important to look back and reflect on how and why this worked. As it turned out, being able to make assumptions and inferences about certain goings-on was crucial for bounding the problem and iteratively verifying components and solving for unknowns. Recall that by effectively canceling out interleaving and forward error correction, I was able to effectively split the problem in two. This enabled me to solve for whitening, even though "gray indexing" was unknown there were only three permutations, and with that in hand, I was able to solve for the interleaver, since FEC was understood to some extent. Just like algebra or any other scientific inquiry, it comes down to controlling your variables. By stepping through the problem methodically and making the right inferences, we were able to reduce 4 independent variables to 1, solve for it, and then plug that back in and solve for the rest.

7.6 Remaining Work

While the aforementioned process represents a comprehensive description of the PHY, there are a few pieces that will be filled in over time.

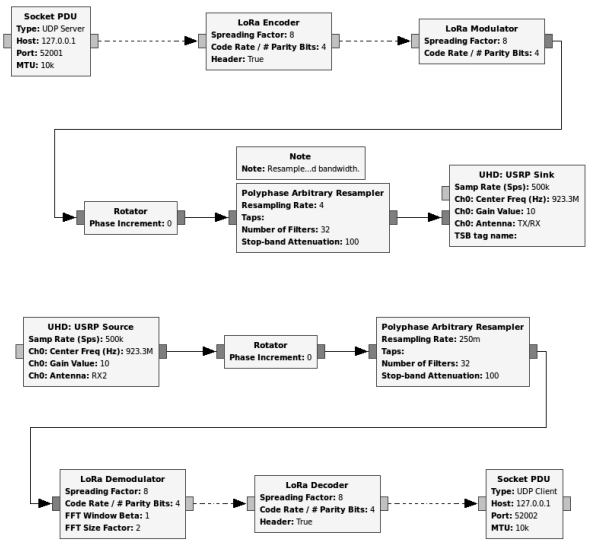
The LoRa PHY contains an optional header with its own checksum. I have not yet reversed the header, and the Microchip LoRa module I've used to generate LoRa traffic does not expose the option of disabling the header. Thus I cannot zero those bits out to calculate the whitening sequence applied to it. It should be straightforward to fill in with the correct hardware in hand.

The PHY header and service data unit/payload CRCs have not been investigated for the same reason. This should be easy to resolve through the use of a tool like CRC RevEng once the header is known.

In my experience, for demodulation purposes clock recovery has not been necessary beyond getting an accurate initial sync on the SFD. However should clock drift pose a problem, for example if transmitting longer messages or using higher spreading factors which have slower data rates/longer over-the-air transmission times, clock recovery may be desirable.

7.7 Shameless Plug

I recently published an open source GNU Radio OOT module that implements a transceiver based on this derived version of the LoRa PHY. It is presented to empower RF and security researchers to investigate this nascent protocol.[25]



[25] `git clone https://github.com/BastilleResearch/gr-lora`
`unzip pocorgtfo13.pdf gr-lora.tar.bz2`

									Top
	p1	p2	p4	p3	d1	d2	d3	d4	
D	1	0	0	0	**1**	1	0	1	1101 = 0xD
E	0	0	1	0	1	1	1	0	1110 = 0xE
A	**1**	0	0	1	1	0	1	0	1010 = 0xA
D	**1**	0	0	0	1	1	0	1	1101 = 0xD
B	0	1	0	0	**1**	0	1	1	1011 = 0xB
E	0	0	1	0	1	1	1	0	1110 = 0xE
E	0	0	1	0	1	1	1	0	1110 = 0xE
F	1	1	1	1	1	1	1	1	1111 = 0xF

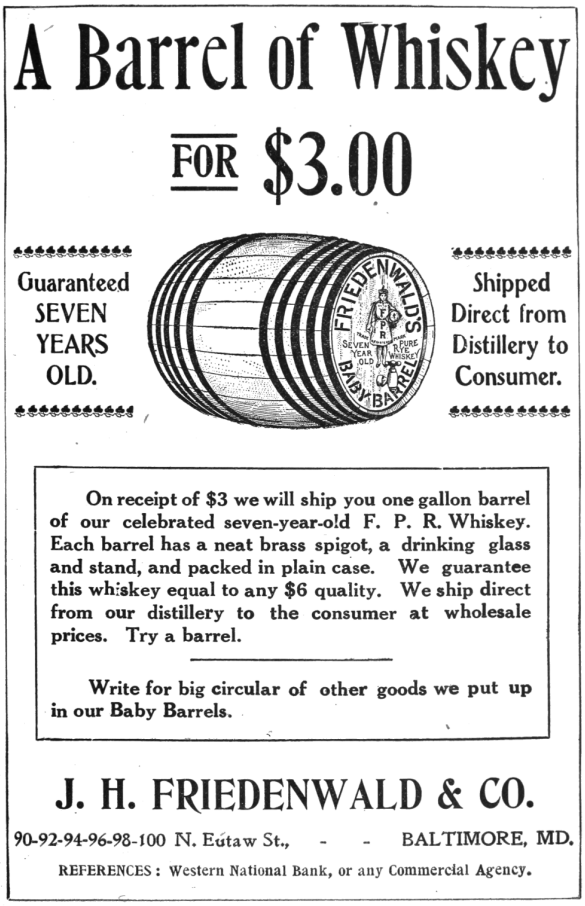
Figure 14. Forward Error Corrected bits shown in bold

7.8 Conclusions and Key Takeaways

Presented here is the process that resulted in a comprehensive deconstruction of the LoRa PHY layer, and the details one would need to implement the protocol. Beyond that, however, is a testament to the challenges posed by red herrings (or three of them, all at once) encountered throughout the reverse engineering process. While open source intelligence and documentation can be a boon to researchers – and make no mistake, it was enormously helpful in debunking LoRa – one must remember that even the most authentic sources may sometimes lie!

Another point to take away from this is the importance of bounding problems as you solve them, including through making informed inferences in the absence of perfect information. This of course must be balanced with the first point about OSINT, is knowing when to walk away from a source. However as illustrated above, drawing appropriate conclusions proved integral to reducing and solving for each of the decoding elements within a black-box methodology.

The final thought I will leave you with is that wireless doesn't just mean Wi-Fi anymore - it includes cellular, PANs, LPWANs, and everything in between. Accordingly, a friendly reminder that security monitoring and test tools don't exist until someone creates them. Monitor mode and Wireshark weren't always a thing, so don't take them for granted: it's time to make the next generation of wireless networks visible to researchers, because know it or not it is already here and is here to stay.



8 Plumbing, not Popper; or, the Problem with STEP

by Pastor Manul Laphroaig

Gather round, neighbors. We are going to a magical place. One that we hardly ever notice in our busy lives, but which has a way of taking over your entire day when you are forced to visit it. We are going on a trip to the plumbing closet![26]

Look at the miracle that is the clump of pipes, looking right back at you. Its message is clear: *do not approach without skill*, unless you *like* wet, gigantic messes. This message is universal: it speaks to a politician, a professor, an NYT columnist, a movie actor, and a hedge fund manager alike. It transcends languages and beliefs.

Even though these worthies and civic leaders might agree the country could use more plumbers, it has not yet occurred to them to approach the problem by putting a big P into some popular slogan like "STEP" (Science, Technology, Engineering, Plumbing), by setting up a federal Department of Plumbing, or by lionizing a professional coveralls-wearer TV personality who goes by "A Plumbing Guy," despite never having fixed a pipe in his life.

They somehow know that these things will do diddly squat to address the shortage of plumbers. They know deep down that to learn plumbing—and even to not sound ridiculous about it—one needs to study with a plumber, attach oneself to a plumber, and do what a plumber does for a while. This, neighbors, is how deep the plumbing magic goes.

Science, alas, has not been so lucky.

– — — – — — — – — — –

It is fashionable to talk about how we need more scientists, and how we can direct and improve science, quoting grand theories that explain science, while similarly educated people nod approvingly. After all, they all know what science is, as befits all forward-thinking people these days. No one feels awkward; everyone feels good.

Perhaps this happens because our social betters all experienced helplessness at the sight of broken plumbing, but would not recognize broken science, much less a hopelessly broken science textbook. You see, science lab equipment is OK with a patronizing, self-satisfied gaze, whereas plumbing has a way of glaring back contemptuously, daring you to use your general theoretical understanding.

With plumbing, it's either practical skill or a huge mess in your basement. Messing with how plumbers learn and teach this skill guarantees messes in thousands of basements. If you value your plumbing, it's wise to leave plumbers alone even if you believe every word of every newspaper column you've ever read on plumbing economy.

It may be a surprise to the readers of Karl Popper and Imre Lakatos[27] that actual scientists are helped by philosophy of science in exactly the same way as plumbers are helped by the Zen of Plumbing. Although these very same people are likely to believe they understand plumbing too, they usually have the sense to leave the plumbing profession well alone, and not apply their philosophical understandings to it—being empirically familiar with the fact that when you need plumbing done, philosophy is useless; only the skill stands between the water in your pipes and your expensive library.

– — — – — — — – — — –

[26] For those of you fortunate to own a house, it's probably in the corner of your basement, an equally magical place, whence all science and innovation springs forth—but let us not digress.

[27] Lakatos the philosopher is considered to be a great intellectual authority. For what it's worth, you might also want to read about how he applied his philosophy in real life: `unzip pocorgtfo13 freudenthal.pdf`

PLUMBERS' TOOLS

Plate 2211
Tap Borer. Price, per Doz. **$6.00**

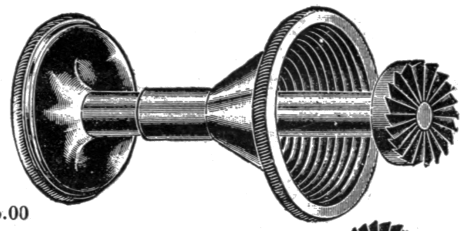


Plate 2213
Shave Hook.
Price, per Doz.... **$6.00**

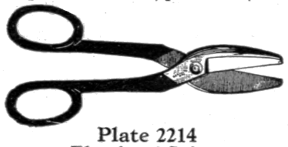


Plate 2214
Plumbers' Snips

No. 8	3½″	Cut Each..	**$3.50**
No. 9	3	"	**3.00**
No. 10	2½″	"	**2.50**

Plate 2212
Bibb reseating Tool with Cutters
For ⅜″-½″-⅝″-¾″ Bibbs.
Price, Each..... **$5.00**
Extra Cutters, per set..... **2.50**

Plate 2215
Bench Mallet.
Price, per Doz..... **$6.00**

Plate 2217
Lead Pipe Bending Spring

Sizes	1″	1¼″	1½″	2″
Price, Each.....	**$1.25**	**$1.50**	**$1.75**	**$2.00**

Plate 2216
Burner Pliers

Sizes	5″	6″	7″
Price, Each.	**$ .75**	**$1.00**	**$1.25**

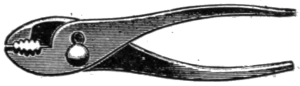


Plate 2219— Combination Pliers

Sizes	6″	8″	10″
Nickel Plated..	**$ 1.50**	**$1.75**	**$2.00**
Polished Steel..	**1.25**	**1.50**	**1.75**
Blue finished Steel....	**1.00**	....	....

Plate 22110
Steel Bend Iron
Price, Each..... **$ .75**

Plate 2218
Boxwood Lead Dresser
Price, per Doz..... **$15.00**

Plate 21111
" Rivetting Hammer "

Sizes	1″	2″	3″
Price.....	**$1.50**	**$1.25**	**$1.00**

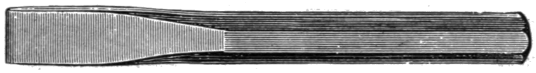


Plate 22112

	½″	⅝″
Cold Chisel.....		
Price, Each.....	**$ .50**	**$ .75**

Plate 22113—Capé Chisel
Price, Each, ¾″..... **$ .50**

Plate 22114
Straight Caulking Chisel
Price, Each..... **$ .75**

Plate 22115
Picking Chisel
Price, Each..... **$ .75**

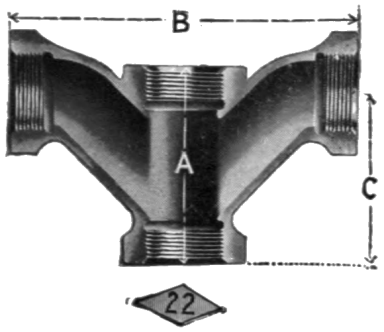
Plate 22116
Regular Caulking Chisel
Size, ¾″ Price, Each.... **$ .75**

Plate 22117
Long Packing Iron
Size, 18″. Price, Each.... **$ .75**

Plate 22118
R and L Hand Caulking Chisel
Price, Each..... **$1.00**

Plate 22119
Round nose Pliers,
Stocked from 4″ to 8″

LONG TURN 90° DOUBLE T.Y.'S



By far the worst hit to a profession is delivered when a part of the professionals actually welcomes philosophers lauding it, politicians bearing gifts and grants, and governments setting up departments to promote it. Forms to fill, ever-growing grant application paperwork, pervasive "performance metrics," and having to explain basic fallacies to the well-meaning but fundamentally ignorant and hugely powerful committees come later—and accumulate. In the context of metrics, charlatans always win, because they don't get distracted by trying for actual results.

Not to mention that the money that goes to charlatans is not net-neutral for actual plumbing (or science); it is net-negative, because charlatans have a way of making the lives of professionals hard where it hurts the most. When Tim "the Tool Man" Taylor waves power tools around with a swagger, the results are immediate and obvious. When learned committees do the professional equivalent thereof to math textbooks and call it nice names like "Discovery Math," "Common Core," or "Critical Thinking" it takes a generation to notice, and then we wonder—how on earth did school math become unteachable and unlearnable?[28]

Plumbers have wisely avoided it, perhaps due to some secret wisdom passed from master to apprentice through the ages. Scientists, I am sorry to say, walked right into it around the middle of the twentieth century.

Sure enough, national agencies got us to the moon—but it seems that all the good science schoolbooks have been put on the rockets going there, never to return. Have you met many scientists who are happy with what schools do to their sciences after half a century of being improved by various government offices?

Funny how it worked out for scientists. Now hear them complain about "publish or perish," the rapidly rising age at which one finally succeeds in getting one's first grant, and the relentless race to rebrand and follow the current big-ticket grant programs.[29]

But don't blame them, neighbors; it was their advisors or their advisors' advisors who fell for it. Better to buy them a drink, and remember their lesson.

Better yet, find some plumbers, and buy them drinks. Perhaps they'll share with you some of their secrets of how to keep the philosophers and their educated and benevolent readers interested in the result, but at a safe distance from the actual plumbing.

[28] We sort of know the answer, neighbors: a roller coaster of reforms and unintelligible standards created a generation of math teachers for whom math did not have to make sense. `unzip pocorgtfo13.pdf wu-preparing-teachers.pdf` and read it. It may apply to whatever else you hold dear.

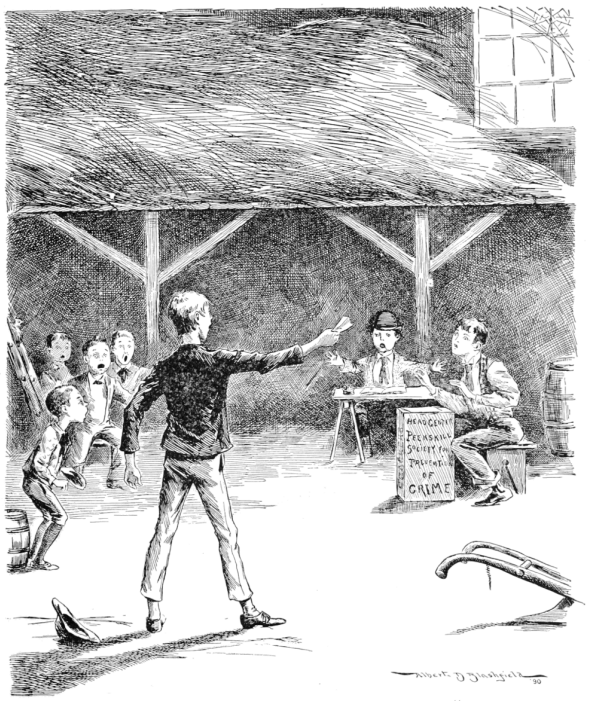
[29] According to Ronald J. Daniels, President of Baltimore's Johns Hopkins University, no less than the whole generation is at risk: "A generation at risk: Young investigators and the future of the biomedical workforce." (`unzip pocorgtfo13.pdf atrisk.pdf`.) For more of this, read "Science in the Age of Selfies" by Donald Geman, Stuart Geman. (`selfies.pdf`.) It's hard to make these things up, neighbors.

9 Where is ShimDBC.exe?

by Geoff Chappell

Microsoft's Shim Database Compiler might be a legend ... except that nobody seems ever to have made any story of it. It might be mythical ... except that it actually *does* exist. Indeed, it has been around for 15 years in more or less plain sight. Yet if you ask Google to search the Internet for occurrences of `shimdbc`, and especially of "`shimdbc.exe`" in quotes, you get either remarkably little or a tantalising hint, depending on your perspective.

Mostly, you get those scam sites that have prepared a page for seemingly every executable that has ever existed and can fix it for you if only you will please download their repair tool. But amongst this dross is a page from Microsoft's TechNet site. Google excerpts that "QFixApp uses the support utility ShimDBC.exe to test the group of selected fixes." Follow the link and you get to one of those relatively extensive pages that Microsoft sometimes writes to sketch a new feature for system administrators and advanced users (if not also to pat themselves on the back for the great new work). This page is from 2001 and is titled *Windows XP Application Compatibility Technologies*.[30]



[30] `https://technet.microsoft.com/library/bb457032.aspx`

9.1 Application Compatibility?

There can't be anything more boring in the whole of Windows, you may think. I certainly used to, and might still for applications if I cared enough, but Windows 8 brought *Application Compatibility* to kernel mode in a whole new way, and this I *do* care about.

The integrity of any kernel-mode driver that you or I write nowadays depends on what anyone else, well-meaning or not, can get into the `DRVMAIN.SDB` file in the `AppPatch` subdirectory of the Windows installation. This particular Shim Database file exists in earlier Windows versions too, but only to list drivers that the kernel is not to load. If you're the writer of a driver, there's nothing you can do at runtime about your driver being blocked from loading, and in some sense you're not even affected: you're not loaded and that's that. Starting with Windows 8, however, the `DRVMAIN.SDB` file defines the installed shim providers and either the registry or the file can associate your driver with one or more of these defined shim providers. When your driver gets loaded, the applicable shim providers get loaded too, if they are not already, and before long your driver's image in memory has been patched, both for how it calls out through its Import Address Table and how it gets called, *e.g.*, to handle I/O requests.

In this brave new world, is your driver really your driver? You might hope that Microsoft would at least give you the tools to find out, if only so that you can establish that a reported problem with your driver really is with your driver. After all, for the analogous shimming, patching, and whatever of applications, Microsoft has long provided an Application Compatibility Toolkit (ACT), recently re-branded as the Windows Assessment and Deployment Kit (ADK). The plausible thoroughness of this kit's Compatibility Administrator in presenting a tree view of the details is much of the reason that I, for one, regarded the topic as offering, at best, slim pickings for research. For the driver database, however, this kit does nothing—well, except to leave me thinking that the SDB file format and the API support through which SDB files get interpreted, created, and might be edited, are now questions I should want to answer for myself rather than imag-

ine they've already been answered well by whoever managed somehow to care about Application Compatibility all along.

9.2 The SDB File Format

Relax! I'm not taking you to the depths of Application Compatibility, not even just for what's specific to driver shims. Our topic here *is* reverse engineering. Now that you know what these SDB files are and why we might care to know what's in them, I expect that if you have no interest at all in Application Compatibility, you can treat this part of this article as using SDB files just as an example for some general concerns about how we present reverse-engineered file formats. (And please don't skip ahead, but I promise that the final part is pretty much nothing but ugly hackery.)

Let's work even more specifically with just one example of an SDB file, shown in Figure 15. It's a little long, despite being nearly minimal. It defines one driver shim but no drivers to which this shim is to be applied.

Although Microsoft *has not* documented the SDB file format, Microsoft *has* documented a selection of API functions that work with SDB files, which is in some ways preferable. Perhaps by looking at these functions researchers and reverse engineers have come to know at least something of the file format, as evidenced by various tools they have published which interpret SDB files one way or another, typically as XML.

As a rough summary, an SDB file has a 3-dword header, for a major version, minor version, and signature, and the rest of the file is a list of variable-size tags which each have three parts:

1. a 16-bit `TAG`, whose numerical value tells of the tag's type and purpose;

2. a size in bytes, which can be given explicitly as a dword or may be implied by the high 4 bits of the `TAG`;

3. and then that many bytes of data, whose interpretation depends on the `TAG`.

Importantly for the power of the file format, the data for some tags (the ones whose high 4 bits are 7) is itself a list of tags. From this summary and a few details about the recognised `TAG` values, the implied sizes and the general interpretation of the data, *e.g.*, as word, dword, binary, or Unicode string—all of which can be gleaned from Microsoft's admittedly terse documentation of those API functions—you might think to reorganise the raw dump so that it retains every byte but more conveniently shows the hierarchy of tags, each with their `TAG`, size (if explicit) and data (if present). A decoding of Figure 15 is shown in Figure 16.

To manually verify that everything in the file is exactly as it should be, there is perhaps no better representation to work from than one that retains every byte. In practice, though, you'll want some interpretation. Indeed, the dump above does this already for the tags whose high 4 bits are 6. The data for any such tag is a string reference, specifically the offset of a `0x8801` tag within the `0x7801` tag (at offset `0x0142` in this example), and an automated dump can save you a little trouble by showing the offset's conversion to the string. Since those numbers for tags soon become tedious, you may prefer to name them. The names that Microsoft uses in its programming are documented for the roughly 100 tags that were defined ten years ago (for Windows Vista). All tags, documented or not (and now running to 260), have friendly names that can be obtained from the API function `SdbTagToString`. If you haven't suspected all along that Microsoft prepares SDB files from XML input, then you'll likely take "tag" as a hint to represent an SDB file's tags as XML tags. And this, give or take, is where some of the dumping tools you can find on the Internet leave things, such as in Figure 17.

Notice already that choices are made about what to show and how. If you don't show the offset in bytes that each XML tag has as an SDB tag in the original SDB file, then you risk complicating your presentation of data, as with the string references, whose interpretation depends on those file offsets. But show the offsets and your XML quickly looks messy. Once your editorial choices go so far that you don't reproduce every byte but instead build more and more interpretation into the XML, why show every tag? Notably, the string table that's the data for tag `0x7801` (`TAG_STRINGTABLE`) and the indexes that are the data for tag `0x7802` (`TAG_INDEXES`) must be generated automatically from the data for tag `0x7001` (`TAG_DATABASE`) such that the last may be all you want to bother with. Observe that for any tag that has children, the subtags that don't have children come first, and perhaps you'll plumb for a different style of XML in which each tag that has no

```
00000000:  02 00 00 00 01 00 00 00-73 64 62 66 02 78 CA 00  ........sdbf.x..
00000010:  00 00 03 78 14 00 00 00-02 38 07 70 03 38 01 60  ...x.....8.p.8.‘
00000020:  16 40 01 00 00 00 01 98-00 00 00 00 03 78 0E 00  .@..........x..
00000030:  00 00 02 38 17 70 03 38-01 60 01 98 00 00 00 00  ...8.p.8.‘......
00000040:  03 78 0E 00 00 00 02 38-07 70 03 38 04 90 01 98  .x.....8.p.8....
00000050:  00 00 00 00 03 78 14 00-00 00 02 38 1C 70 03 38  .....x.....8.p.8
00000060:  01 60 16 40 02 00 00 00-01 98 00 00 00 00 03 78  .‘.@...........x
00000070:  14 00 00 00 02 38 1C 70-03 38 0B 60 16 40 02 00  .....8.p.8.‘.@..
00000080:  00 00 01 98 00 00 00 00-03 78 14 00 00 00 02 38  .........x.....8
00000090:  1A 70 03 38 01 60 16 40-02 00 00 00 01 98 00 00  .p.8.‘.@........
000000A0:  00 00 03 78 14 00 00 00-02 38 1A 70 03 38 0B 60  ...x.....8.p.8.‘
000000B0:  16 40 02 00 00 00 01 98-00 00 00 00 03 78 1A 00  .@...........x..
000000C0:  00 00 02 38 25 70 03 38-01 60 01 98 0C 00 00 00  ...8%p.8.‘......
000000D0:  00 00 52 45 4B 43 41 48-14 01 00 00 01 70 60 00  ..REKCAH.....p‘.
000000E0:  00 00 01 50 D8 C1 31 3C-70 10 D2 01 22 60 06 00  ...P..1<p..."‘..
000000F0:  00 00 01 60 1C 00 00 00-23 40 01 00 00 00 07 90  ...‘....#@......
00000100:  10 00 00 00 28 22 AB F9-12 33 73 4A B6 F9 93 6D  ....("...3sJ...m
00000110:  70 E1 12 EF 25 70 28 00-00 00 01 60 50 00 00 00  p...%p(....‘P...
00000120:  10 90 10 00 00 00 C8 E4-9C 91 69 D0 21 45 A5 45  ..........i.!E.E
00000130:  01 32 B0 63 94 ED 17 40-03 00 00 00 03 60 64 00  .2.c...@.....‘d.
00000140:  00 00 01 78 7A 00 00 00-01 88 10 00 00 00 32 00  ...xz.........2.
00000150:  2E 00 31 00 2E 00 30 00-2E 00 33 00 00 00 01 88  ..1...0...3.....
00000160:  2E 00 00 00 48 00 61 00-63 00 6B 00 65 00 64 00  ....H.a.c.k.e.d.
00000170:  20 00 44 00 72 00 69 00-76 00 65 00 72 00 20 00  .D.r.i.v.e.r. .
00000180:  44 00 61 00 74 00 61 00-62 00 61 00 73 00 65 00  D.a.t.a.b.a.s.e.
00000190:  00 00 01 88 0E 00 00 00-48 00 61 00 63 00 6B 00  ........H.a.c.k.
000001A0:  65 00 72 00 00 00 01 88-16 00 00 00 68 00 61 00  e.r.........h.a.
000001B0:  63 00 6B 00 65 00 72 00-2E 00 73 00 79 00 73 00  c.k.e.r...s.y.s.
000001C0:  00 00 ..
```

Figure 15. ShimDB File

child tags is represented as an attribute and value, *e.g.*,

```
<DATABASE
    TIME="0x01D210703C31C1D8"
    COMPILER_VERSION="2.1.0.3"
    NAME="Hacked Driver Database"
    OS_PLATFORM="0x00000001"
    DATABASE_ID="0x28 0x22 0xAB 0xF9 0x12 0x33
    0x73 0x4A 0xB6 0xF9 0x93 0x6D 0x70 0xE1 0
    x12 0xEF">
    <KSHIM
        NAME="Hacker"
        FIX_ID="0xC8 0xE4 0x9C 0x91 0x69 0xD0 0
    x21 0x45 0xA5 0x45 0x01 0x32 0xB0 0x63 0
    x94 0xED"
        FLAGS="0x00000003"
        MODULE="hacker.sys" />
</DATABASE>
```

Whether you choose XML in this style or to have every tag's data between opening and closing tags, there are any number of ways to represent the data for each tag. For instance, once you know that the binary data for tag `0x9007` (`TAG_DATABASE_ID`) or tag `0x9010` (`TAG_FIX_ID`) is always a `GUID`, you might more conveniently represent it in the usual string form. Instead of showing the data for tag `0x5001` (`TAG_TIME`) as a raw qword, why not show that you know it's a Windows `FILETIME` and present it as 16/09/2016 23:15:37.944? Or, on the grounds that it too must be generated automatically, you might decide not to show it at all!

If I labour the presentation, it's to make the point that what's produced by any number of dumping tools inevitably varies according to purpose and taste. Let's say a hundred researchers want a tool for the easy reading of SDB files. Yes, that's doubtful, but 100 is a good round number. Then ninety will try to crib code from someone else—because, you know, who wants to reinvent the wheel—and what you get from the others will each be different, possibly very different, not just for its output but especially for what the source code shows of the file format. Worse, because nine out of ten programmers don't bother much with commenting, even for a tool they may intend as showing off their coding skills, you may have to pick through the source code to extract the file format. That may be easier than reverse-engineering Microsoft's binaries that work with the file, but not necessarily by much—and not necessarily leaving you with the same confidence that what you've learnt about the file format is cor-

```
00000000:  Header: MajorVersion=0x00000002 MinorVersion=0x00000001 Magic=0x66626473
0000000C:  Tag=0x7802 Size=0x000000CA Data=
00000012:      Tag=0x7803 Size=0x00000014 Data=
00000018:          Tag=0x3802 Data=0x7007
0000001C:          Tag=0x3803 Data=0x6001
00000020:          Tag=0x4016 Data=0x00000001
00000026:          Tag=0x9801 Size=0x00000000
0000002C:      Tag=0x7803 Size=0x0000000E Data=
00000032:          Tag=0x3802 Data=0x7017
00000036:          Tag=0x3803 Data=0x6001
0000003A:          Tag=0x9801 Size=0x00000000
00000040:      Tag=0x7803 Size=0x0000000E Data=
     ⋮

000000BC:      Tag=0x7803 Size=0x0000001A Data=
000000C2:          Tag=0x3802 Data=0x7025
000000C6:          Tag=0x3803 Data=0x6001
000000CA:          Tag=0x9801 Size=0x0000000C Data=0x00 0x00 0x52 0x45 0x4B 0x43 0x41 0x48 0x14 0x01 0x00 0x00
000000DC:  Tag=0x7001 Size=0x00000060
000000E2:      Tag=0x5001 Data=0x01D210703C31C1D8
000000EC:      Tag=0x6022 Data=0x00000006 => L"2.1.0.3"
000000F2:      Tag=0x6001 Data=0x0000001C => L"Hacked Driver Database"
000000F8:      Tag=0x4023 Data=0x00000001
000000FE:      Tag=0x9007 Size=0x00000010 Data=0x28 0x22 0xAB 0xF9 0x12 0x33 0x73 0x4A 0xB6 0xF9 0x93 0x6D
                                                0x70 0xE1 0x12 0xEF
00000114:      Tag=0x7025 Size=0x00000028
0000011A:          Tag=0x6001 Data=0x00000050 => L"Hacker"
00000120:          Tag=0x9010 Size=0x00000010 Data=0xC8 0xE4 0x9C 0x91 0x69 0xD0 0x21 0x45 0xA5 0x45 0x01 0x32
                                                     0xB0 0x63 0x94 0xED
00000136:          Tag=0x4017 Data=0x00000003
0000013A:          Tag=0x6003 Data=0x00000064 => L"hacker.sys"
00000142:  Tag=0x7801 Size=0x0000007A Data=
00000148:      Tag=0x8801 Size=0x00000010 Data=L"2.1.0.3"
0000015E:      Tag=0x8801 Size=0x0000002E Data=L"Hacked Driver Database"
00000192:      Tag=0x8801 Size=0x0000000E Data=L"Hacker"
000001A6:      Tag=0x8801 Size=0x00000016 Data=L"hacker.sys"
```

Figure 16. ShimDB File (Decoded from Figure 15)

```
<INDEXES>
  <INDEX>
    <INDEX_TAG>0x7007</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_FLAGS>0x00000001</INDEX_FLAGS>
    <INDEX_BITS></INDEX_BITS>
  </INDEX>
  <INDEX>
    <INDEX_TAG>0x7017</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_BITS></INDEX_BITS>
  </INDEX>
  ...
  <INDEX>
    <INDEX_TAG>0x7025</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_BITS>0x00 0x00 0x52 0x45 0x4B 0x43 0x41 0x48 0x14 0x01 0x00 0x00</INDEX_BITS>
  </INDEX>
</INDEXES>
<DATABASE>
  <TIME>0x01D210703C31C1D8</TIME>
  <COMPILER_VERSION>0x00000006</COMPILER_VERSION>
  <NAME>0x0000001C</NAME>
  <OS_PLATFORM>0x00000001</OS_PLATFORM>
  <DATABASE_ID>0x28 0x22 0xAB 0xF9 0x12 0x33 0x73 0x4A 0xB6 0xF9 0x93 0x6D 0x70 0xE1 0x12 0xEF</
    DATABASE_ID>
  <KSHIM>
    <NAME>0x00000050</NAME>
    <FIX_ID>0xC8 0xE4 0x9C 0x91 0x69 0xD0 0x21 0x45 0xA5 0x45 0x01 0x32 0xB0 0x63 0x94 0xED</
    FIX_ID>
    <FLAGS>0x00000003</FLAGS>
    <MODULE>0x00000064</MODULE>
  </KSHIM>
</DATABASE>
<STRINGTABLE>
  <STRINGTABLE_ITEM>2.1.0.3</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>Hacked Driver Database</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>Hacker</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>hacker.sys</STRINGTABLE_ITEM>
</STRINGTABLE>
```

Figure 17. Illegible XML From a ShimDB Dumping Tool

rect and comprehensive. Writing a tool that dumps an undocumented file format may be more rewarding for you as a programmer but it is not nearly the same as documenting the file format.

9.3 Reversing XML to SDB

But is there really no definitive XML for representing SDB files? Of all the purposes that motivate anyone to work with SDB files closely enough to need to know the file format, one has special standing: Microsoft's creation of SDB files from XML input. If we had Microsoft's tool for that, then wouldn't most researchers plumb for reversing its work to recover the XML source? After all, most reverse engineers and certainly the popular reverse-engineering tools don't take binary code and unassemble it just to what you see in the debugger. No, they disassemble it into assembly language that can be edited and re-assembled. Many go further and try to decompile it into C or C++ that can be edited and re-compiled (even if it doesn't look remotely like anything you'd be pleased to have from a human programmer). In this context, the SDB to XML conversion to want is something you could feed to Microsoft's Shim Database Compiler for compilation back to SDB. Anything else is pseudo-code. It may be fine in its way for understanding the content, and some may prefer it to a raw dump interpreted with reference to documentation of the file format, but however widely it gets accepted it is nonetheless pseudo-code.

The existence of something that someone at Microsoft refers to as a Shim Database Compiler has been known for at least a decade because Microsoft's documentation of tag `0x6022` (`TAG_COMPILER_VERSION`), apparently contemporaneous with Windows Vista, describes this tag's data as the "Shim Database Compiler version." And what, then, is the `ShimDBC.exe` from the even older TechNet article if it's not this Shim Database Compiler?

But has anyone outside Microsoft ever seen this compiler? Dig out an installation disc for Windows XP from 2001, look in the Support Tools directory, install the ACT version 2.0 from its self-extracting executable, and perhaps install the Support Tools too in case that's what the TechNet article means by "support utility." For your troubles, which may include having to install Windows XP, you'll get the article's `QFixApp.exe`, and the Compatibility Administrator, as `CompatAdmin.exe`, and some other possibly useful or at least instructive tools such as `GrabMI.exe`, but you don't get any file named `ShimDBC.exe`. I suspect that `ShimDBC.exe` never has existed in public as any sort of self-standing utility or even as its own file. Even if it did once upon a time, we should want a modern version that knows the modern tags such as `0x7025` (`TAG_KSHIM`) for defining driver shims.

For some good news, look into either `QFixApp.exe` or `CompatAdmin.exe` using whatever is your tool of choice for inspecting executables. Inside each, not as resources but intermingled with the code and data, are several instances of ShimDBC as text. We've had Microsoft's Shim Database Compiler for 15 years since the release of Windows XP. All along, the code and data for the console program `ShimDBC.exe`, from its `wmain` function inwards, has been linked into the GUI programs `QFixApp.exe` and `CompatAdmin.exe` (of which only the latter survives to modern versions of the ACT). Each of the GUI programs has a `WinMain` function that's first to execute after the C Run-Time (CRT) initialisation. Whenever either of the GUI programs wants to create an SDB file, it composes the Unicode text of a command line for the fake `ShimDBC.exe` and calls a routine that first parses this into the `argc` and `argv` that are expected for a `wmain` function and which then simply calls the `wmain` function. Where the TechNet article says QFixApp *uses* `ShimDBC.exe`, it is correct, but it doesn't mean that QFixApp executes `ShimDBC.exe` as a separate program, more that QFixApp simulates such execution from the ShimDBC code and data that's built in.

Unfortunately, CompatAdmin does not provide, even in secret, for passing a command line of our choice through `WinMain` to `wmain`. But, c'mon, we're hackers. You'll already be ahead of me: we can patch the file. Make a copy of `CompatAdmin.exe` as `ShimDBC.exe`, and use your favourite debugger or disassembler to find three things:

- the program's `WinMain` function;
- the routine the program passes the fake command line to for parsing and for calling `wmain`;
- the address of the Import Address Table entry for calling the `GetCommandLineW` function.

Ideally, you might then assemble something like

```
call    dword ptr [__imp__GetCommandLineW@0]
mov     ecx,eax
call    SimulateShimDBCExecution
ret     10h
```

over the very start of `WinMain`. In practice, you have to allow for relocations. Our indirect call to `GetCommandLineW` will need a fixup if the program doesn't get loaded at its preferred address. Worse, if we overwrite any fixup sites in `WinMain`, then our code will get corrupted if fixups get applied. But these are small chores that are bread and butter for practised reverse engineers. For concreteness, I give the patch details for the 32-bit `CompatAdmin.exe` from the ACT version 6.1 for Windows 8.1 in Table 2.

For hardly any trouble, we get an executable that still contains all its GUI material (except for the 17 bytes we've changed) but never executes it and instead runs the console-application code with the command line that we give when running the patched program. Microsoft surely has `ShimDBC.exe` as a self-standing console application, but what we get from patching `CompatAdmin.exe` must be close to the next best thing, certainly for so little effort. It's still a GUI program, however, so to see what it writes to standard output we must explicitly give it a standard output. At a Command Prompt with administrative privilege, enter

```
shimdbc -?  >help.txt
```

to get the built-in ShimDBC program's mostly accurate description of its command-line syntax, including most of the recognised command-line options.

To produce the SDB file that is this article's example, write the following as a Unicode text file named `test.xml`:

```
<?xml version="1.0" encoding="UTF-16" ?>
<DATABASE NAME="Hacked Driver Database"
 ID="{F9AB2228-3312-4A73-B6F9-936D70E112EF}">
  <LIBRARY>
    <KSHIM NAME="Hacker" FILE="hacker.sys"
     ID="{919CE4C8-D069-4521-A545-0132B06394ED}
     "
     LOGO="YES" ONDEMAND="YES" />
  </LIBRARY>
</DATABASE>
```

and feed it to the compiler via the command line

```
shimdbc Driver test.xml test.sdb >test.txt
```

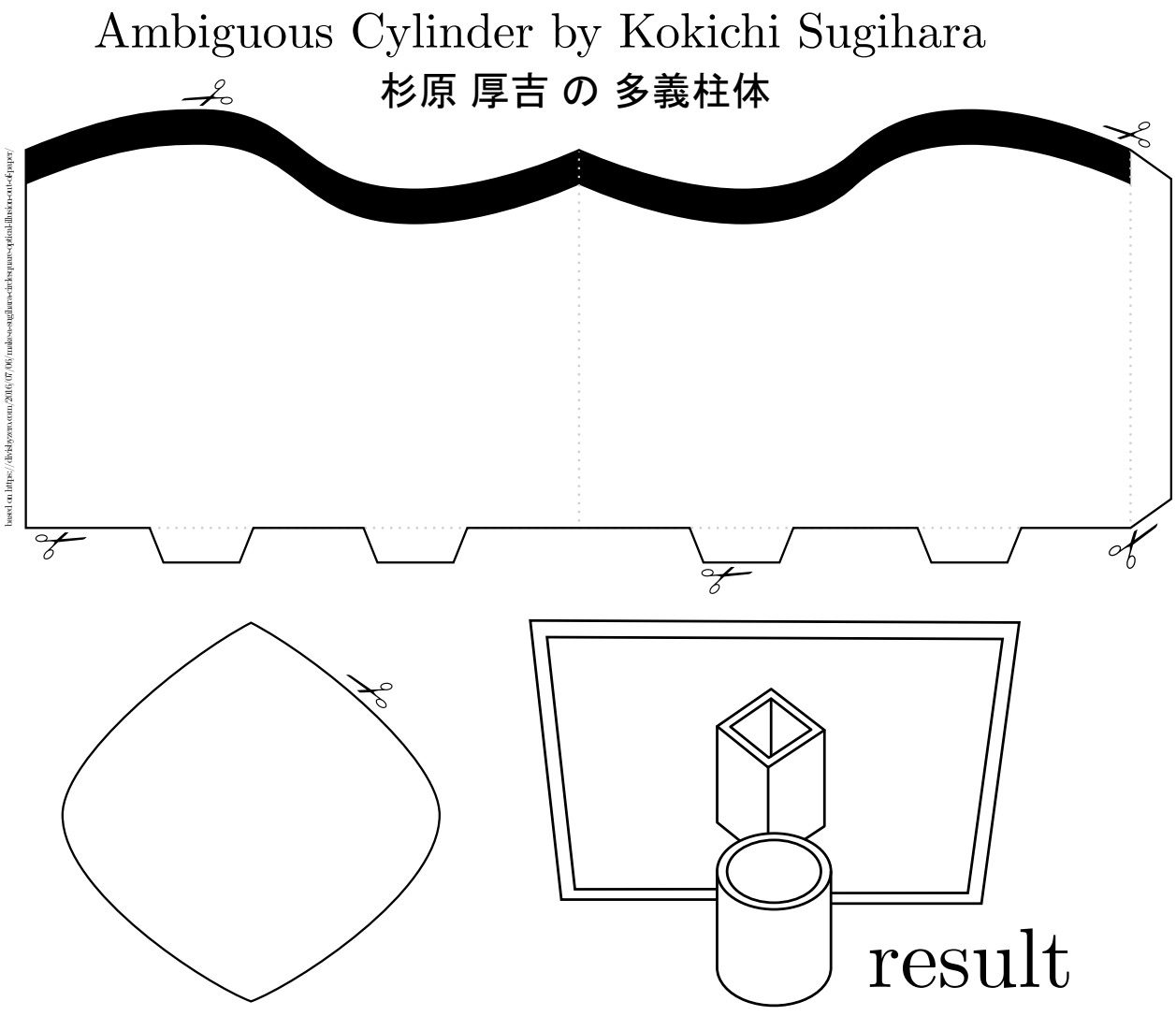
I may be alone in this, but if you're going to tell me that I should know that you know the SDB file format when all you have to show is a tool that converts SDB to XML, then this would better be the XML that your tool produces from this article's example of an SDB file. Otherwise, as far as I'm concerned for studying any SDB file, I'm better off with a raw dump in combination with actual documentation of the file format.

Do not let it go unnoticed, though, that the XML that works for Microsoft's ShimDBC needs attributes that differ from the programmatic names that Microsoft has documented for the tags or the friendly names that can be obtained from the `SdbTagToString` function. For instance, the `0x6003` tag (`TAG_MODULE`) is compiled from an attribute named not `MODULE` but `FILE`. The `0x4017` tag (`TAG_FLAGS`) is synthesised from two attributes. Even harder to have guessed is that a `LIBRARY` tag is needed in the XML but does not show at all in the SDB file, *i.e.*, as a tag `0x7002` (`TAG_LIBRARY`). So, to know what XML is acceptable to Microsoft's compiler for creating an SDB file, you'll have to reverse-engineer the compiler or do a lot of inspired guesswork.

Happy hunting!

File Offset	Original	Patched	Remarks
0x0002FB54	8B FF	EB 08	jump to instruction that will use existing fixup site
0x0002FB56	55		
0x0002FB57	8B EC		
0x0002FB59	81 EC 88 05 00 00		
0x0002FB5E		FF 15 D0 30 49 00	incorporate existing fixup site at file offset 0x0002FB60
0x0002FB5F	A1 00 60 48 00		
0x0002FB64	33 C5	8B C8	
0x0002FB66	89 45 FC	E8 55 87 01 00	no fixup required for this direct call within `.text` section
0x0002FB69	8B 45 08		
0x0002FB6B		C2 10 00	
0x0002FB6C	53		
0x0002FB6D	56		

Table 2. Patch details for the 32-bit `CompatAdmin.exe` from the ACT version 6.1 for Windows 8.1.



10 Post Scriptum: A Schizophrenic Ghost

by Evan Sultanik and Philippe Teuwen

A while back, we asked ourselves,

> What if PoC‖GTFO had completely different content depending on whether the file was rendered by a PDF viewer versus being sent to a printer?

A PostScript/PDF polyglot seemed inevitable. We had already done MBR, ISO, TrueCrypt, HTML, Ruby, ... Surely PostScript would be simple, right? As it turns out, it's actually quite tricky.

```
$ gv pocorgtfo13.pdf
```

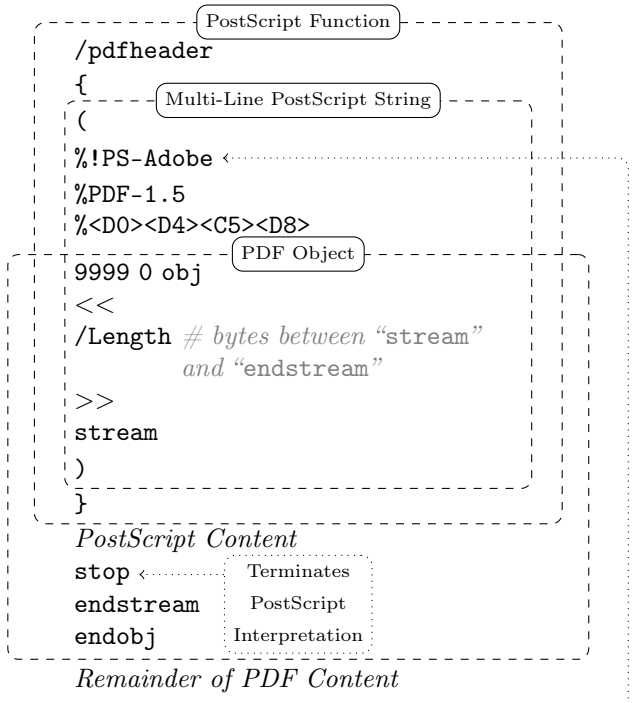
There were two new challenges in getting this polyglot to work:

1. The PDF format is a *subset* of the PostScript language, meaning that we needed to devise a way to get a PDF interpreter to ignore the PostScript code, and *vice versa*; and

2. It's almost impossible to find a PostScript interpreter that doesn't *also* support PDF. Ghostscript is nearly ubiquitous in its use as a backend library for desktop PostScript viewers (*e.g.*, Ghostview), and it has PDF support, too. Furthermore, it doesn't have any configuration parameters to force it to use a specific format, so we needed a way to *force* Ghostscript to always interpret the polyglot as if it were PostScript.

To overcome the first challenge, we used a similar technique to the Ruby polyglot from `pocorgtfo11.pdf`, in which the PDF header is embedded into a multi-line string (delimited by parenthesis in PostScript), so that it doesn't get interpreted as PostScript commands. We halt the PostScript interpreter at the end of the PostScript content by using the handy `stop` command following the standard `%%EOF` "Document Structuring Conventions" (DSC) directive.

This works, in that it produces a file that is *both* a completely valid PDF *as well as* a completely valid PostScript program. The trouble is that Adobe seems to have blacklisted any PDF that starts with an opening parenthesis. We resolved this by wrapping the multi-line string containing the PDF header into a PostScript function we called `/pdfheader`:



The trick of starting the file with a PostScript function worked, and the PDF could be viewed in Adobe. That still leaves the second challenge, though: We needed a way to trick Ghostscript into being "schizophrenic" (*cf.* PoC‖GTFO 7:6), *vi&.*, to insert a parser-specific inconsistency into the polyglot that would force Ghostscript into thinking it is PostScript.

Ghostscript's logic for auto-detecting file types seems to be in the `dsc_scan_type` function inside `/psi/dscparse.c`. It is quite complex, since this single function must differentiate between seven different filetypes, including DSC/PostScript and PDF. It classifies a file as a PDF if it contains a line starting with "`%PDF-`", and PostScript if it contains a line starting with "`%!PS-Adobe`". Therefore, if we put `%!PS-Adobe` anywhere before `%PDF-1.5`, then Ghostscript should be tricked into thinking it is PostScript! The only caveat is that Adobe blacklists any PDF that starts with "`%!PS-Adobe`", so it can't be at the beginning of the file (which is typically where it occurs in DSC files). But that's okay, because Ghostscript only needs it to occur *before* the `%PDF-1.5`, regardless of where.

This article continues in the PostScript!

11 Tithe us your Alms of 0day!

from the desk of Pastor Manul Laphroaig,
International Church of the Weird Machines

Dearest neighbor,

Do you remember what it was like when you first learned to program a computer? Not when you first realized that you could do it well, but when you first realized that you could do it at all? How did it feel?

And do you remember what it was like when you first learned how to use calculus? Not when you first learned how complicated differential equations could become, but when you first realized that with a handful of rules, you could bounce back and forth between position, velocity, acceleration, and jerk as if they were all the same thing? How did that feel?

And do you remember what it was like when you first learned how to use a screwdriver? Not when you first learned what to do after removing the screw, but when you first realized that with a screwdriver—with the *right* screwdriver—you could take apart anything? How did that feel?

When I was sixteen, I was a bit of an asshole, and I asked my automechanics teacher a question about a distributor's angular momentum. I don't recall my exact question, but I do recall that it was the sort of thing no one could be expected to know, and that, being a jerk, I asked it in the vocabulary of calculus.

Coach Crigger could've called me out for being rude, or he could've dodged the question. He could've done any number of things that you might expect. Instead, he walked out of the classroom while two and half dozen hooligans started a racket audible from the other side of the campus.

Ten minutes later, he returned to the classroom. He walked right up to my desk and slammed a '72 Ford's distributor onto my desk along with the screwdriver to open it. It felt good!

Do this: write an email telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

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.

Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Powerpoint deck. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

Don't tell us that it's possible; rather, teach us how to do it ourselves 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.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, D.D.