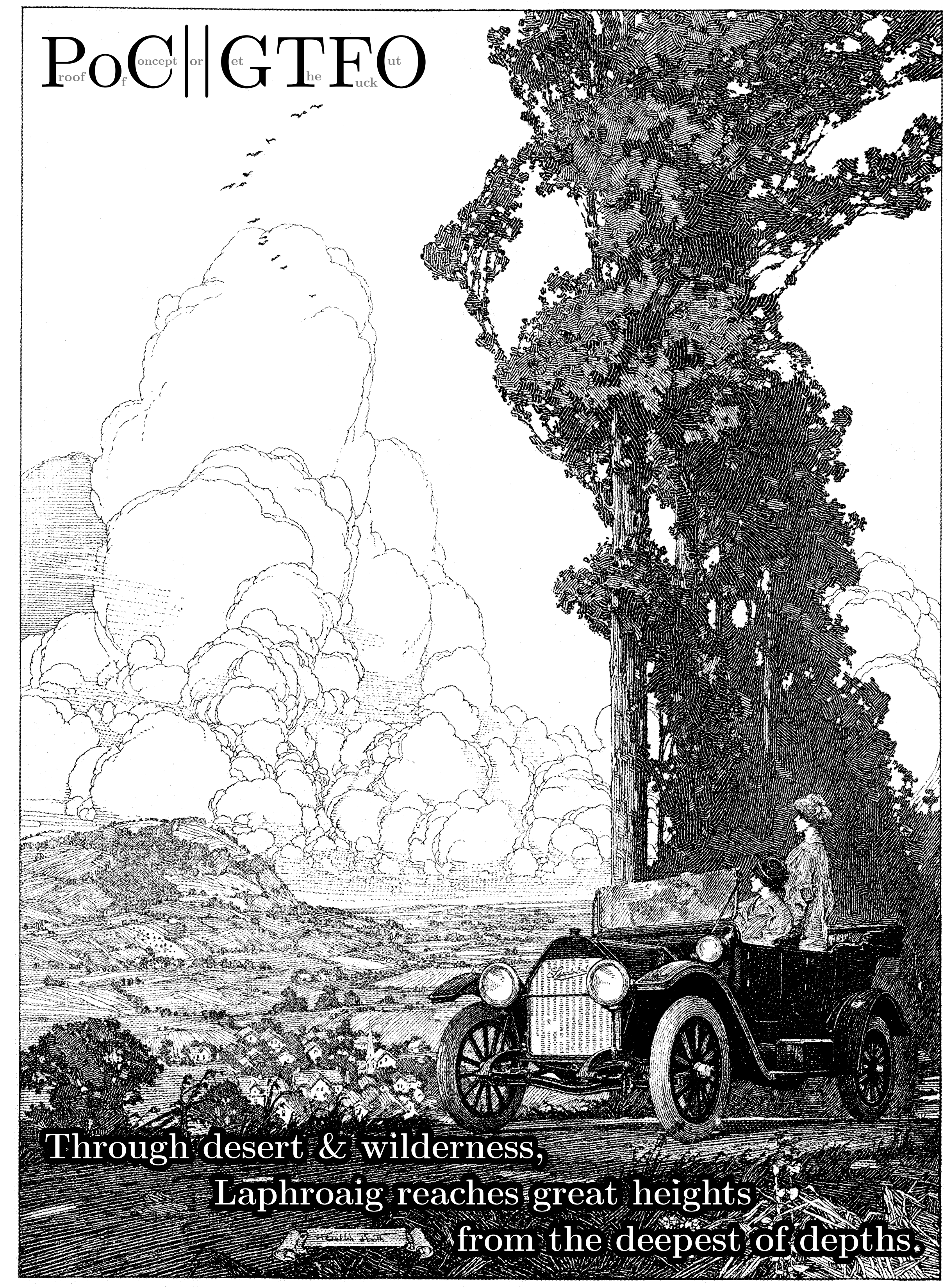


22:02 (p. 5) A Mask ROM Tool
22:03 (p. 11) Mitra and Mocky
22:04 (p. 17) More Letters from Screwtape
22:05 (p. 19) Inside out
22:06 (p. 23) Abusing XFG
22:07 (p. 30) Timecryption
22:08 (p. 32) An Электроника and a Casio
22:09 (p. 39) Renesas M16C and R8C
22:10 (p. 46) A Tourist's Guide to Эльбрус
22:11 (p. 52) Janus Polyglot

Eleven thousand persons have suffered death rather than submit to break eggs at the smaller end.
Compiled for a dozen reasons many dozens of times, the last of which was on February 3, 2024.
€ 0, $0 USD, 0 SEK, $50 CAD, 6×10^{29} Pengő, 100 JPC. Mieux vaut vivre avec des remords qu'avec des regrets.

Legal Note: Did you personally sign a copy of the Official Secrets Act of 1939 before receiving this document? If not, then there's probably no law against sharing it.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo22.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/         https://www.sultanik.com/pocorgtfo/
git clone https://github.com/angea/pocorgtfo
```

Technical Note: The electronic edition of this magazine is valid as both PDF and ZIP. Thanks to Ange Albertini, it is also a polymock with many bogus file type signatures — check page 11.

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 with pages 1, 2, 67 and 68 should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt install texlive-extra-utils
pdfbook2 --short-edge --paper=a3paper --no-crop pocorgtfo22.pdf
```

Man of The Book	Manul Laphroaig
Scooby Bus Driver	Ryan Speers
Editor of Last Resort	Melilot
TeXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funkmaster of File Formats	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Stunts (Uncredited)	Alexei Bulazel

with the good assistance of

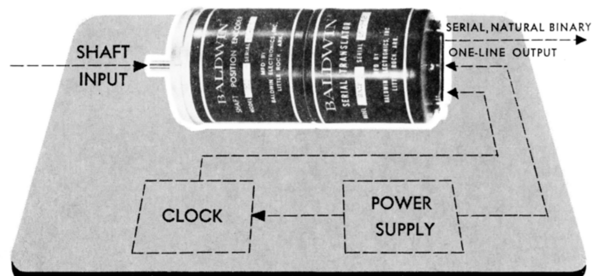
Tree Killer	EVM

22:01 Need something good to read, my good neighbor?

Neighbors, please join me in reading this twenty-third 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 the study of weird machines. This release is a gift to our fine friends in Washington, D.C.

BALDWIN®

ANNOUNCES ANOTHER NEW ADDITION TO ITS GROWING FAMILY OF PHOTO-ELECTRIC SHAFT POSITION ENCODERS



NATURAL BINARY, SERIAL OUTPUTS

The Serial Translator is a plug-in unit for the standard 200-Series Encoder accepting an 11 or 12-bit parallel word at the input and converting it to a serial natural binary word which can be transmitted over one line.

The binary bits are 10 microseconds wide and 10 microseconds apart. The amplitude of the pulses is 4.5V open circuit, 2.3V into a 50 ohm load. Outputs can be in either polarity. The serial output is especially desirable for long cable runs, telemetering links, etc.

Interrogation for the serial translator can be furnished by a free-running or externally driven clock. The word rate of the free-running clock is set at a nominal 1 K.C. (1000 word/sec.). The maximum word rate for the externally driven clock is 4 K.C. (4000 word/sec.).

The encoder and translator combination is 2.6″ dia. and 5½″ long. Clock housing is 2-5/6″ x 5¼″ x 4¼″.

Model 222 Encoder (11-bit) $450.00
Serial translator $248.00
Clock $269.00
(lots of 1 through 10)

P. S. — Our family of standard encoders now numbers over 80, and we are proud of every one of them.

BALDWIN ELECTRONICS, INC.

1101 McALMONT STREET • P. O. BOX 627 • CODE 501-375-7351
LITTLE ROCK, ARKANSAS 72203

If you are missing the first twenty two releases, we suggest asking 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, the thirteenth in Montréal, the fourteenth in São Paulo, San Diego, or Budapest, the fifteenth in Canberra, Heidelberg, or Miami, the sixteenth release in Montréal, New York, or Las Vegas, the seventeenth release in São Paulo or Budapest, the eighteenth release in Leipzig or Washington, D.C., the nineteenth in Montréal, the twentieth in Heidelberg, Knoxville, Canberra, Baltimore, or Raleigh, the twenty-first in Leipzig or Washington, D.C., or the twenty-second in D.C. Three collected volumes are available from No Starch Press, wherever fine books are sold.

On page 5, Travis Goodspeed shares his tools for reverse engineering a photograph of a mask ROM into a ASCII art bitstream, and then converting that physically ordered bitstream into logically ordered bytes that might work in a disassembler or emulator. If you need to reverse engineer microcontroller firmware from before flash memory became cheap and plentiful, this is the tool for you.

Ange Albertini wrote PoC||GTFO 7:6, the classic article on abusing file formats with polyglots. On page 11, he presents a follow-up with better classifications and the idea of "polymocks," which are not polyglots but easily confuse `libmagic` and its friends into believing that file is valid in dozens of formats.

Eighty years ago, C.S. Lewis published the Screwtape Letters, a classic of apologetics presented as letters from a senior demon named Screwtape to his junior nephew, Wormwood. On page 17, Pastor Laphroaig shares with us a more recent set of mis-delivered letters, in which Wormwood—now a senior demon—writes to his young nephew Malört about modern video clips, computer programming and how hard it is for a concerned demon to earn the wages of sin.

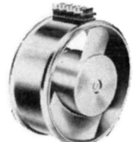
On page 19, Ange presents a series of tricks building up to generic, reusable hash collisions for tarballs and zipped XML files, such as `.docx` files.

We make this airmover unit,

Vaneaxial gas bearing fan, ultra-high reliability, used in *Minuteman.* Only IMC produces it.

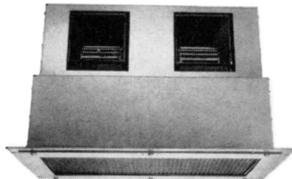
and this unit,

The Boxer®, standard sized, distributor-stocked. Rugged metal frame, our own impeller and motor, moves air economically and reliably.



and this unit,

Tubeaxial, the IMCool is distributor stocked in 10 types, inputs of 60, 400, and 1000 cps, air delivery from 18 to 108 cfm.



and this unit,

Rackmounted double blower. This one installed right in the broadcast area, delivers 500 quiet cfm to a high-power broadcast transmitter.

but this unit will design and build an airmover to fit your specifications.

IMC is the single-source specialist for airmovers. We design and build the entire system—motor, blade, bearing, housing—and can assume full responsibility in meeting your needs. For quick response, contact Sales Dept., Eastern Division, Telephone (516) 334-7070, or TWX 516 333-3319. Plants: Westbury, N. Y.; Maywood, Calif.; Rochester, N.H.; and Tempe, Ariz. Products: Airmovers; induction, hysteresis, synchronous, torque, servo, and stepper motors; synchros; solenoids; pressure switches; hydraulic and pneumatic valves; and fuel atomizers. For more data, write:

IMC

IMC Magnetics Corp. Marketing Division, 570 Main Street, Westbury, N.Y. 11591.

Windows, LLVM and Grsecurity all have control flow integrity schemes that can restrict the targets of indirect calls, such as function pointers. Aleksandar Nikolic has been playing with the eXtended Flow Guard scheme from Windows 11, using the hashed integrity markers as a means of reverse engineering the calling conventions of functions. What began as a mitigation against memory corruption exploits has become an oracle for reverse engineering!

Stefan Kölbl and Ange Albertini have been playing around with CTR mode, coming up with near-polyglots that have a different meaning and file format for each of a few different key/nonce pairs. Page 30.

A long time ago in an evil empire far away, the Soviet Union's consumer electronics monopoly produced a pocket calculator, the Электроника MK-51. This looks exactly like Casio's fx-2500, and on page 32, Travis Goodspeed deconstructs both calculators to show that the MK-51 counterfeits not just the look and feel of the Casio, but also its NEC microcontroller and every last bit of mask ROM.

We've recently been including tourist guides to new computer architectures, and this release is no exception. Christopher Hewitt and Niccolò Izzo describe the M16C and R8C series of microcontrollers from Renesas on page 39, beginning with the basics and working their way up to a fault injection attack. EVM can't let them have all the fun, so page 46 presents his guide to the Elbrus 2000 architecture, Russia's domestically designed VLIW architecture with register windowing.

Harvey Phillips shares on page 52 his Janus polyglot from the Binary Golf Grand Prix. It's valid as an x86 bootloader, ELF, COM, RAR, and a GNU Multiboot2 image, but also as program for the Commodore 64! To keep the size to a minimum, many of these formats have useful sections overlapping.

On page 68, we pass the collection plate, not for bitcoins or wooden nickels, but for nifty stories. What fine stories do you have, left untold except at your local pub? With what clever tricks might you grace our readers?

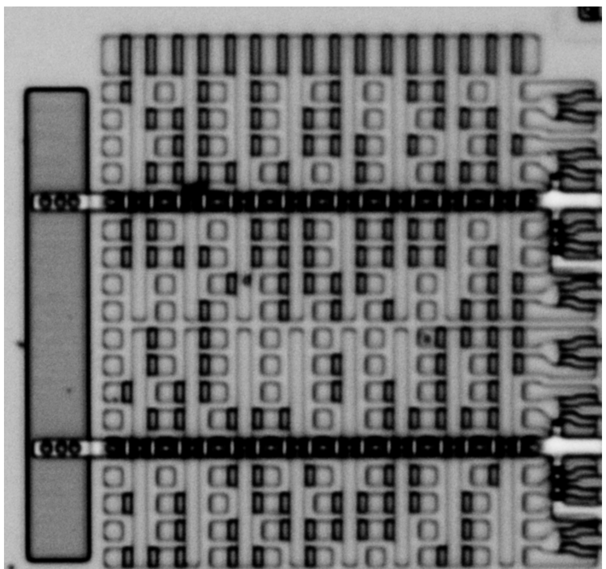
22:02 A Mask ROM Bit Extraction Tool

by Travis Goodspeed

Lately I've been writing a book on extracting firmware from locked microcontrollers; rather, trying to write that book, because I fell into a rabbit hole of mask ROM reverse engineering. So for a few months, instead of writing prose, I wrote a tool in C++ for extracting bits from ROM photographs and a matching tool to decode those bits into logically ordered bytes, suitable for disassembly or emulation.[1]

Let's begin with a little background: SRAM, DRAM, Flash ROM, and EPROM memory technologies hold bits invisibly as some form of electrical charge. Mask ROM is different, in that bits are written into one of the lithography masks that produce the chip. This is very expensive per unique program, but very cheap per chip.

Many chips include a nice, orderly grid of bits that contain code or data. Sometimes this is encoded in metal vias, which you can see from the surface of a decapsulated chip. Sometimes bits are in the diffusion layer, and you need to remove the upper layers of the chip with hydrofluoric acid to expose them. And sometimes bits are implanted into the doping difference between P and N silicon, requiring a procedure called a "Dash Etch" to stain a color difference into the bits after exposing them.



Whatever the chemical procedure, the end result from the lab is a panorama photograph of the ROM under high magnification, with bits visible to the naked eye. Like the ones on page 6, you will see that some bits are bright while others are dark. These are our ones and zeroes!

If you are more patient than I, you might type these in manually, reading the ones and zeroes off the page in the same way that as children we typed BASIC program listings out of magazines. Instead, it's nice to let a computer do that work, with a human providing a minimum amount of guidance.

Prior Work

The first of these tools to be published was Rompar by Adam Laurie in 2013.[2] It's a GUI application in Python, in which the operator draws a grid to mark the bit positions. OpenCV takes care of a bit of image preprocessing, to make the bits stand out by tossing away unneeded color channels.

Published later in 2019, but perhaps written earlier, is Chris Gerlinsky's Bitract.[3] The user first loads an image and then describes an "area of interest," a box containing so many rows and columns of bit positions.

These tools certainly work, but they have some problems that frustrated me enough to write something new.

Bitract requires a commercial image processing library to compile in Borland C++. As a Windows program, it ignores command line parameters and has no CLI. As a Python script, Rompar makes too much use of command-line parameters, requiring the row and column count of each bit grouping to be defined before startup rather than worked out on the fly.

Both Rompar and Bitract expect bits to be perfectly ordered in a grid, which is great for reducing the operator's labor, but difficult on very large projects, where camera or stitching distortions might move something just barely out of the grid. It's also inconvenient on some 4-bit microcontrollers, where the final group of bits sometimes has fewer columns than the others.

[1] `git clone https://github.com/travisgoodspeed/maskromtool`
[2] `git clone https://github.com/AdamLaurie/rompar`
[3] `git clone https://github.com/SiliconAnalysis/bitract`

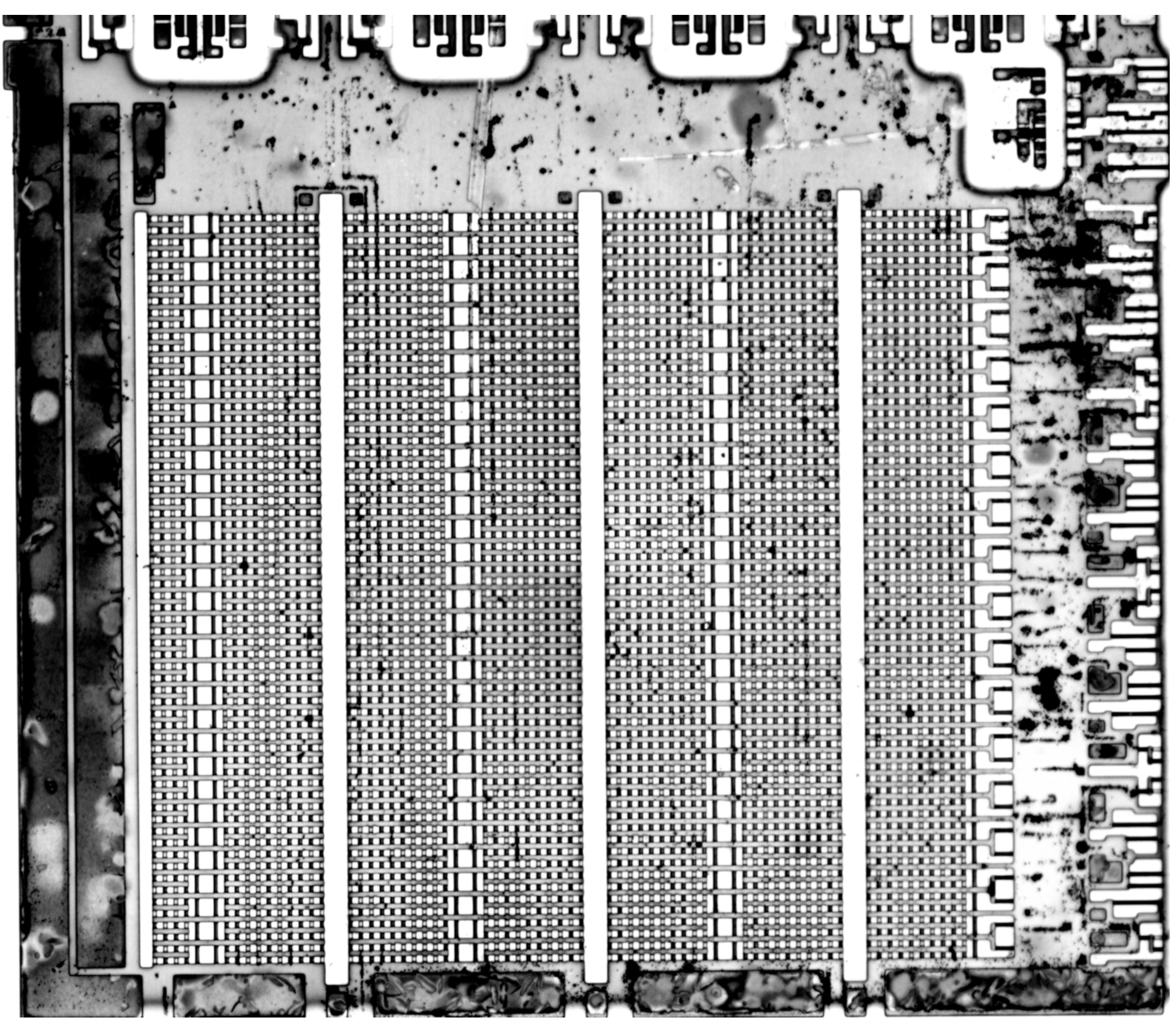


Figure 1: Font ROM from a TMP47C434N

Bitract uses the mouse wheel to zoom, which is infuriating on a multitouch pad that ought to be able to both zoom and pan. Rompar, by contrast, traps the user at the native resolution of the image.

I write these things not to criticize their work. These were damned handy tools for their day. I just think that things could be more convenient.

A Fresh Start

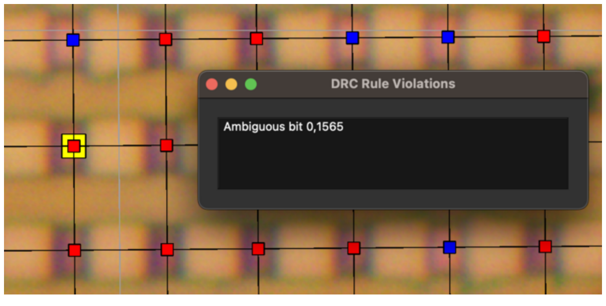
So I began from scratch, with a design on a paper.

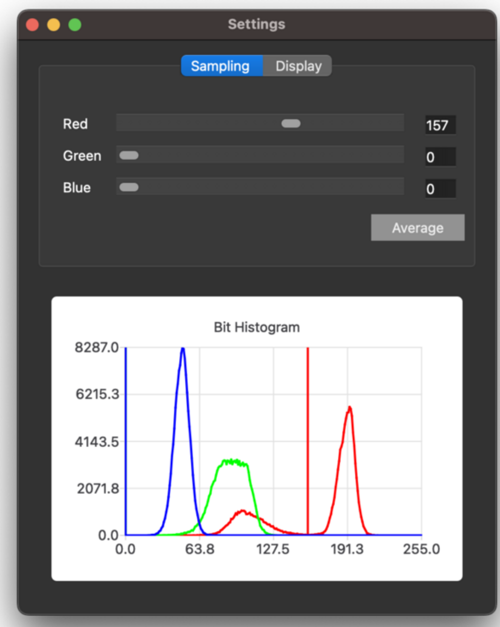
For starters, I decided to support both a GUI and a CLI. I wrote the GUI in Qt6, for portability to Linux, Windows and macOS. The CLI is handy for regression testing and scripting, but it is strictly optional. All important features are available without it.

Like Rompar, I chose JSON as a save format. Like Bitract, my tool shows handy histograms to quickly choose the best bit threshold.

I decided to avoid having a strict grid of bits, but instead to use other objects to generate bit positions. In my early drafts, this was done by drawing row and column lines, then identifying their crossing points as bit locations. Because only the bit locations really matter, there's plenty of time to add support for marking grids later.

Since this is a CAD program at heart, I took a few lessons from the PCB layout tools that I regularly cuss at. Design Rule Checks (DRCs) were written early, implementing such features as identifying overlapping bits, ensuring that each row is the same length, and sanity checking the design in other ways. Each DRC violation has a position in the project view, appearing as a yellow box beneath the bits but above the photograph. DRC violations can also be used for providing other feedback; the list isn't necessarily restricted to errors.





Determining a Bit's Value

Knowing the position of a bit, how do we determine whether it's a one or a zero? The short answer is that we can look at how bright or dark it is, but there are some complications to consider.

The first is the color space. Often the bits are distinct in one color channel but absolutely indistinguishable in another. And once we know the right channel, we must select the right threshold to distinguish them.

I found that by drawing a histogram of the number of bits of a given color value, I could quickly see a bimodal distribution in some channel between ones and zeroes. Sliding a channel threshold automatically updates all visible bits, as well as updating a marker in the histogram to visualize where your threshold is set.

I don't use OpenCV or similar libraries to preprocess my images. Rather, I've found that most implant and contact ROMs consistently have a color difference that can be found on a single pixel when working with losslessly compressed images.

Diffusion ROMs are a little different, in that they are low in the chip, and when the chip has been processed a little too long, there's no color difference in the bit's center. Rather, the bit has a dark border. For these and other edge cases, my tool abstracts

away the measurement as a class that returns an RGB triplet. To support this edge case, I simply wrote classes that measured a thin horizontal or vertical strip of pixels, returning the darkest point in each color channel along that strip.

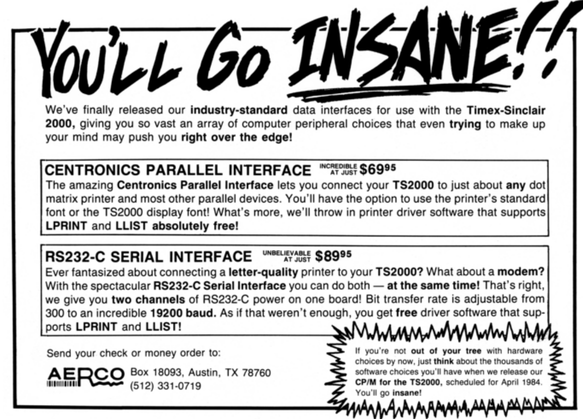
Aligning Bits Into Rows

After the user draws lines for all the rows and columns of the ROM, we take those intersecting points and produce a set of bits positions. Because we don't have a definite grid, it's necessary to *align* these bit objects into rows.

Initially I solved this problem very inefficiently, implementing a function to find the next-to-the-right of any bit by restricting its angle and marking all bits I'd already passed. This worked great for small ROMs, but it scaled horribly, and by one hundred kilobits it was taking twenty minutes to align the bits!

To come up with a faster algorithm, I realized that sorting all of the bits by their X coordinate would *almost* group them into columns. The exceptions come from the image's tilt. Sometimes the leftmost bits of the second column are to the left of the rightmost bits of the first column.

We can therefore identify the leftmost bits by following the sorted list. Whenever the Y gap is small, say less than a few times the average gap, we're still on the first column and we've identified another row header. If the Y gap is large, we're seeing a bit from the second column, and we ought to pass it by. When we start to see many large gaps, we've passed the first column entirely and know all the row header bits.



So to align the bits, I first build an array of the rightmost bits of each column that I've yet passed. This array is seeded with the row header bits at the far left. I then walk through the sorted list of all remaining bits, overwriting their nearest row element in the array after updating the old bit's `nexttoright` pointer to aim at the new bit.

This is lightning fast, reliably arranging hundreds of thousands of bits in the blink of an eye.

From Physical Bits to Logical Bytes

By this point in the article, you should understand how you might use MaskRomTool to mark the bits of a photograph and arrange them into a table of bit values. You also understand how the DRC mechanism might flag bits which are too near the threshold between a one and a zero. But there's a very important piece we haven't yet covered: How does the software convert this table of physically-ordered bits into logically-ordered bytes?

Let's begin with the prior art. John McMaster's Zorrom tool is built as a set of Python scripts with libraries for CH340, LC5800, LR35902, MCS48, PIC1670, and some TMS320 chips.[4] For those chips that it doesn't directly support, it has a solver feature that will attempt many permutations of decoding until the bytes match a defined pattern, such as setting the stack pointer in the first instruction. John's solver works for roughly half of targets, and it's far easier than manually guessing permutations. This was the tool that I used until I recently wrote my own decoder.

Chris Gerlinsky's BitViewer uses the a totally different strategy.[5] Rather than automatically searching permutations, it instead graphically displays the bits with adjustable grouping into columns. This helps a human operator explore the layout, while overdosing on caffeine in a hyperfocused fugue until eventually the bits make sense. This understanding doesn't come easily, but I and others have done it.

I wanted the best of both these worlds. From Zorrom, I wanted a CLI tool that could quickly process my projects, driven by a Makefile to rerun them in order to catch regressions in my decoder or images. I also desperately needed a good search feature, and Zorrom was the only example of such a thing when I started. And from BitViewer, I wanted

[4] `git clone https://github.com/JohnDMcMaster/zorrom`
[5] `git clone https://github.com/SiliconAnalysis/bitviewer`

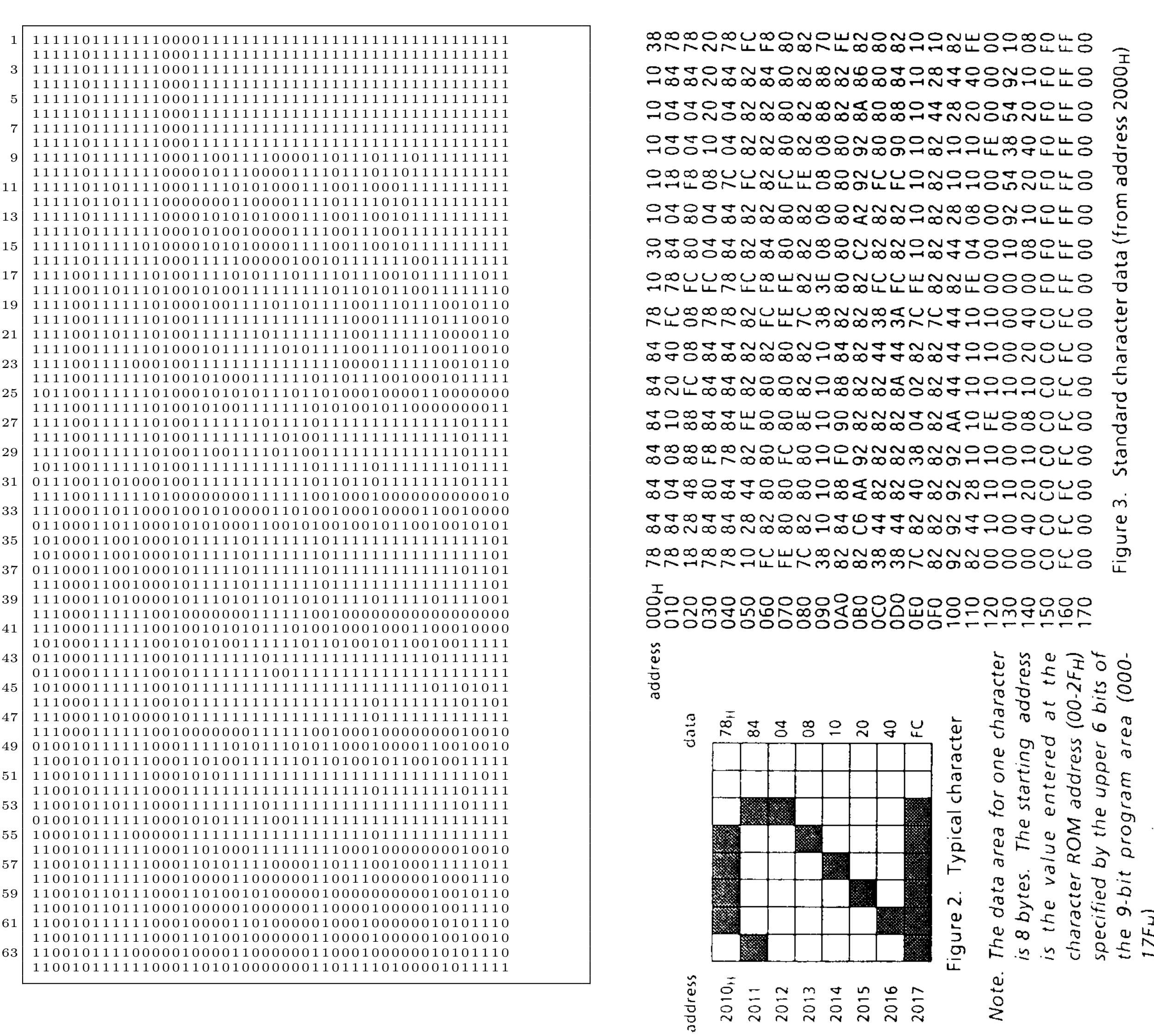


Figure 2: Extracted Bitstream and Datasheet Bytes of the TMP47 Font ROM

some graphical connection to my project file, so that a nearly correct guess could be explored until the error was found.

My decoder is called GatoROM. It is used either as a CLI tool without the GUI overhead of MaskRomTool, or as a C++ library within the GUI.

Since Zorrom was the gold standard of solving for unknown layouts, I began my decoder with the complete set of Zorrom permutations from an arbitrary bitstream, essentially exporting every case that it would investigate. Once I could match decodings of all these, and also permute between all settings, I knew that my solver had feature parity with McMaster's.

GatoROM uses its own class to represent a bit, different from that in MaskRomTool. The class holds values such as its address and bitmask during the most recent decoding, as well as a void pointer that might point to a matching bit in the GUI.

All of the GatoROM bits begin in a table of their input positions, and transformations (flip, rotate, etc) produce a table of bits in the output order. This output table is then passed to the parser for decoding, when the address and bitmask fields of the bit class instances are updated to record their logical positions. Zorrom does these steps in roughly the same order, but by passing values instead of pointers, it does not preserve relationships between the inputs and outputs.

I wrote earlier that I also wanted something like BitViewer's interactive nature in my tool. By recording the address and mask of every bit that is decoded, my tools can easily show their work. It's no trouble to select the first few bytes of a decoding, then ask the tool to highlight the bits of those bytes in physical order.

Specifications

± .25 dB 15 Hz - 40 KHz
± 1 dB 5 Hz - 100 KHz

< .01% T.H.D.

Equivalent input noise < -127 dBM

+16 dBM into 600 Ω
25 Mw into 40 Ω headphones

8—12 hours continuous operation on internal 9 volt batteries depending on phantom load.

$620

3.7"

An Excellent Mic Pre-Amp,
the AERCO MP-2 is designed with what we believe to be the very best components available.

Jensen Input Transformers
have no known rival. Exotic materials and fanatical devotion combine for 3dB points of 2 Hz and 250 KHz.

Extremely Low Noise
We selected the Linear Technology LT–1028 amplifier on the basis of sonic clarity and super low noise performance that will instantly bring a grin to your face. The facts that it drives +24 dBM into 600 Ω, that it maintains less than 100 uV of D.C. offset at the output, and that it consumes just 200 mW of power are more icing on the cake.

High Efficiency Power
converter operating at 500 KHz supplies all the internal power requirements. It has been designed for minimum noise generation and maintains a conversion efficiency well in excess of 90%. Input power can range from 7 - 20 Volts D.C. without regard to polarity and with no electrical connection to the audio circuit.

High Gain... Low Gain
eight independently selected gain settings for each channel from 20 to 50 dB. Gains are set by a switch and network of precision resistors instead of a pot. This ensures the lowest possible noise and eliminates the reliability problems inherent in small pots. The switches are accessed through the RCA jacks.

Phantom Power
is implemented with the industry-standard 52 volt circuit for proper operation of 12 volt and 48 volt microphone types. Individual jumpers for each channel disable the phantom voltage for use in unbalanced environments.

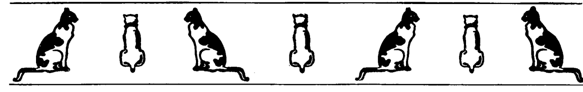
Great Quality - Small Price
We've priced the 2 channel unit at $560 to make you a fast friend. Then we plan to sell you more neat stuff.

AERCO
Box 18093 Austin, TX 78760
(512) 451-5874

Neighborly greetings to John McMaster for his ground breaking work on Zorrom, for helping me get my lab back together, and for patiently explaining all those things of semiconductor reverse engineering that I had fallen behind on in the past years. And cheers to Vicki Pfau for being smart enough to decode the arrangement of the TMP47 font ROM, which will soon lead to a general decoder for TMP47 program ROMs.

My tools don't yet solve every ROM that was ever manufactured, but I'm happy to say that they are now the best tools for any particular ROM extraction job. They are fast, they hardly ever crash, and they run reliably from the command line or from an OpenGL GUI, whichever you might prefer.

Now that I've solved that problem, perhaps I can get back to finishing my book?



TS 2068 DISC SYSTEM

FD-68 INTERFACE
Controls 1-4 drives

3 inch to 8 inch drives
Shugart compatible

Single or double sided

40/80 tracks per side

64K RAM & 8K ROM
on board

RGB monitor output

SYSTEM COMPONENTS
$199 FD-68 Interface

$99 Drive 40T/DS/DD
5 inch/400 kilobyte

$99 Dual Drive Cabinet
and 5 amp Pwr Pack

$3 Per Item S&H

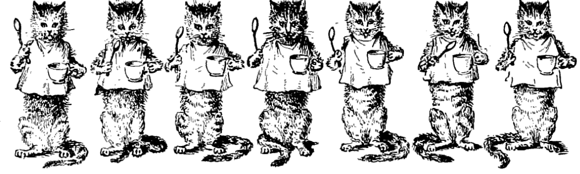
Texas Residents add 5%

VISA/MasterCard add 5%

Enhance the performance of your TS 2068 with the **AERCO** Disc System. All of the speed and convienience of a full-out floppy disc system. Load programs at an incredibly fast 250,000 bits/sec. Fully compatible with all Shugart type drives, including those already in use with the **AERCO** 1000 Disc System. The 64K of on-board RAM can be used as a second bank of system memory or a soon to be released full-blown CP/M System (version 2.2). The RGB output is crystal clear and rock steady. The power supply is a 5-amp high efficiency switcher. We offer a variety of other hardware for all models of SINCLAIR-TIMEX.

	TS/2068	TS/1000-1500
Floppy Disc Interface	$199	$179
Disc Drives	from 99	from 99
Power Supplies	99	99
Centronics Printer I/O	69	99
Dual RS-232C Serial I/O	99	99
Direct Video Mod (DV-1)	n/a	15
C ITOH 8510 Printer	375	375
C ITOH 7500 Printer	275	275
ROM Bd. with Auto Disc Boot	n/a	59
RGB Cable (specify monitor)	30	n/a
CP/M (V. 2.2)	coming soon	n/a

Box 18093 Austin TX 78760
Ph (512) 451-5874



22:03 Mitra and Mocky: Near-polyglots and Mocks

by Ange Albertini

Our readers are encouraged to read Abusing File Formats (PoC||GTFO 7:6). This is a follow-up with better classification, leading to the creation of tools to automate the generation of various kinds of weird files: mocks, polyglots and near-polyglots. We also share a tool, Mitra, to help apply these techniques.

The basic idea to abuse files is to make space for foreign data that will be ignored by parsers. This can rely on various features of the targeted file format.

Signature Some formats like MP4 and PostScript are parsed from offset zero, but they don't enforce a magic at that offset. As a consequence, it's possible to make a MP4 / PostScript polyglot by abusing the length of an MP4 *atom* to encode a PostScript declaration for a dummy function that will cover more MP4 structure.

A polyglot MP4/PS header starting a dummy function but also an MP4 *free* block.

```
    /PostScript/ whitespace then line comment
00:00 00 00  %  f  r  e  e \r \n  %  !  P  S \r \n
   \----MP4---\ Declares a comment of length 0x25

   /PostSc./ Declares a function name
              then a line comment
10: / {   (  % 00 00 00 00 00 00 00 00 00 00 00 00
20:00 00 00 00 00 00 00 XX  f  r  e  e \r \n  )  }
              \---MP4----\ Declares a comment len
```

The same bytes look quite different when interpreted as a PostScript file, where `00 00 00 XX` declares the length of a free chunk covering the whole PostScript.

```
\0\0\0%free
%!PS
/{(\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\xXXfree
)}
```

Starting parsing at offset zero is only a half-measure against file-format abuse. Only when signatures are enforced at the very beginning will file format abuse and confusion become impossible.

On the other hand, having a signature for each frame such as Ogg or ILDA can be superfluous if the frames are properly length-defined.

[6] See "The .xz File Format, v1.0.4."

Cavities Some formats start with *cavities*, which are totally ignored. We used to call these empty spaces.

A typical example is the ISO format, being a raw dump starting with empty sectors. This is commonly used in IsoHybrid to make a single ISO that boots either as a thumbdrive or as a CD-ROM.

Another example is the DICOM format, which by courtesy skips the first 128 bytes of the file. A magic signature and enforced structure are required after that range. This is particularly convenient for TIFF files, since unlike most picture formats, the eight-byte long Image File Header points to the File Directory, which points to the image data, so it's easy to move the big chunks of structure and data around the DICOM ones. In short, a TIFF/DICOM polyglot is easy, and a TIFF/DICOM chimera—sharing the same image data—is even possible.

Appended data Most parsers will ignore any further data once the file is considered "complete enough," which is determined by whether some kind of recovery is required or not.

It might be explicit via some kind of terminating marker. Or it might not, when enough data is present or by intentionally triggering a parsing bug to end the parsing. An example of triggering a recursion exhaustion warning can be found in PoC||GTFO 17's AGC polyglot.

Data added after the end—and ignored—is called appended data, but it's rarely mentioned explicitly in the specifications. It's simply tolerated.

Some formats actively prevent appended data by having a footer. A footer can be thought of as a specific structure required to be at then the very end of the file such as ID3v1, which is a footer appended itself to a MP3/Layer3 stream, or the XZ archive format, which officially enforces a footer to define a file as complete.[6]

Other formats like Dicom, Ogg, ILDA or PCAP[NG] enforce the whole file to follow a given structure: they're just pure sequences, sometimes even requiring a magic signature for each structure such as ILDA or Ogg. They just go on parsing everything from the file, and returns an error since it's invalid.

Metadata Since it's present in the file but not needed for parsing or rendering the file's contents, metadata is a great source of abuse. Many old school formats have fixed length fields, as hard coding was the norm back then.

Comments are typically ignored empty space with typically a set length that is declared before, and are present in most file formats. Unlike PDF, XML enforces an encoding for its comments like the rest of the file, but in general, comments are just ignored, and preserved, no matter their amount, their length or their content.

Comments are not the only source of abuse. Extensible metadata with a user-chosen ID, or fields like file names in an archive can also be used to store some foreign data. A notable exception is that in Gzip, the optional comment and file name are null-terminated, which shows that they're intended to store standard text, while the also optional Extra Field is defined with a 2-byte length.

As a side note, metadata may seem like a permanent risk entirely, and it's natural to wonder why we define them officially in every format if they are so easily abused later. While metadata doesn't seem like an initial requirement to keep the format simple — like the Quite OK Image format — it is eventually needed to be able to keep extra information in the file, which is exactly what happened for the MP3 files.

At the release of *l3enc* (the original Mpeg Layer 3 encoder in 1994), the files initially had an `l3` extension, had no file format whatsoever. They were pure sequences of layer 3 frames, each with their own frame header with no signature, making them hard to detect and easy to confuse with other data such as JPEG segments.

Since there was no way to store any metadata in L3 files, the compatible ID3v1 *footer* with a hard coded length was unofficially defined. More structures were defined in other clumsy ways around the L3 stream (Xing, Lame, APE, ...), showing the need for proper definition of metadata storage from the beginning. ID3v2 eventually defined a header, a magic which gave at last a proper format to MP3 streams.

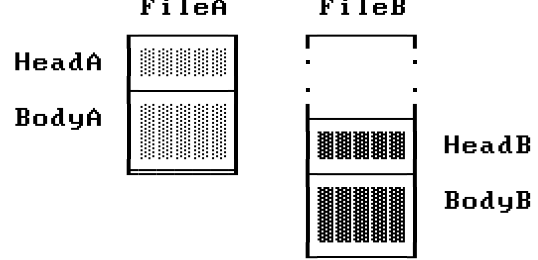
It's a shortsighted move to come up with a great compression algorithm (e.g., MP3, QOI) and define a way to store some data in a file format without the ability to extend it with new but optional data in the future. Even if it means that these structures can be abused, you can't have an extensible format that can't be abused, and people *will* extend or fork your format if not, requiring an extra format that could have been avoided from the beginning.

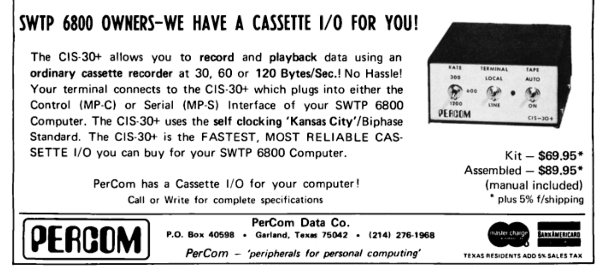
Wrappending Some formats don't tolerate appended data, but they can end with a parasite-hosting structure, which acts like appended data, just wrapped in a declarative structure. Since most data-storing structures have all their declaration before their data itself, it behaves like appended data from the outside, even if the length of the appended data has to be declared somewhere. It looks like appended data — ignored but tolerated but it's technically an appended parasite — declared and skipped.

Zipper Some formats have very strong constraints: a tiny cavity, or very small parasite length (256 bytes for a GIF). Rather than parasitizing a whole file, let's just add the declaration of the file, a declaration of a comment, and then store the rest of the file as appended data.

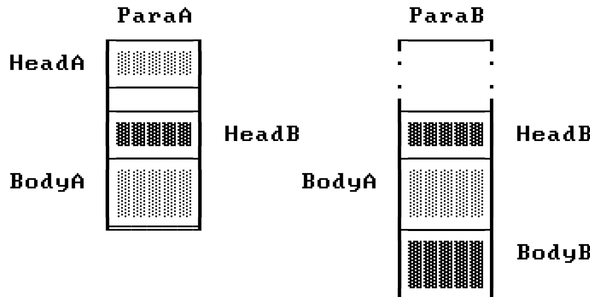
A zipper is a fabric construct of two sliders where each tooth interlocks with the other side's tooth. As an analogy, a *zipper* is a file construct where each format comments the other format's elements, and each tooth is a parasite for the other format.

The simplest form of a zipper is made of two formats: Format A starts at offset zero, tolerates appended or wrappended data, and Format B starts with a cavity. Both formats can be parasitized.

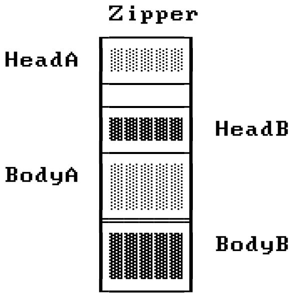




We parasitize File A with Head B, adding padding if required. We also parasitize File B with Body A, *wrappending* Body B in advance if required.



When we merge these files, it looks like this.



Zippers combines various format features (cavity, parasite, appended data) to overcome limitations and make even more weird formats combine.

Mitra is a tool that combines all this knowledge for 40+ different format, generating hundreds of format combinations with different strategies.[7]

Mitra is a simple tool. It doesn't understand file formats structure, it just contains the minimum amount of information to identify and parasitize a file format. It expects standard files as input!

Abuses

Payload embedding The simplest form of exploitation is to just embed a payload that doesn't need to be a valid file. In this case, use the `--force` command line parameter.

The universal example for that is HTML or JavaScript that can be embedded in most file formats. If the file is too big, the HTML page might take too long to load entirely. In that case, use JavaScript to break out of the appended data and limit the parsing to the web payload only.

Mocks While 7:6 covered file type identification, it didn't cover any exploitation. The easiest way to exploit file type identification is just to give a binary blob the right signature at the right offset. It could even happen accidentally. Such a file is a *mock* file. A simple example is `FF D8`, a two byte file that can be identified as a `JPEG image`.

Polymocks It's also interesting to just add a mock signature at a given offset in a valid file via any of the previously mentioned techniques.

`Mocky` is a tool that uses the Mitra library to insert specific filetype signatures at specific offsets to create polymocks.[8] Since programs like `file` have so many signatures that they are scanned by alphabetic order of their category, it's possible to predict which detection will be returned first, and the order might not be what you'd expect from a threat model perspective.

Of course, it's possible to cram as many signatures as possible within the constraint of the target format, such as this issue, a valid PDF with many extra mock signatures stored in a standard stream object. Here is an example of such a *polymock* file, with many filetype detections yet no valid content:

0+2 Dos executable
2+2 Arj
4+2 JPEG 2000
6+2 UnicOS
8+4 Symbian
C+4 Sndh
10+4 Nintendo Switch
14+4 Zoo
18+4 Nintendo Wii
1C+4 Rar v1.4
20+4 AFS
24+4 zImage
28+4 PkZip
2C+4 PolyTracker
30+6 SymbOS
36+6 7-zip
3C+4 SoundFX
40+4 VirtBox
46+2 Int 21h
48+4 PkZip
4C+4 ScreamTracker
50+8 Rar v5
58+4 LrZip
5C+8 Plot84
64+7 Rar v4
6D+5 EZD Map
72+6 Xz
78+4 LZ4
7C+4 LZ4
80+4 DICOM
84+C PDF

[7] `git clone https://github.com/corkami/mitra`

[8] `unzip pocorgtfo22.pdf mocky.py`

Running `file` with `--keepreading` gives an impressive list of detected formats:

```
Plot84 plotting file
SymbOS executable v7.z, name: ...
Old EZD Electron Density Map
Zoo archive data, vj., modify: v78.88+
Symbian installation file
Scream Tracker Sample adlib drum stereo...
Poly Tracker PTM Module Title: "MZ...
SoundFX Module sound file
Nintendo Wii disc image: "NXSB...
DICOM medical imaging data
Linux kernel ARM boot executable...
VirtualBox Disk Image, minor 8653 (MZ...
JPEG 2000 image
ARJ archive data
COM executable for DOS
unicos (cray) executable
data
```

Mocky has a `--combine` flag for to try and insert as many signatures in a file as possible.

`file` has an extra weakness that it has special support for tar files before any other format, and can identify tar files not by their magic, but by the validity of their header checksum, even without any tar signature in the file. This is used in this issue too, so even if the file is a standard PDF starting with a generic PDF signature, `file` with no parameter sees it as a `tar archive`, even if it doesn't contain a magic Tar signature.

Mocky will adjust the tar checksum if used for a polymock file. Here is such a empty `mock.tar` file, detected generically as `tar archive` even if it contains no signature at all:

```
000: 00000000 00000000 00000000 00000000
...
090: 00000000  4 0 0       00    00000000
...
1F0: 00000000 00000000 00000000 00000000
```

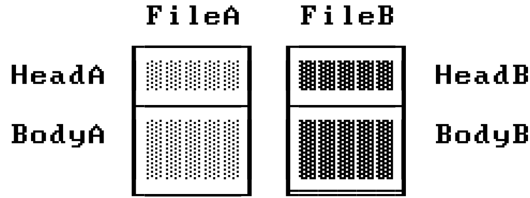
Adding a valid tar checksum to the previous polymock example will indeed return a tar filetype — if the `--keep reading` parameter isn't used — despite all the other present signatures.

Near-polyglots Formats that require a different signature at the same offset can't be combined in a polyglot. However their combination can still be exploited in different conditions. *Near polyglots* are files that are almost polyglots, except that some bytes have to be replaced so that the file type changes.

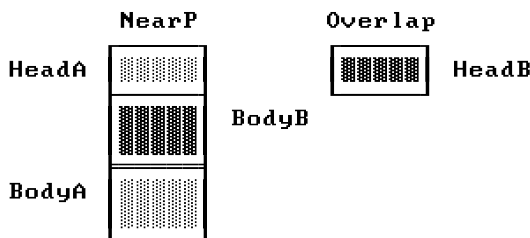
This change could happen over the network if some packets arrive in a different order. It could also happen due to weak bits, leading to different contents. And it can happen via a cryptographic operation in which case you can call them *crypto-polyglots*.

One of these use case is Angecryption, introduced in PoC||GTFO 3:11, where I demonstrated abusing the Initialization Vector of CBC, CFB or OFB block modes to replace the first block of the crypto-polyglot. A new abuse of pseudo-polyglots is presented in the TimeCryption article on page 30 of this release.

To generate a near polyglot, you need very light constraints. Format_A can be parasitized and Format_B can start at the same offset, and needs to tolerate appended or wrappended data. Technically, generating a near polyglot is like parasitizing a format, ignoring that they both start at overlapping offsets, and keeping the head.



Just parasitize File_A with Body_B and keep Head_B as the overlap:



The minimum length overlap is basically the number of bytes where you can declare a file then have some unparsed space, either naturally or by declaring a comment. This value can change drastically between file formats, as shown on page 15.

For example, it's 1 for PostScript because you can declare a line comment with %, so provided there's no encoded newline *after* decryption, this will be a valid ignored space. The PE file format's minimum overlap is two bytes — `M Z` — because you can abuse the DOS header, limiting you to a 58-byte parasite.

A JPEG header with comment declaration is `FF D8 FF FE XX YY`, which is six bytes, with `XXYY` being the length of the comment in big endian. However, if you need a `0x3489`-long comment, a `0x35??`-long comment will do the trick, so you don't have to bruteforce the `YY` byte. If you feel luck, you might

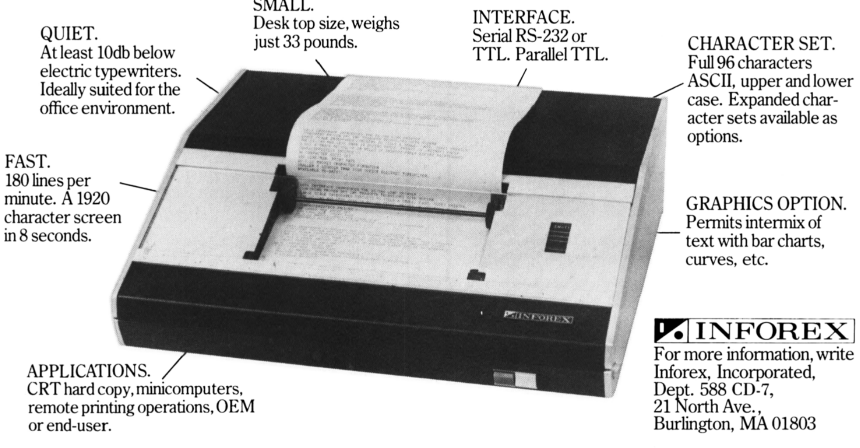
```
                                                               Variable  Unsupported
                                                                offset     parasite
Minimal start offset
         1 2 4 8     9  16  20  23  28  34  40  64  94  132    12  28
                      12          26  32  36      68  112 226    16

         P P J F M T F W G P R I R B C I P C J P E A P I I J    W B O      B E G L N
         S E P l P I L A Z N I D T M P L S A P C L R C C C a    A P G      Z B I N E
             G a 4 F V D   G F 3 F P I D D B 2 A F   A O C v    S G G      2 M F K S
               c   F         F v     O A       P     P     a    M            L
                               2               N
                                               G

1* PS    . M A ? ? ? ? ? ? A ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?    ? ? ?      ? ? ? ? ?
2@ PE    M . A A A A A A A A A A A A A A A A A A ! ! ! ! ! !    M M M      ! ! ! ! !
4+ JPG   A A . A A A A A A A A A A A A A A A A A A A A A A A    A A A      A A A A A
.  .
.  .     [the table could go on but would take too long to bruteforce]

           X: automated     ?: likely possible
           M: manual        !: unknown
```

* Hack that relies on line comments with GhostScript. Requires the parasite not to contain any new line, after encryption.

@ Hack relying on overwriting the DOS Header, therefore restricting the parasite space to offsets 2-60.

\+ Signature, comment declaration and length are two bytes apiece. To specify them all is six bytes, but if we round up the big-endian length and leave its low byte uncontrolled, we only need five. And if we leave the length entirely uncontrolled, we only need to fix four bytes.

Figure 3: Minimal start offsets of file formats, and exploitation via near polyglots



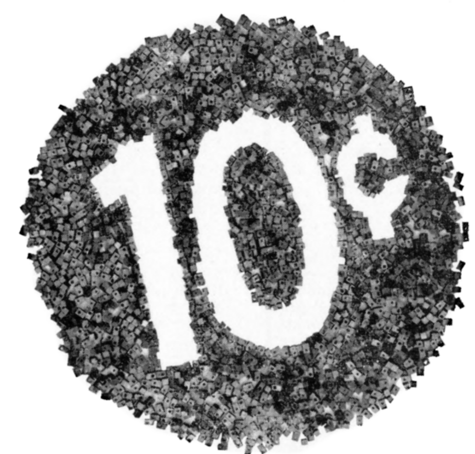
also gamble on the length and not bother to brute-force `XX` either.

Mitra can generate such files with the `--overlap` parameter. It keeps the overlap's content in the filename as well as the offsets where the content changes formats, to be re-used later by AngeCryption or TimeCryption scripts.

Ambiguity Files with different interpretations depending on parsers we now call *ambiguous* and previously called *schizophrenic*, werewolves or shapeshifters. There are plenty of sources of this ambiguity.

When a value such as a pointer never changes across standard files, it's tempting for a parser to simply ignore it. Putting some contents under unusual conditions while putting other contents under the typical conditions might reveal a difference between the two parsers.

Sometimes a value is represented twice. For example, a buffer with a declared length might also end with a null terminator. What if that terminator happens earlier than the declared length? Which length value is the real one? Or if you declare the same value twice, and there isn't an error, does the first or the second declaration take priority?



BIAX® MEMORIES NOW 10¢ A BIT
The new BIAX NANOLOK electrically alterable memory gives 3-Plus megacycle TRUE NDRO operation at the low cost of 10 cents per bit in 200,000 bit systems in quantity. NANOLOK is designed for commercial and industrial data use — but can be adapted readily to shipboard and mobile MIL-Spec environments.
BIAX used to be expensive, but not any more — not with NANOLOK.
Do your bit and write today for all the details on NANOLOK in Data File B-129.
RAYTHEON
RAYTHEON COMPUTER, 2700 South Fairview Street, Santa Ana, California 92704

If you corrupt a format on purpose and the parser tries to rebuild the file, how does it do it first? What if you put a valid file structure in a comment? Such recovery algorithms are typically not officially specified, so each developer might do it differently.

Some formats are extending older formats. Both the old and the new formats are present in the file. These formats are naturally ambiguous at a format level, and we might call them ambiguous polyglots.

A widespread example is the Portable Executable, defined as an extension of the DOS format. Preciously few PE files—such as `regedit95.exe`—have a meaningful DOS payload. Valid PE files are expected to just have the same DOS stub with no unique code.

Robert Xiao proved us wrong by crafting a universal Doom binary, which works from DOS 6 to Windows 10, both as valid DOS and PE payloads in the same file.[9] This is something like a father and son having the same name, with no distinctive suffix whatsoever.

Ciphertexts can be ambiguous too, even despite authenticated encryption! Check the TimeCryption article page 30 in this issue for abuses of GCM, GCM-SIV and OCB3.

Collisions PoC||GTFO 19:05 covered a lot of details for exploiting hash collisions. However tar.gz and DOCX (ZIPped XML)—which were initially thought to be unexploitable—are properly dealt with and explained in the Inside Out article on page 19.

Conclusion

With basic knowledge of file format identification and abuse, Mitra can try different strategies and generates many forms of file abuse: payload embedding, mock files, polyglots and pseudo-polyglots.

Pseudo-polyglots are the unified. form of file formats abuses to be combined with cryptographic operations. They include both AngeCryption, covering ECB, CBC, CFB, OFB modes, and TimeCryption, covering CTR, OFB, GCM, OCB3, GCM-SIV modes.

Extensions of Mitra might cover ambiguous files with standard strategies, hash collisions and hash collisions over different formats.

[9] `git clone https://github.com/nneonneo/universal-doom`

22:04 More Letters from Screwtape

by the Demon Wormwood, and certainly not by Manul Laphroaig

My good neighbors,

Some of you surely remember those letters from a certain Uncle Screwtape to his nephew Wormwood that C.S. Lewis published eighty some years ago. Though some discount those letters as apologetics, a believer's fictional account of demons discussing the best way to corrupt a well meaning (but poorly behaving) soul into perdition, I've often wondered with which modern sins Screwtape might be corrupting his patients these days.

Imagine my surprise when a mistake of the post sent the following to my door, which I reproduce faithfully and without redaction or comment.

–PML

– — — – — — — — – — –

My dearest nephew Malört,

I'm overjoyed to have heard that your patient still spends long, wasted days on social media, seeing that others have adventures but never participating in them first hand. The cat videos worry me a little, but so long as he can't give the cat noms or scritches I guess it can't feed his soul.

Do be careful, though. The same Internet that feeds your patient and endless supply of computer-generated voices reading forum posts over a video game is entirely capable of giving him more dangerous things. It can teach him to repair a car, and it has all the novels that we worked so hard to have censored back in my uncle Screwtape's day.

Censorship back then the real deal. We had so much fun having our patients light a pyre and tossing books into it, that we eventually forgot the whole point was to keep the books from being read. These days, the Opposition has perverted our fine tradition into something called Banned Book Week, where they give away free copies of the books we worked so hard to squash! One of them even portrays your great grandfather Behemoth as a pudgy cat, thrown out of street cars *after* paying his fare and getting into a shoot-out with the NKVD. How insulting.

Keep me apprised of your patient's progress, and be sure to watch for any signs of his finding anything useful out there.

In service of our Lord Below,
–Wormwood

Malört, we have to talk.

I read in your last letter that your patient has declared himself to be a "computer programmer," and that his television set has been off for a week while he repeatedly stumbled through the same chapter of a 21 Days book.

You needn't worry that two more weeks will destroy your careful work in maintaining his illiteracy, but two more months might see you transferred to Search Engine Optimization or some other department with no souls to corrupt. Pay attention!

A number of strategies might work here, but I heard of an excellent new one from your cousin, Legion. The idea is not to dissuade your charge from learning to program, but to slowly twist the idea of programming until he learns nothing useful.

Begin with the choice of language. Ages ago, we'd start endless debates about whether BASIC was harmful and whether Pascal was for idiots. That kept good engineers from learning to read spaghetti code or use a language with safe strings, but now language arguments are going out of fashion. No one cares to debate them endlessly, just as the humans have largely forgiven one another for using different text editors.

No, the trick these days is Artificial Intelligence. They might call it Large Language Models, and they recently called it Machine Learning, but the beauty is that the user *feels* like he's programming a computer while never actually writing any code.

So direct him back to the thirty-second videos, and whisper in his ear that "prompt engineering" is the bees knees. Get him to scroll endlessly, and know that if he does try some stuff on the prompt, he will never see the endless gigabytes of linear algebra beneath it all. It will feel like it makes sense, at least for thirty seconds, and then there will be another video.

Your uncle,
–Wormwood

Nephew Malört,

You are awfully worried that your patient has joined a hackerspace, and not without reason. After that gambit of yours to make him unemployed for a few months, he now has time on his hands to join a club, and there's danger that club might teach him something.

To prevent this, you just need to make the space a hassle for him instead of an inspiration. Take soldering for an example: if he keeps at it, he will quite soon become good at it. And if he gets good at it, he'll be able to assemble surface mount kits that the others cannot, which might give him the confidence to design his own. This can't be allowed to continue!

So whisper a little in his ear. Make him turn the iron too hot, or let it crust up overnight when no one is watching, to get those barnacle covered tips that make an expert struggle.

If he uses enough flux, tell him it's too much, and if he really uses too much that's not a problem, so tell him it's an embarrassment to his grandfather's grandfather that so much flux is wasted on an LED throwie.

When he makes a reparable mistake, like using the wrong resistor value, be sure to fill him with shame. And if no magic smoke escapes, double up on the shame, as if he's the very first to have an LED that's a little bright or a little dim on account of its series resistor.

When you come visit the eighth circle, we might discuss other ideas. My grandfather once got some monks to fight for a century over whether they were "of" the Enemy or "with" the Enemy, and I bet we could trigger a similar fight between 60/40 and lead-free solder.

–Wormwood

Nephew Malört,

In your last letter, you made smug references to your patient "blinking LEDs with an Arduino," and I worry that you don't understand how serious this might be. Arduinos might use 8-bit microcontrollers, but they are programmed in an easy dialect of C!

And worse, Arduino is a dialect of C for which a thousand convenient examples are shared without intimidation. If your patient first blinks LEDs, he might later blink them in sequence, or display the temperature on them. Pretty soon he might extend the examples with original code, and I worry that you might make side remarks about this too, ignoring the danger.

For the danger in these blinky lights is that they are projects. In the same way that one might fail to learn Spanish from a book for years, but quickly learn the basics when there's no other way to buy food, a project has the power to make a boring language exciting. Your same patient—the one who a few short months ago wasted a month without getting to the second chapter of his 21 Days book—might soon finding himself *giving a shit* about C.

Once he gives a shit, how long do you expect it might take him to learn a language with just 32 keywords?

–Wormwood

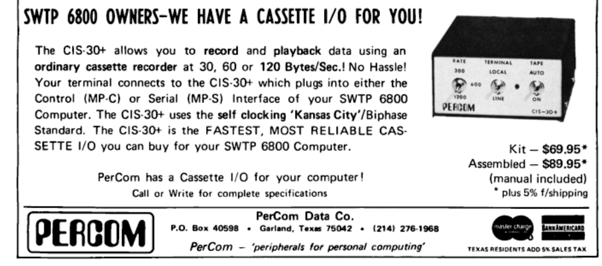
– — — – — — — — – — –

Uncle Wormwood,

I write to you just before sending my resignation to Our Father Below. The Patient has indeed picked back up his 21 Days book, and this afternoon I overheard him explaining pointer arithmetic to a friend.

I'm ashamed to say, the explanation was right.

Your disgraced nephew,
–Malört



22:05 Inside Out; or, Abusing archive file formats.

by Ange Albertini

We have previously demonstrated hash collisions in documents with blocks of 64 bytes, such as the great MD5 pileup in PoC||GTFO 19:05. This used colliding, aligned blocks in `pocorgtfo19.pdf` to match a hash of `pocorgtfo19.exe`, `pocorgtfo19.png` and `pocorgtfo19.mp4`. That is to say, these files were not identical, but they did share an MD5 hash.

This research started with an incorrect assumption that Zip, TAR, and GZIP couldn't be generically exploited with collisions. Even with the almighty chosen-prefix collisions, I thought that Zips *may* not work, XML will *never* work, and GZIP will *always* trigger a warning.

Zip is the most collision unfriendly of standard file formats: bottom-up, pointers everywhere, duplicated data... Since they are officially parsed bottom-up, you can't even use a Chosen Prefix Collision on a pair of Zip files if their size difference is bigger than 64 kb, as the EoCD (end of Central Directory record) of the smaller archive will be too far from the end of the file to be found, thus making the file invalid.

On top of that, some critical data (such as file length, name, and content CRC32) is duplicated in the Local File Headers and in the Central Directory for a given file, which means it is present before and after the file contents—thus preventing any generic exploitation.

And unlike most archive formats, Zip is a tree of pointers between structures instead of sequences, so any size change of file content will propagate on the rest of the file: the last structure of the file contains a pointer and the number of archived files.

XML files also don't play nice with collisions: `CDATA` comments are defined in XML files, but they have to use the defined encoding, which is incompatible with the randomness of collision blocks.

XML files don't tolerate appended data either. It's another totally collision-unfriendly format.

DOCX files are Zip archives containing XML files and various data files, such as JPEG and PNG images.

[10] `unzip pocorgtfo22.pdf APPNOTE-1.0.txt`
[11] `git clone https://github.com/resilar/crchack.git`

Root file In DOCX files, the `/_rels/.rels` file plays a very special role. It's the *root* of the document, which points to other XML files of the document. It defines the *relationships* between the files.

You can move the files around provided you update the root, which requires a hard-coded path and filename in the archive. You can also make *two documents co-exist* in the same archive, pointing to either in the root file. A valid strategy to generically collide two documents seems possible.

Collision blocks You can't store the collision blocks after the XML content, since that would invalidate the root file's XML structure. And we can't easily forge the CRC of collision blocks, so we can't store them in the contents of a dummy file.

However, we can store the collision blocks in the Extra Field of a file, since Extra Fields don't have a CRC. Extra Fields were defined in 1990, in the very first version of the specifications.[10] They are commonly used and very extensible, so many implementations both ignore this field and preserve it. Extra Fields are stored before file contents, so they can't be stored in the Local File Header of the root; a dummy file stored after the root file can be used as a host for them.

It's easy to force the same length for the root file. We just need to choose two close paths for each document. Storing them rather than compressing them guarantees the lengths to be identical and predictable.

CRC You need to keep the root file CRC constant despite the collision blocks, since the CRC is duplicated near the end of the file in the Central Directory.

Forging a CRC is easy, but CRCHack makes it super easy![11] Just specify the bits you want, and it instantly gives you the requested output with the requested CRC32 without any encoding violation.

As an example, we now demonstrate forging a CRC with ASCII characters.

```
$ cat ascii
```

```
<!--ABCDEF-->
$ crchack \
  -b 4.0:+.8*6:1 -b 4.1:+.8*6:1 \
  -b 4.2:+.8*6:1 -b 4.3:+.8*6:1 \
  -b 4.4:+.8*6:1 -b 4.5:+.8*5:1 \
   ascii 0xdeadf00d
<!--tuI_\Y-->
```

Only with the uppercase bit of letters:

```
$ cat letters
<!--THESEKINDSOFCRCAREVERYIMPRESSIVE-->
$ crchack -b 4.5:+.8*32:.8 letters 0xcafebabe
<!--thEsEKIndsOFcRcAReVEryiMPREssIVe-->
```

So now we have two versions of the root files, with the same CRC, the same length, and via a dummy file with Extra Field containing HashClash collision blocks: the two Local File Headers that give the archive the same MD5.[12]

Results Unlike most reusable generic collision prefixes with a header and no body, this actually gives us two reusable generic collision pre-archives that are totally valid and manipulatable with standard tools. Provided you're careful with timestamps—either ignoring them in the source files or recompiling within two seconds—doing the same operations on both pre-archives will maintain the equality of hash values of both files, which is nice and very unusual.

Even better, deleting any archived files beside the root and the dummy collision block file will revert to the original hash values without any further modification required! Who would have expected that standard Zip tools could give you predictable hash values?

```
$ md5sum docx*zip
6c33d52590ff0bb0cc8cdafe6aa5153b *docx1.zip
6c33d52590ff0bb0cc8cdafe6aa5153b *docx2.zip
$ zip -oXll docx1.zip zinsider.py
  adding: zinsider.py (deflated 64%)
$ zip -oXll docx2.zip zinsider.py
  adding: zinsider.py (deflated 64%)
$ md5sum docx*zip
d12044feee801ad0530a911fa7f18db5 *docx1.zip
d12044feee801ad0530a911fa7f18db5 *docx2.zip
$ zip -d docx1.zip zinsider.py
deleting: zinsider.py
$ zip -d docx2.zip zinsider.py
deleting: zinsider.py
$ md5sum docx*zip
6c33d52590ff0bb0cc8cdafe6aa5153b *docx1.zip
6c33d52590ff0bb0cc8cdafe6aa5153b *docx2.zip
```

Supported formats This trick is applicable to any file format made of a Zip-ed XML with a root file. It works for `.docx`, `.pptx`, and `.xlsx` from Office, for the open container format in ePub, and for other open packaging conventions, such as `.3mf` for 3D manufacturing and the XML Paper Specification, `.xps` and `.oxps`.

Corkami collisions' `zInsider` makes it possible to instantly collide any of these formats, with precomputed prefix archives.[13]

This is easy to extend to any other similar format, but a new prefix pair must be recomputed for any new format.

Some formats like Quake's PK3 aren't exploitable: they don't have a root file to abuse. The Open Document Format requires their root file to mention every other file, which isn't generic. APK, JAR, and XPI are even worse: they require all the other files' hashes!

Gzip

TAR files have no room for any abuse: pure sequences of headers with hardcoded size and offsets, then file contents. No declared lengths, no skippable content. You can use chosen-prefix collisions on them, but that's it: nothing generic.

Gzip doesn't seem to be playing nice with hash collisions either: any extra data is placed before the compressed file contents, and appended data typically triggers a warning and is not taken into account for parsing anyway. Gzip collisions are possible, but not in a generic way.

However, while most Gzip files start with the typical `1F D8` structure—called a *member*—it's actually specified that a Gzip file can contain several of these members, in which case the data of each will be decompressed and concatenated. So a member with no compressed data but with extra data acts as a comment that can be parasitized, albeit quite a complex one.

Since the length of the Extra Field is stored on two bytes in little-endian before the Extra Field itself, it's even exploitable with UniColl![14]

[12] `git clone https://github.com/cr-marcstevens/hashclash`
[13] `git clone https://github.com/corkami/collisions; find collisions -name zinsider.py`
[14] `git clone https://github.com/corkami/collisions; find collisions -name unicoll.md`

So a generic reusable hash collision for Gzip is actually possible via a classic sequence of comments. First one comment to align the rest of the file to collision block boundaries, then one comment whose length is variable—its encoded value will be overlapping with one of the differences in the collision blocks—and then we start two chains of comments to toggle one payload or the other, exactly like we did for JPEG, MD5, or SHA1.

Colliding GZIPs like JPEGs Like JPEG, we have this limit that extra field can't be bigger than 64 kb, but recompressing data in chunks of 64 kb is much easier with Deflate than with JPEG! Since the decompressed data of all members is concatenated, we just need to cut the archived data in chunks.

This idea isn't new. Some formats like BGZIP (2008) chunk the data in several members and store an index in the extra field, making it easier to decompress some contents separately while maintaining a standard Gunzip-compatible structure. This is a common source of multi-member Gzip files.

So it gives us reusable hash collisions for anything that relies on Gzip as outer encryption, such as `.tar.gz` or SVGZ. As long as the data is decompressed, the structure of the outer archive can be freely modified.

However, some programs like Inkscape use their own lightweight implementation of Gzip, which doesn't support files made of several members, so our collision strategy will not work in these exceptional cases.

Conclusion

While Zip, XML, GZIP, and TAR seemed very hostile to collisions, combining several tricks made it possible to get generic reusable hash collisions for GZip archives (`.tar.gz`) and Zipped XML files with a root, such as DOCX files.

The strategies are very different, even if they both rely on the extensible Extra Field which is similar in both formats. For DOCX, it's a merge of two documents inside the same Zip, with two versions of the same root file. For Gzip archives, Extra Fields are used as comments, and two independent archives are interleaved via two chains of skip and data.

Other formats aren't playing that nice: Bzip2 is a pure compressor, bit-based with only bit alignment, and no padding and no form of comments. Other formats such as XZ, AR or Compress (`.Z` archives) are just too simple for any exploitation. RAR applies CRC16 to headers, which does not help our cause.

Thanks for Yann Droneaud for the TAR.GZ challenge, and Philippe Lagadec for the DOCX challenge!

Still using MD5? It might feel useless to still care about MD5, but as MD5, SHA1, and SHA2 use the same construct, exploits of hash collisions via file format tricks will be re-usable for other hash collisions while being cheaper to pull off with MD5. These techniques would work for SHA1 via the Shamble attack too, except that it costs $45,000 USD to compute it. And at least, MD5 is still widespread enough that it has enough targets to attack in practice, unlike MD2 and MD4!

You might be tempted to still use MD5 to designate a file, but using MD5 will expose you to all kinds of tricks and confusion that SHA2 or Blake2 don't.

Fastcolls are very quick to compute and can be chained, providing one bit of stored data while keeping the MD5 constant.[15] They will make it trivial to watermark a file, and a very short shellcode can easily detect which version of the file is running, then adjust its behavior accordingly. Using a stronger algorithm would prevent any possible pranks or confusion, at least for some years until we get better collisions.

Bonus: ZGIP Zip can use Deflate among other compression algorithms. On the other hand, Gzip only uses Deflate.

Both are wrapping Deflate data around different structures that are not compatible. By abusing structures, it's possible to make ZGIP, a chimera of Zip/GZIP: a polyglot file sharing the compressed data.[16]

By abusing Deflate stored blocks and dummy members, it's even possible to partially hide some data from the other format, even if they belong to the same stream.

In short, this is just going the extra mile to prove that GZIP is not a wrapper around Zip, nor Zip is a wrapper around GZIP.[17]

[15] `git clone https://github.com/brimstone/fastcoll`
[16] `git clone https://github.com/corkami/pocs; find pocs -name zgip`
[17] `https://speakerdeck.com/ange/gzip-equals-zip-equals-zlib-equals-deflate`

Bonus: The craziest colliding file The latest advanced MD5 manipulation is a very clever ZStandard+Tar hashquine+polyglot by David 'Retr0id' Buchanan,[18] also known for his beautiful PNG hashquine.[19]

It can either be just a Zst file, but also a Tar.zst, so the Tar header can be toggled on or off, as well as the complete tar checksum. To be a reusable hashquine, it's able to output any MD5 and Tar checksum while keeping the whole file's MD5 constant.

The same prefix is reusable in three different ways. First it can be a pure ZStandard file hashquine.

```
$ md5sum hashquine.zst
720ca7f6842f1a608fcb924f5811ebb9 *hashquine.zst

$ zstd -cd hashquine.zst
The MD5 of hashquine.zst is:
720ca7f6842f1a608fcb924f5811ebb9
```

Second, it can be a Zstandard(tar) file.

```
$ md5sum hashquine.tar.zst
703911cf9e409965cebd05392acc1503 *hashquine.tar.zst

$ tar -Oxf hashquine.tar.zst hash.md5
The MD5 of hashquine.tar.zst is:
703911cf9e409965cebd05392acc1503
```

Finally, it can be a self-checked "auto-manifest" Tar.zst.

```
$ md5sum self.tar.zst
f068d54fabb12dbb1b359745a80d78fc *self.tar.zst
~
$ tar -xvf self.tar.zst
x hash.md5
x hello.txt

$ cat hash.md5
f068d54fabb12dbb1b359745a80d78fc *self.tar.zst
ed076287532e86365e841e92bfc50d8c *hello.txt

$ md5sum -c hash.md5
self.tar.zst: OK
hello.txt: OK
```

The whole prefix uses 653 Unicolls to toggle Zstandard frames and output optional contents after decompression.

For the optional Tar Header (generic for any `hash.md5` contents), it uses one frame for the constant Tar header start, 8∗11 frames for the `hash.md5` file size in octal, one frame for the constant Tar timestamp `14412572240`, 8∗6 frames for any tar header checksum in octal, and one frame for the rest of the tar header.

For the optional text prefixes in the file contents, it uses one frame for the constant prefix of "The MD5 of hashquine.tar.zst is" and other for "The MD5 of hashquine.zst is" in ASCII. Finally, it uses 32∗16 collisions for all nybble possibilities of an MD5 hash.

Bonus: Wordpad weird files Colliding `.docx` files will show the same document with Microsoft Wordpad. It turns out that Wordpad ignores the root files entirely, and just locates the document file via the Content Types files. Really!

As you would expect with such sloppiness, it doesn't check if all files in the archive are declared in the Content Types file, which can turn any Zip archive into a very weird `.docx` that is Wordpad-only with just two XML files. Sadly, this issue being far from a standard file. Wordpad is confused as it should be, and we can't make this issue a Wordpad-compatible DocX file too. Extract an example from this PDF's attachments.[20]

[18] `git clone https://github.com/corkami/collisions; find collisions -name hashquines`
[19] `unzip pocorgtfo22.pdf retr0id.zip; unzip retr0id.zip hashquine_by_retr0id.png`
[20] `unzip pocorgtfo22.pdf mini.docx`

22:06 Mitigations are a reverser's friend; or, Abusing XFG

by Aleksandar Nikolic

Control flow integrity protections, with its various implementations, have been the latest iteration of compiler mitigations for memory corruption exploits. They hope to make code reuse attacks more difficult or impossible. Implementation details vary, but all boil down to restricting possible valid targets of indirect calls. LLVM's is called "Control Flow Integrity," Grsecurity has "Reuse Attack Protector" and Microsoft's is called "Control Flow Guard" (CFG).

The core idea behind Microsoft's CFG is ensuring that function pointers can only point to valid function entry points before being used to perform a function call. The compiler inserts checks that, during runtime, inspect every indirect call instruction and terminate the process if the target isn't a valid and known function start.

Putting aside the completeness or effectiveness of this mitigation, let's ask whether we can glean some extra information about the code itself by the presence of these checks. As Deroko points out in Control Flow Guard Instrumentation,[21] CFG mechanisms can serve as a way to hook all indirect calls in a binary without specifically looking for them in advance. They can also precisely identify function entry points, which is not always a trivial task.

With the release of Windows 11, Microsoft is introducing another iteration of control flow integrity mitigation called "eXtended Flow Guard" or XFG. In short, it further restricts targets of indirect calls to not only valid function entry points, but to a subset of functions that have a particular signature consisting of return value type, number and types of parameters and other function properties.

Surely, this added metadata can somehow aid us in our reverse engineering process. To see how, we'll need to understand the implementation details.

What is XFG and how it works

Extended Flow Guard is introduced as a compiler extension that can be enabled via `/guard:xfg` switch that is available in MS's C and C++ compilers since at least the 19.27.29112 version of Visual Studio 2019. It hasn't seen full support or much public use until release of Windows 11. Consider an example:

```
int test( ){
  return 0;
}

int (*cfgTest[1])() = {test};

int main(){
  cfgTest[0]();
}
```

This code has a simple function pointer array `cfgTest` and makes a call to `test` using that function pointer. If compiled with `cl /Zi /guard:xfg simple.c` its assembly looks a little odd.

```
1  sub     rsp, 38h
   mov     eax, 8
3  imul    rax, 0
   lea     rcx, cfgTest
5  mov     rax, [rcx+rax]
   mov     [rsp+38h+var_18], rax
7  mov     r10, 0D30527475E523070h
   mov     rax, [rsp+38h+var_18]
9  call    cs:__guard_xfg_dispatch_icall_fptr
   xor     eax, eax
11 add     rsp, 38h
   retn
```

[21] `unzip pocorgtfo22.pdf cfghook.zip`

This is some peculiar code. There is no indirect call to function `test`, rather there's a call to `__guard_xfg_dispatch_icall_fptr` with certain arguments. The function pointer is actually saved in `rax` and an odd-looking constant is moved into `r10` before `__guard_xfg_dispatch_icall_fptr` is called. This odd-looking constant is what we will call an XFG hash. Interestingly, if we take a look at `test` function's prologue on page 25, we'll see (almost) the same data.

Long story short, before invoking the target function, `__guard_xfg_dispatch_icall_fptr` will check that the hash in `r10` matches the hash located right before the function. If they don't match, process is terminated.[22]

This ensures that only legal target functions can be executed at this particular indirect function call. The next obvious question is: how is this function hash derived? That brings us to the core idea behind XFG.

If we think about it, no matter how an indirect call instruction happens to be generated by the compiler, several things are true for all the possible, valid, target functions in a valid program. All possible target functions must have the same number of arguments, the same calling convention, same argument types, same return value type and so on. Even if the compiler doesn't know of all possible target functions in advance, it must know all of these facts about those targets. It can, then, generate a unique representation of those facts when it encounters an indirect function call. On the other hand, for every function that could be a possible target for indirect call, the same unique representation can be calculated and those two can be compared during runtime.

This unique representation of function prototype information is what constitutes an XFG hash.

How is an XFG hash generated?

Francisco Falcon over at Quarkslab has already done the hard work of reverse engineering most of XFG internals. Their extended writeup provides a number of examples.[23] XFG hash generation happens in the `cl.exe` compiler's frontend `c1.dll` and revolves around gathering function prototype information and using the SHA256 hashing algorithm on it while following certain rules. A list of function properties that figure into the XFG hash is (as far as C code is concerned at least) as follows:

- number of arguments
- the types of individual arguments
- type of return value
- whether the function is variadic or not
- the calling convention

When preparing to calculate the hash, each of these is represented in a specific way. Some are simple constants, while others have more structure and are often recursively defined. For example, the number of arguments is just represented as a 32-bit integer, the calling convention appears to be a 16-bit constant, and variadic is one byte boolean. Return value and argument types, on the other hand, are more complicated.

Those consist of values specifying type qualifiers (`const`, `volatile`), type groups (primitives, pointers, structs/unions/enums), and values according to the type group. Calculating values for primitive types are the simplest and are just a table lookup:

```
"void"                     :0xe,
"char"                     :0x1,
"signed char"              :0x1,
"unsigned char"            :0x1,
"__int8"                   :0x1,
"char8_t"                  :0x1,
"__int16"                  :0x6,
"short int"                :0x6,
"unsigned short int"       :0x86,
"float"                    :0x11,
"int"                      :0x7,
"__int32"                  :0x7,
"unsigned int"             :0x87,
"long int"                 :0x10,
"unsigned long int"        :0x8a,
"double"                   :0x12,
"__int64"                  :0x8,
"long  double"             :0x12,
"long long int"            :0x8,
"unsigned long long int"   :0x88,
"unsigned long long"       :0x88,
```

Notice that there are several distinct primitive types that have the same value. Structs, unions, and enums are treated the same, and their actual (verbatim text) names are included as part of a hash calculation.

[22] A great in-depth description from Connor McGar is available as *Exploit Development: Between a Rock and a (Xtended Flow) Guard Place: Examining XFG.*

[23] See *How the MSVC Compiler Generates XFG Function Prototype Hashes* by Francisco Falcon.

```
.text:0000000140001008                 dq 0D30527475E523071h
.text:0000000140001010
.text:0000000140001010 ; ============ S U B R O U T I N E ===============================
.text:0000000140001010
.text:0000000140001010
.text:0000000140001010 ; int test(...)
.text:0000000140001010 test            proc near        ; DATA XREF: .rdata:__guard_fids_table
.text:0000000140001010                                  ; .data:cfgTest
.text:0000000140001010                 xor     eax, eax
.text:0000000140001012                 retn
.text:0000000140001012 test            endp
```

Figure 4: `test` Function's Prologue

Pointers of any kind are the most complicated, as their value is the hash of the type they point to, requiring recursive evaluation.

This can look a bit confusing and — although it's covered in great detail in the referenced Quarkslab article — we'll illustrate the process with the simplest example. We'll add a void pointer as an argument to `test` from before:

```
int test(void *arg);
```

First, there's only a single argument to this function, so we will append "`\x01\x00\x00\x00`" to our data to be hashed (`data0`). Second, we need to consider function arguments, calculate their hashes, and append them to data to be hashed. There is only one argument and it's a pointer without qualifiers. Starting a new hash (`data1`), we append "`\x00`" for qualifiers, "`\x03`" for type group but then we need to consider the type of pointer and calculate that hash separately. Starting yet another hash calculation (`data2`), we append "`\x00`" for qualifiers, "`\x01`" for type group and finally "`\x0e`" for primitive type. Calculate the SHA256 of `data2` and append its first 8 bytes to `data1` that completes necessary data for calculating first argument hash. Hash `data1` and append the first 8 bytes to `data0`. That concludes the argument part of the hash. Next is whether the function is variadic, so we append "`\x00`" and what the calling convention is, which defaults to just "`\x01`". The last segment is the return value type which is an integer primitive, so it's simply "`\x00`" for qualifiers, "`\x01`" for type group and finally `0x7` for a primitive type. The hash of that is appended to `data0`.

Putting that together gives us the following, with all SHA256 results truncated to the first eight bytes.

```
sha256("\x01\x00\x00\x00"
  +sha256("\x00\x03"+sha256("\x00\x01\x0e"))
  +"\x00"+"\x01"+sha256("\x00\x01\x07"))
```

After some final transformations, the result of the operation is the "`719a5e103606e1b2`" value that appears before the `test` function in the binary.

An implementation of this algorithm, in Python, that parses a given C function prototype and generates its corresponding hash can be found as an attachment.[24]

INFORMATION SYSTEMS SCIENTISTS AND ENGINEERS

Bendix Research Laboratories has excellent career opportunities for B.S., M.S., and Ph.D. graduates with 2 to 15 years experience in one or more of the following key areas:

- **ARTIFICIAL INTELLIGENCE,** pattern recognition, trainable systems, adaptive logic.
- **COHERENT OPTICAL PROCESSING,** spatial filtering, optical correlation, electro-optical systems and components.
- **DIGITAL COMPUTER DESIGN,** systems analysis, logic and circuit design, real-time computer control.
- **ANALOG COMPUTERS AND SYSTEMS,** information theory, control theory, circuit and servo analysis, correlation techniques.
- **REAL-TIME COMPUTER PROGRAMMING,** mathematical analysis, scientific computing.

Assignments involve research and development in automatic extraction of information from photographic records, image processing and analysis, adaptive and trainable control systems, digital and hybrid computing techniques, and advanced real-time computer control applications.

Interested individuals are invited to call collect or to send a resume to our Personnel Director.

Research Laboratories Division
Southfield, Michigan • (313) 353-3500
An equal opportunity employer

[24] `unzip -p pocorgtfo22.pdf xfg-scripts-args.tgz | tar -xzvf- gen_hash_from_ast.py`

Using XFG to resolve indirect jumps

Now that we know how XFG works, we can consider how it can be of use as a reverse engineering aid.

The first, and most obvious idea is that it can reduce the uncertainty of analyzing indirect calls. Since all indirect calls in an XFG-protected binary will inevitably be dispatched through `__guard_xfg_dispatch_icall_fptr` that must match callsite's hash and target function's hash, it should be possible to enumerate all possible targets completely statically (assuming all possible linked code is known/available for analysis).

Let's illustrate this with an example. Throughout the rest of the article, we'll use `ntdll.dll` binary from Windows 11 for illustrations and testing. If we go to function 'LdrQueryProcessModuleInformationEx' and take a look at the following piece of assembly:

```
18000174e  488d04bf
  lea  rax, [rdi+rdi*4]
180001752  49ba7048da56963e...
  mov  r10, 0x85f13e9656da4870
18000175c  498b44c118
  mov  rax, qword [r9+rax*8+0x18]
180001761  ff15a9181900
  call qword
    [rel __guard_xfg_dispatch_icall_fptr]
    {j_sub_1800aa130}
180001767  4c8d0df2b71200
  lea  r9, [rel data_18012cf60]
```

While we don't know without debugging what possible target this XFG dispatch call might have, we can see that its hash must be `0x85f13e9656da4871` (the 1 is added at the end of the supplied hash by dispatcher). If we search the binary for functions that have this XFG hash, we'll find many results: `LdrQueryModuleInfoLocalLoaderUnlock`, `LdrShutdownThread`, `LdrShutdownProcess`, `RtlDetectHeapLeaks`, `TpTrimPools`, `RtlCleanUpTEBLangLists`, `RtlFreeThreadActivationContextStack`, `LdrProcessInitializationComplete`, `RtlFlushHeaps`, `RtlReleasePebLock`, `RtlAcquirePebLock`, `LdrFastFailInLoaderCallout`, ...

Obviously, from the function names, not all of these make sense as possible targets for this indirect call because of their differing semantics, but there's a good chance that all with `Ldr` prefix are actual possible targets.

Why are there so many hash hits that are unlikely to be real targets? It's probable that the target function prototype in this case is very simple, and matches many other functions. In fact, hash `0x85f13e9656da4871` represents the simplest possible case of 'void fname()'. As another example, the `TppCallbackEpilog` function has the following indirect call:

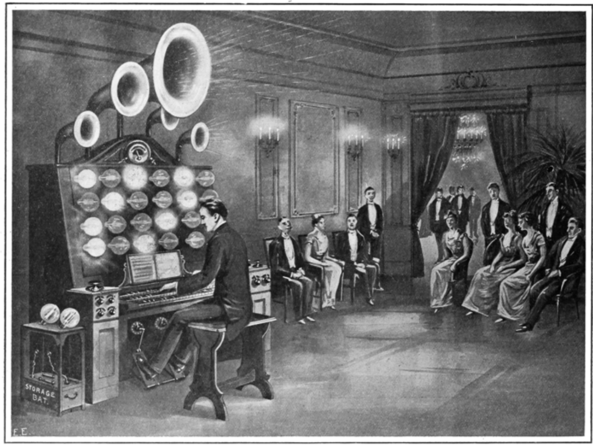
```
18001766e  488b8eb8000000
  mov  rcx, qword [rsi+0xb8]
180017675  4c89aeb8000000
  mov  qword [rsi+0xb8], r13  {0x0}
18001767c  488b4108
  mov  rax, qword [rcx+0x8]
180017680  49ba70125178f527...
  mov  r10, 0xa6d127f578511270
18001768a  488b4008
  mov  rax, qword [rax+0x8]
18001768e  ff157cb91700
  call qword
    [rel __guard_xfg_dispatch_icall_fptr]
    {j_sub_1800aa130}
```

Looking up the target hash, `0xa6d127f578511271`, in the binary yields: `TppSimplepFree`, `TppWorkpFree`, `TppAlpcpCallbackEpilog`, `TppJobpCallbackEpilog`, `TppFreeWait`, `TppTimerpFree`, `TppIopFree`, `TppAlpcpFree`, `TppJobpFree`, `TppWorkCancelPendingCallbacks`, `TppIopCancelPendingCallbacks`.

All of these look like possible real targets given their context.

So while not completely precise, this simple static analysis that relies on XFG hashes definitely sheds some light on indirect calls that might otherwise remain completely unresolved.

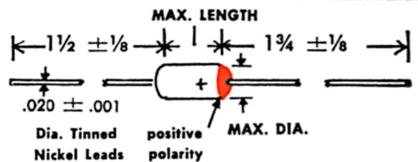
Attached is a Binary Ninja plugin that annotates indirect calls with information gained by XFG analysis.[25]



[25] `unzip -p pocorgtfo22.pdf xfg-scripts-args.tgz | tar -xzvf- xfg_analyzer.py`

ECONOTAN

SOLID TANTALUM CAPACITORS



METAL CASE			POLYESTER SLEEVE INSULATION		
SERIES	MAX. DIA.	MAX. LENGTH	SERIES	MAX. DIA.	MAX. LENGTH
CT	.090	.240	CT	.095	.260
CM	.128	.300	CM	.133	.320
CL	.175	.325	CL	.180	.325

SMALL SIZE

WITH MAXIMUM CAPACITANCE

LONG SHELF LIFE

ALL DRY CONSTRUCTION

LOW COST

WELL BELOW 20¢ EACH

IN PRODUCTION QUANTITIES

AVAILABLE RATINGS

STANDARD CAPACITANCE TOLERANCE ±20%

PART NUMBER	CAP MFD	WV DC	MAX DF	MAX IL
CT683	.068	25	.08	1.0
CT104	.10	25	.08	1.0
CT154	.15	25	.08	1.0
CT224	.22	25	.08	1.0
CT334	.33	25	.08	1.0
CT474	.47	25	.08	1.0
CT684	.68	25	.08	1.0
CT105	1.0	25	.08	1.0
CT155	1.5	20	.08	1.0
CT225	2.2	15	.08	1.0
CT335	3.3	10	.08	1.0
CT475	4.7	6	.10	1.0
CT685	6.8	4	.12	1.0
CT106	10.0	2	.12	1.0

PART NUMBER	CAP MFD	WV DC	MAX DF	MAX IL
CM155	1.5	25	.08	1.0
CM225	2.2	25	.08	1.0
CM335	3.3	25	.08	1.0
CM475	4.7	20	.08	1.0
CM685	6.8	15	.08	1.0
CM106	10.0	10	.08	1.0
CM156	15.0	6	.10	1.0
CM226	22.0	4	.12	1.0
CM336	33.0	2	.12	1.0
CL475	4.7	25	.08	2.0
CL685	6.8	25	.08	2.0
CL106	10.0	20	.08	2.0
CL156	15.0	15	.08	2.0
CL226	22.0	10	.08	2.0
CL336	33.0	6	.10	2.0
CL476	47.0	4	.12	2.0
CL686	68.0	2	.12	2.0

Visit us at
BOOTH 2A-48
IEEE Show

TECHNICAL BULLETIN AVAILABLE ON REQUEST

COMPONENTS, INC.
SMITH STREET, BIDDEFORD, MAINE
PHONE A.C. — 207-284-5956
PLANTS AT BIDDEFORD AND KENNEBUNK

Brute forcing XFG hashes for function prototype recovery

Another, more involved, idea stems from the fact that XFG hashes aren't random and actually encode function prototypes. Surely, there would be a way to recover at least some of that information and make use of it.

While it is not possible to reverse the hash back to function prototype directly, it is perfectly feasible to precompute a lookup table for all possible function prototypes (up to certain number of arguments). If we ignore structs, unions and enums for a second, there are only a fairly small number of primitive types. In fact, if we remove the duplicates, there's a total of only 12 primitive types (with distinct values as far as XFG generation is concerned). Adding in type qualifiers (`const`, `volatile`) and pointers, a bit of simple combinatorics tells us that total number of all possible function prototypes is roughly $(12 * 3)^{num_args} + 1$.

This gets big very fast as we increase the number of arguments, but the whole list is precomputed in minutes for functions up to three arguments.

```
import sys
import itertools
from jinja2 import Template
types = ["void", "char", "short int",
    "unsigned short int", "float",
    "int", "unsigned int", "long int",
    "unsigned long int", "double",
    "long long int", "unsigned long long"]
# add all types as pointers
types += [x + " *" for x in types]
# and as consts
types += ["const " + x for x in types]
# and as volatiles
types += ["volatile " + x for x in types]

j2_template = Template("""
{{ret_type}} fname( {%- for param_type in
    param_types -%} {{param_type}} arg{{loop
    .index}}{{ "," if not loop.last }} {%-
    endfor -%});
""")

max_func_params = 3
f = open(sys.argv[1], "w")
i = 0
for ret_type in types:
  for pn in range(4, max_func_params+1):
    for c in itertools.product(types,
                               repeat=pn):
      f.write(j2_template.render({"ret_type"
    : ret_type, "param_types": c}))
  i+=1
f.close()
```

This code uses a jinja2 template to generate an exhaustive list of all possible function prototypes starting with given primitive types. These generated prototypes can then be fed into the hash generation algorithm to compile a lookup table.

So, does this work? We'll test this on `ntdll.dll` again. This particular version of the DLL has a total of 1564 functions that have an XFG hash associated with them. Out of those 1564, there are a total of 995 unique XFG hashes. After lookups, this simple matching has identified function prototypes for 131 unique hashes, corresponding to a total of 294 functions!

By simply precomputing all possible function prototypes up to three parameters (using nothing target specific, only primitive types) we were able to recover precise function prototypes for about 13% of unique hashes in `ntdll.dll`. Figure 5 has some examples.

The proof-of-concept works, but there are a couple of reasons why we didn't get a higher hit rate. First and most obvious is that many functions simply have more than three arguments, but even bigger factor is the fact that `ntdll.dll` code heavily relies on use of structures, enums, and structure pointers. Since hashes for struct, union, and enum types include their names directly, straight up brute forcing isn't practical, but seeding certain (domain specific) names would greatly increase the hit rate. XFG hash calculation implementation supports structs in function prototypes, and since structs, enums, and unions are treated the same, all we need to do to add struct names is to expand the list of primitive types. Adding `struct in_addr` to list of primitive types leads to following result:

```
7139d252a1b76de8 char *func(
  const struct in_addr *arg1, char *s)
```

This calculated hash matches the XFG hash for RtlIpv4AddressToStringA. By adding target specific, commonly used, structs to prototype generation we can greatly increase the number of found hashes at the expense of a larger lookup table. Since structures and other type information are sometimes publicly available even if function prototypes are not, this allows for very precise function prototype recovery.

How do we know that these results are actually correct? Let's take another look at an example where we do know the function prototype. Function 'RtlSetUserValueHeap' has four arguments. Binary Ninja guesses its prototype to be:

```
void* const* RtlSetUserValueHeap(
  int64_t arg1, int32_t arg2,
  int64_t arg3, int64_t arg4);
```

Similarly, IDA guesses:

```
char __fastcall RtlSetUserValueHeap(
  __int64 a1, unsigned int a2,
  __int64 a3, __int64 a4)
```

This function's XFG hash is `0xc76c3600585af171` and a lookup reveals the following function prototype:

```
char RtLSetUserValueHeap(
  void *arg1,unsigned long int arg2,
  void *arg3, void *arg4);
```

Notice how both Binary Ninja and IDA cannot know that some of the arguments are pointers. This simple fact adds a lot of information that greatly aids further function analysis and decompilation. And what about correctness? While source for 'RtLSetUserValueHeap' isn't available, it is reimplemented in ReactOS where its function prototype is:

```
BOOLEAN
NTAPI
RtlSetUserValueHeap(
  _In_ PVOID HeapHandle,
  _In_ ULONG Flags,
  _In_ PVOID BaseAddress,
  _In_ PVOID UserValue
);
```

While the prototype gathered from XFG analysis lacks some extra annotations, the types themselves match precisely!

In Conclusion

Even though mitigations like XFG pose a real challenge when it comes to exploitation, it sometimes pays off to take a step back and consider the possible side effects that can be handy in other ways. We've shown that a very simple lookup table can recover a treasure trove of information that can be helpful when reverse engineering an XFG-protected binary. As XFG adoption spreads to code other than Microsoft's, this can definitely lead to some interesting discoveries.

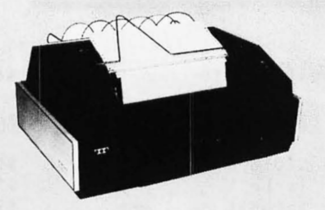
```
char RtlGetSecurityDescriptorRMControl(void *arg1, char *arg2);
unsigned long int RtlNumberOfSetBitsUlongPtr(unsigned long long int arg1);
char RtlEqualWnfChangeStamps(unsigned long int arg1, unsigned long int arg2);
unsigned long int RtlSetProxiedProcessId(unsigned long int arg1);
void RtlWnfDllUnloadCallback(void *arg1);
void *memchr(const void *arg1, int arg2, unsigned long long arg3);
char *strchr(const char *arg1, int arg2);
unsigned long long strcspn(const char *arg1, const char *arg2);
unsigned long long strnlen(const char *arg1, unsigned long long arg2);
char *strpbrk(const char *arg1, const char *arg2);
char *strrchr(const char *arg1, int arg2);
unsigned long long strspn(const char *arg1, const char *arg2);
char *strstr(const char *arg1, const char *arg2);
int tolower(int arg1);
int WinSqmCommonDatapointSetDWORD64(
    unsigned long int arg1, unsigned long long arg2, unsigned long int arg3);
int WinSqmCommonDatapointSetString(
    unsigned long int arg1, const unsigned short int *arg2, unsigned long int arg3);
int WinSqmGetInstrumentationProperty(
    const unsigned short int *arg1, const unsigned short int *arg2,
    unsigned short int *arg3, unsigned long int *arg4);
int WinSqmIsOptedInEx(unsigned long int arg1);
void AlpcGetCompletionListLastMessageInformation(
     void *arg1, unsigned long int *arg2, unsigned long int *arg3);
unsigned long int DbgPrompt(const char *arg1, char *arg2, unsigned long int arg3);
char RtlQueryProcessPlaceholderCompatibilityMode();
char RtlSetProcessPlaceholderCompatibilityMode(char arg1);
char RtlIsNonEmptyDirectoryReparsePointAllowed(unsigned long int arg1);
char RtlIsZeroMemory(void *arg1, unsigned long long arg2);
unsigned short int RtlLogStackBackTrace();
void *RtlLogStackTrace(unsigned long int arg1);
void RtlReleaseStackTrace(void *arg1);
```

Figure 5: Example Prototypes from `ntdll.dll`

TELETYPES®

IMMEDIATE DELIVERY

MODEL 40 300 LPM PRINTERS



- Mechanism or complete assembly
- 80-column friction feed
- 80-column tractor feed
- 132-column tractor feed

INTERFACES

- EIA-RS232
- Simplified EIA-like interface
- Standard serial interface
- Parallel device interface

MODEL 43 TERMINALS

- 4310 RO (Receive Only)
- 4320 KSR (Keyboard Send-Receive)
- 4340 BSR (Buffered Send-Receive)

INTERFACES

- TTL Serial
- EIA RS232 or DC20 to 60ma
- 103-type built-in modem

FEDERAL Communications Corporation

11126 Shady Trail, Dallas, Texas 75229, (214) 620-0644,
TELEX 732211 TWX 910-860-5529

22:07 Timecryption, OTP with Near-polyglots

by Ange Albertini and Stefan Kölbl

Our foundation for this is the CounTeR (CTR) block cipher mode, which effectively turns a block cipher into a stream cipher. From a Nonce and a Key, it generates a keystream. The plaintext is then xored with this keystream to obtain the ciphertext. This mode acts as a one-time pad. Just an xor against a keystream, so encryption and decryption are the same operation. The cipher's decryption operation itself isn't used. If we decrypt with a different key, we end up xoring with a different keystream.

STUDEBAKER STUDEBAKER
STUDEBAKER
Yes, Studebaker has been in business for 112 years, with quality craftsmanship, & has & will continue to make quality cars in our new modern plant in Canada. This means better service to our customers, PLUS better buys!
You can now purchase a '64 COMMANDER 2 door equipped with radio, heater, undercoat, padded dash, dual brakes, cigar lighter & 15 inch wheels for
ONLY $1895 Delivered
PLUS Daytonas - Hardtops - Convertibles - Sliding Roof Wagons - Solid Roof Station Wagons - Cruisers - Commanders or Challengers 2 and 4 door Sedans or Wagons.
ALL AT NEW LOW PRICES
At
HILL Motors Inc.
30 BROAD ST., MANASQUAN
OPEN 'TIL 7 p.m. Phone 223-1708

What about crafting an ambiguous ciphertext? We define this as a ciphertext that gives meaningful plaintexts for different keystreams!

To do this, recall that we can freely modify the ciphertext: the keystream is set by *(Nonce, Key)*, and plaintext and ciphertext aren't involved, which means that for a given keystream if we change ciphertext bytes, we set the plaintext bytes, as it's simply an xor against a keystream.

So, we can directly create such a ciphertext with a binary polyglot whose interpretation varies by the eky. We just independently encrypt the different ranges of the file with the different keys, then combine the two ciphertexts at the right offsets.

Making Decryption Relative to Time

But we run into a key question: how do we have an uncooperative system decrypt to two different results? We postulate that in real-world applications, specifically those having key rotation, we can do this leveraging time.

If we know the key rotation scheme used by a system, we can craft a file that, when encrypted with the current key, might be authentically decrypted later with a different key added to the key ring. (Typically, newest keys are tried first, and decrypted plaintext is returned as soon as the decryption is authenticated.) So the file will be transparently decrypted to something else, something that you decided in advance:

Timecryption combines what you want now with what you want later. You control both. When implemented against a known key rotation scheme, it's transparent and works as intended.

Near-Polyglots

Typically, each ciphertext byte belongs to one payload and one only. But if we leverage two keys from the key rotation scheme—K_1 for now and K_2 for later—we can bruteforce a nonce that will get some bytes decrypted to two different sets of values.

This means that we can make two formats that will coexist in the same file starting at offset zero, such as PDF/PE or JPG/PNG, or the same format twice, where JPG/JPG would be a near ambiguous file.

There are two ways we identify to handle these pairs of files with the same format. One way to do this is with a technique such as causing a different comment length, a bit like a hash collision for JPG/JPG. In this case, it's a file with one header and two contents. Another way is to do so for formats that work from any offset such as HTML/HTML. In this case, it's two contents coexisting in the same file.

Note that the smaller number of bytes in the overlap of the two formats, the faster the nonce bruteforce will be! The overlap only needs to be as long as is required given the specific formats. For example, *ICC* requires any parasite to start at offset 0x132, which is impractical to bruteforce. This technique can be exploited quickly with formats like JPG since it has a very small minimal offset of 4.

The Mitra repository has all the tooling for CTR, OFB and GCM modes with precomputed examples.[26]

With Authenticated Encryption

In the case of CTR encryption, it's possible to change keys because the encryption is unauthenticated, a known security risk. For this reason, the Galois/Counter Mode (GCM) was created, which is just CTR with authentication via an extra authentication data and tag. However, it's possible to forge one of the blocks such that decryption will be valid for several keys, so GCM is vulnerable too.

Secondly, more complex modes are exploitable too, such as OCB3 and GCM-SIV.[27] These cipher modes work at the block level and not at the byte level, so you need to align payloads to the block boundary. They also require more than one block to compute the authentication collision, but that's a small overhead.[28]

It's even possible to set an arbitrary content in the authentication tag!

Authenticated encryption isn't enough if the key isn't committed to the encryption. It's possible to craft ciphertexts that authentically decrypt with different keys, which is something that multiple schemes were independently found vulnerable to.[29]

Conclusion

Near-polyglots are the starting point for funky polyglot-like with cryptography, whether for AngeCryption (ECB, CBC, CFB and OFB) or Timecryption (CTR, OFB, GCM, OCB3 and GCM-SIV).

Mixing near-polyglots (CTR, OFB) and forging contents to get the same authentication tag is possible for GCM, OCB3 and GCM-SIV mode.[30]

Mitra's handling of near-polyglots makes it very easy to merge dozens of different file formats, and the *key commitments* tools forge the tags. Using these techniques and tools, exploiting authenticated collisions only requires a few command line invocations!

[26] `git clone https://github.com/corkami/mitra`
[27] `git clone https://github.com/kste/keycommitment`
[28] Note that GCM-SIV's computation cost is relative to payload size, so try it with smaller files first!
[29] `unzip pocorgtfo22.pdf project_MircoStauble.pdf` % "Actually Good Encryption? Confusing Users by Changing Nonces" by Mirco Stäuble
[30] `unzip pocorgtfo22.pdf withoutcommit.pdf`

22:08 The Электроника МК-51 is a Casio fx-2500

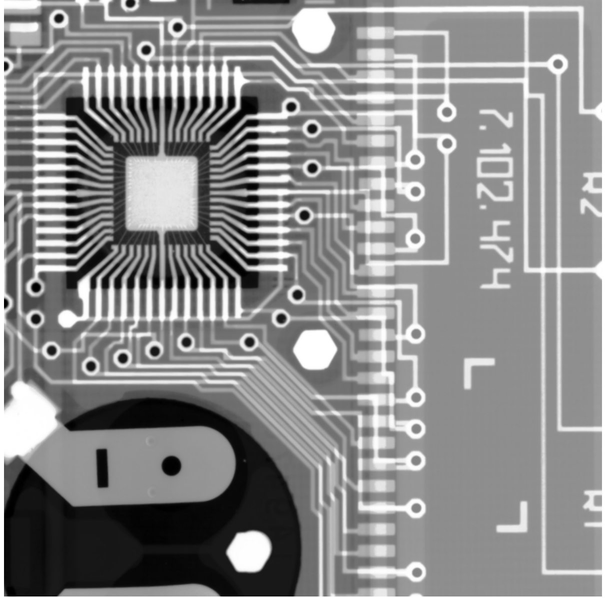
by Travis Goodspeed

Howdy y'all,

In the USSR, there was a calculator called the Электроника МК-51.[31] It looks an awful lot like a calculator from Japan, the Casio fx-2500. In this short article, I'll demonstrate that in addition to looking similar, the MK-51 is in fact a clone of the fx-2500, and that the Soviets went so far as to counterfeit the NEC microcontroller inside the fx-2500 with just minor alterations. Every last bit of the program ROMs are identical.

Let's begin with a little background. Introduced in 1978, the fx-2500 is a small scientific calculator that's held in a plastic wallet with its instruction manuals. The MK-51 joined it in 1982; manufactured until 2000, it also features a plastic wallet and instruction manuals. They both have an 8-digit LCD. The Casio is 121x67x11 mm, while the Elektronika is 130x71x8 mm. The keyboard layouts are a little different, some keys in different positions.

As I suffer from a disease late at night that involves alcohol and Ebay, it wasn't long before a few units of the Elektronika arrived from Ukraine. The Casio was surprisingly a little harder to find, but I found that one, too.



[31] Elektronika MK-51, if you don't follow Cyrillic.

I first tore down the MK-51. The zebra strip on this unit's LCD had long since turned to stone, but otherwise it seemed in decent condition for its age. The calculator is built around a single microcontroller in an epoxy blob package that rides within the plane of the PCB, a trick that I've also seen for reducing thickness on the HP-28 calculator. The blob in my calculator had no part number, both other sources online say that this chip is the К757ИП1-2. In X-ray, you can clearly see the hole cut out of the CPU to allow the CPU to fit with less thickness.

Going a little deeper, I desoldered the microcontroller, cut its legs off, and dropped it in a hot bath of HNO_3. The epoxy blob was torn off by nitric acid just fine, but it softened first in a way that Western QFP and DIP packages never do. I wonder what it's made of, but I'll leave that to professional reverse engineers of Soviet plastics rather than guess.

A full die photograph is shown in Figure 7. Nearly a third of the die is consumed by a diffusion ROM, and all pins are numbered, which was quite nice of the designers and convenient for reverse engineers.

Bits in a diffusion ROM are rarely surface visible, so I had to delayer this chip in dilute hydrofluoric acid. Ten minutes in hot Rust-Go did the job on the very first try. Bit rows are found in groups of four, with plenty of spacing between groups, but MaskROMTool – presented page 5 – was able to make short work of recognizing them.

One important trick with diffusion ROMs is that after delayering, they sometimes have no unique color, just a border line. So in addition to marking the center of each bit, I had to instruct my decoder to sample a wide stretch of pixels, recording the darkest color in each channel. This made the bits pop out, just a few dozen decoding errors.

The ROM itself is 352 columns wide and 64 rows tall, holding 22,528 bits or a little more than two kilobytes in total. While some very clever souls have decoded ROMs without knowing the architecture and instruction set, I wasn't very hopeful of doing the same. Who the hell knows what 4-bit microcontrollers were Ivan's favorite in the eighties?

So by this stage, I had die photographs and an export of the physically ordered ROM bits, but not

MY CASIO ALMOST GOT ME BUSTED AT THE BORDER.

When the customs agent at the border pulled the Casio M-811 out of my suitcase, he thought I was some kind of secret agent.

Because the Casio M-811 is the smallest calculator in the world, and looks like something out of the spy movies.

Wafer-thin, the size of a matchbook cover, it adds, subtracts, multiplies, divides, does percentages, and has a memory for chain operations.

I couldn't make the customs agent believe it wasn't some kind of electronic bugging device. "It's just a great little calculator," I said. "Lots of people have them in the States. Only costs $29.95."

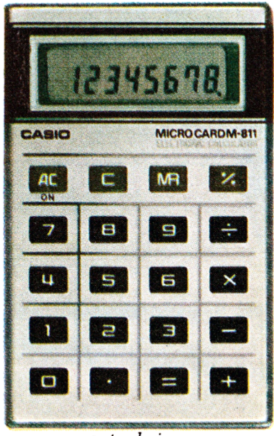
"You lie," he replied.

It took me an hour to convince him I wasn't Mata Hari.

Now, I'm not going to name the country involved (I don't want to start a war or anything) but if they're going to give everyone with an M-811 such a rough time, they're not going to have any tourist business left.

And besides, if I was a spy, I wouldn't be that obvious.

World's smallest calculator



actual size

AT CASIO®, MIRACLES NEVER CEASE.

Casio, Inc. Consumer Products Division, Executive Offices: 15 Gardner Road, Fairfield, N.J. 07006 New Jersey (201) 575-7400, Los Angeles (213) 923-4564.

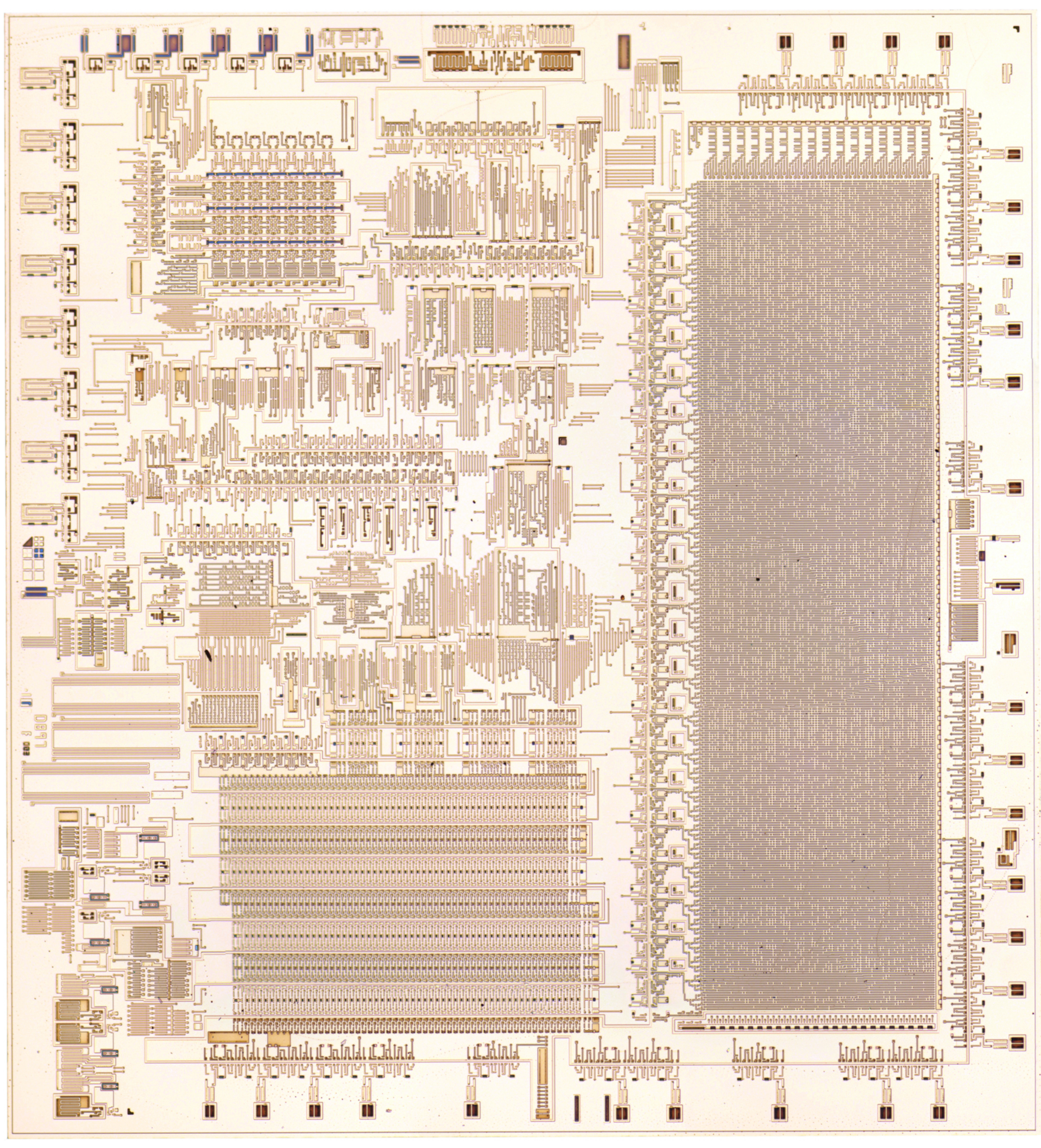


Figure 6: NEC D897G Die from the Casio fx-2500, Delayered

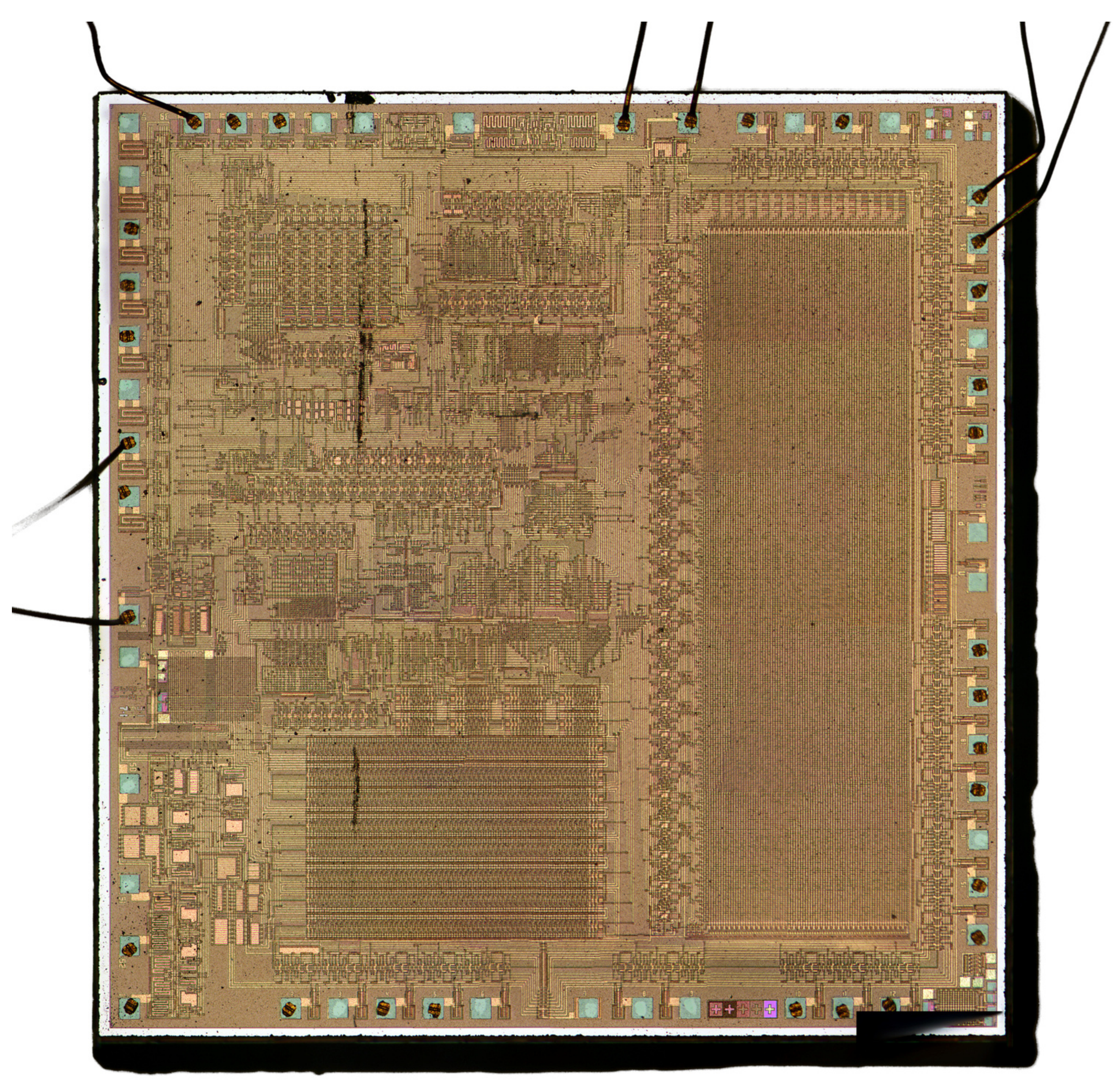


Figure 7: К757ИП1-2 from an Электроника МК-51, Top Metal

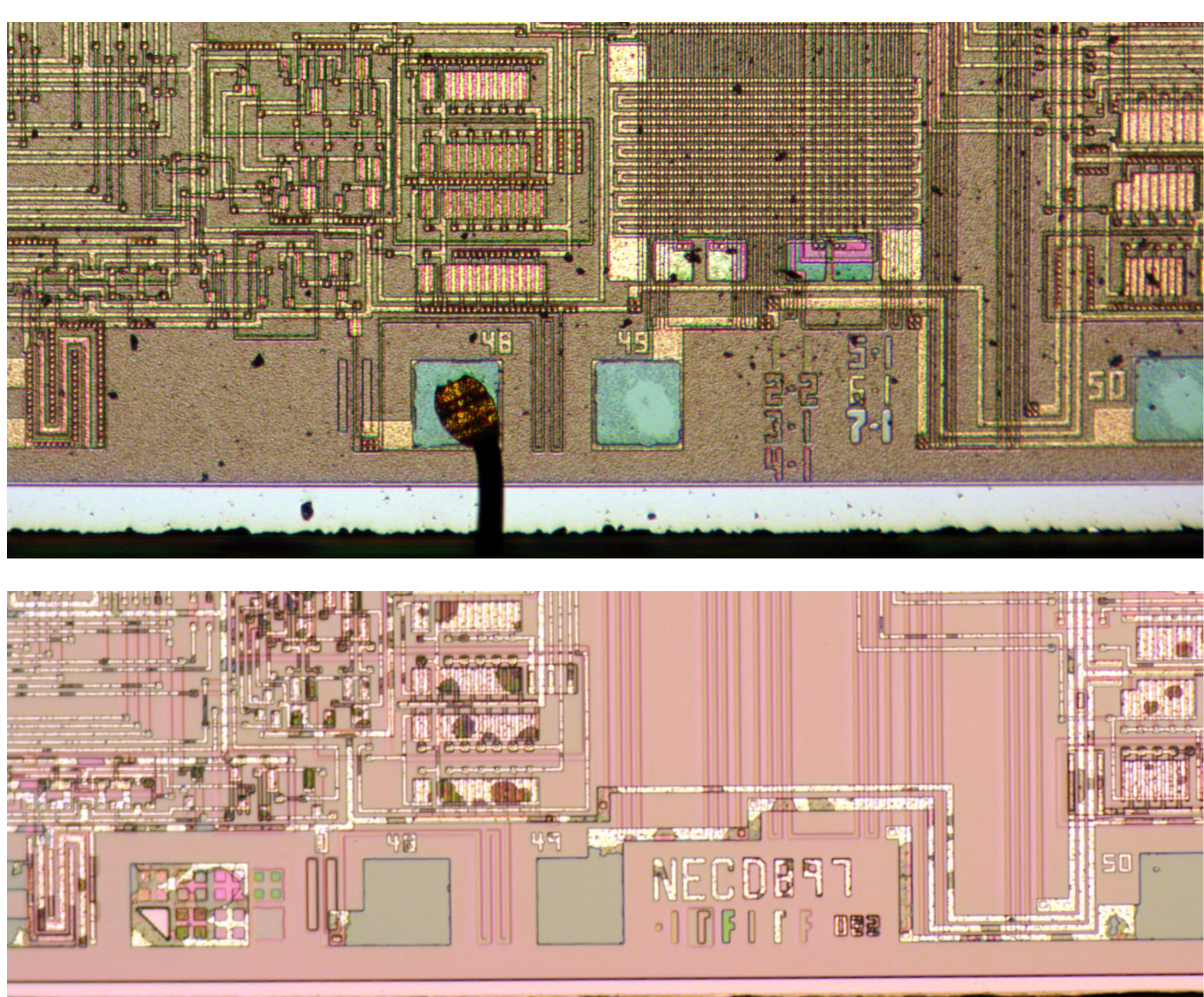


Figure 8: Mask Labels from Электроника МК-51 (top) and NEC D897G (bottom)

yet a decoding into logical bytes or error correction of the marked bits. To get further, our story leaves the Soviet Union and moves next door, to Japan.

My sample of the Casio fx-2500 arrived a week after the Elektronika. From the very first glance, it's clear that the exterior casing of one was modeled after the other. Minor differences in size and plastic quality were visible, and some of the button rows are in a different order.

Inside the case, the Casio also uses a single microcontroller, but this is attached to the PCB as a normal QFP package, rather than sitting in a cutout of the PCB. It has a visible part number of NEC D897G.

I decapped and then delayered it in the same way as I did the Soviet chip, then delayered it to get Figure 6. Both chips have pin numbers around the perimeter; both chips have layer labels between pins 49 and 50; and, both chips have a diffusion ROM that's roughly a third of the surface area.

The mask ROM of the NEC chip also contains groups of four rows, and it also contains exactly 22,528 bits in exactly 64 rows and 352 columns. The bits are identical, and I was able to correct the few dozen bit errors that I made in the Elektronika extraction by having MaskROMTool flag all differences with the Casio as errors. This shows that not only are these two calculators running the same architecture, but they have exactly the same firmware.

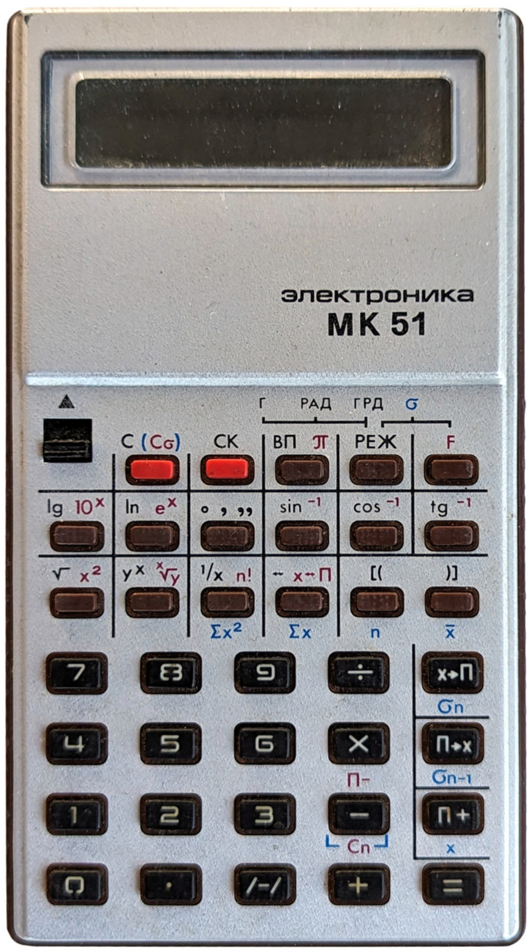
Unlike the Электроника calculator, which lacks a part number on the die, the NEC chip is labeled with D897G. This part number isn't in my databook collection, but the naming convention fits members of NEC's µCOM43 family that appeared in 1977, just a year before the calculator.

I grabbed Computer Gin and Fabulous Fred, µCOM43 chips which had been decoded by Sean Riddle. Between those and my handy NEC databook, I found that Fabulous Fred begins with `8f, 91`. `8f` is an `LDZ` instruction, loading `0f` into the data pointer. `91` is an `LI` instruction, loading `01` into the accumulator. Computer Gin begins with `15, 3e`. `15` is `ADC` instruction, and add with carry. `3e` is an `XI` instruction,

Plugging these instruction pairs into MaskRomTool's byte solver, it recommends a solution similar to the Riddle's ROMs.[32] With that decoding, the ROM begins to look something like µCOM43 machine language!

```
0:00: db      JCP    0x1B    Call a function
0:01: 15 5d   LDI    0x5D    Load the DP
0:03: 63      RPB    3       Clear I/O pins
0:04: 3d      XMI    1
0:05: 66      REB    2
0:06: 28      X              Move DP to A
0:07: 69      RMB    1       Clear bit of RAM
0:08: 01      DI             Disable Ints
```

Today I've not only shown that the Электроника MK-51 and the Casio fx-2500 not only look and behave similarly, but also that the MK-51 has a clone of the NEC microcontroller from the fx-2500, that the firmware is identical between the two calculators, and that the firmware is similar if not identical to that of the NEC µCOM43 architecture. With time this ROM dump might be built into an emulator, though we will have to figure out after which calculator to name the MAME module!



[32] `gatorom bits.txt -o rom.bin --decode-cols-downl-swap -i -r 90`

The Broadmoor 4-door Station Wagon—luxury interiors, powerful engine, rugged frame—Studebaker builds *both* beauty and utility into station wagons. The best ride, too, because only Studebaker station wagons have double-teamed springing—loaded or empty, they ride like fine sedans. *Craftsmanship* makes the big difference! Try the Broadmoor at your dealer's, *today!*

See Your Neighborhood Dealer Today

22:09 A Tourist's Guide to Reversing Renesas M16C and the R8C, too!

by Christopher Hewitt and Niccolò Izzo

Ehilà, vicino!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the architecture of smaller devices as quickly as possible, with a minimum of fuss and formality.

Those of you who have already worked with Renesas M16C or similar architectures might find it to be a useful refresher, while those of you new to the architecture will find that it really isn't as strange as you've been led to believe. If you've already reverse engineered binaries for any platform, even C-SKY CK803, you'll soon feel right at home.

We've written this guide using a device in the R8C/Tiny series for specific examples, but with minor differences it applies well enough to the R8C and M16C families as a whole. For larger Renesas parts, such as those used in engine control units and portable amateur radios, you might be better served by a different introduction. Either way, be sure to keep reading for a case study on applying power analysis and fault injection techniques to successfully recover firmware from an R8C/Tiny target with protected flash memory.

Architecture

Von Neumann
16-bit words

Registers

R0-R3: Data Registers (R0 and R1 as split 8-bit halves.)
A0-A1: Address Registers (A0 and A1 as combined 32-bit A1A0 register.)
FB: Frame Base
PC: 20-bit Program Counter
INTB: Interrupt Table (available as split 4-bit INTBH and 16-bit INTBL registers.)
USP: 16-bit User Stack Pointer
ISP: 16-bit Interrupt Stack Pointer
SB: 16-bit Static Base
FLG: 11-bit Flag register

Instructions

89 instructions, where instruction encoding is variable width
Opcode is 8-bit for the most frequently used
Opcode is 16-bit for the others

Instruction Set Basics

The first generation of R8C devices appeared in 2003 and were marketed as a cost-reduced alternative to Mitsubishi's M16C family, following the formation of Renesas as a joint venture between the semiconductor operations of Mitsubishi and Hitachi. The R8C family features the same 16-bit CISC architecture as M16C with binary level compatibility and the internal data bus reduced to 8 bits. These families are also compatible at the assembly level with the M32C family.

The instruction set is composed of 89 discrete instructions with many common instructions only requiring a single clock cycle. Instruction encoding has variable length and the opcode (8-bit for the most frequently used and 16-bit for the others) is followed by source and destination operands specified through different addressing modes.

Instruction mnemonics can be suffixed to prioritize one of the following four possible instruction formats with the assembler choosing an optimal format if one is not explicitly specified.

Generic (:G) Op-code (2 bytes), source (0-3 bytes), destination (0-3 bytes)

Quick (:Q) Op-code with immediate data (2 bytes), destination (0-2 bytes)

Short (:S) Op-code (1 byte), source (0-2 bytes), destination (0-2 bytes)

Zero (:Z) Op-code (1 byte), destination (0-2 bytes)

There are also numerous addressing modes categorized across three different types.

General Instruction Addressing Immediate, register direct, absolute, address register indirect, address register relative, SB relative, FB relative, SP relative

Special Instruction Addressing 20-bit absolute, address register relative with 20-bit displacement, 32-bit address register indirect, 32-bit register direct, control register (PC, INTB, USP, ISP, FLG) direct, PC relative

Bit Instruction Addressing Register direct, absolute, address register indirect, address register relative, SB relative, FB relative, FLG direct

Registers and Calling Convention

Be aware of different calling conventions depending on the compiler and options used. For example, the IAR C and C++ compiler for R8C and M16C supports a "normal" calling convention and a "simple" one. The normal (and default) calling convention is optimized to use registers as much as possible (with A0, R0, and R2 used as scratch registers), then defers to the stack for passing parameters. The simple calling convention, however, only passes the first parameter through R0L, R0, or R2R0 (depending on size), then defers to the stack for remaining parameters. R0 and R2R0 are also used for returning values from a function.

There are also subtle differences between Renesas' own compilers such as the NC30 compiler used for the M16C and R8C families, and the NC308 compiler used for M32C and certain M16C family parts. For example, NC30 preserves registers during function calls on the caller side, while NC308 does so on the called side.[33]

Regardless of the compiler used, stack frame manipulation is evident by the presence of `ENTER` and `EXITD` instructions to build and deallocate stack frames respectively.

Memory Map

Note that different documents have conflicts. We used a Chinese language datasheet for our figure on page 41 where things differed.[34]

Also note that this article's PoC dumps actual code from a region that isn't supposed to be valid for this specific part number, but is valid for different catalog part numbers (likely sharing the same die).

Editors Note: We have included a die photo on page 45 taken from processing a R5F21194, for anyone who wishes to perform future comparisons to other catalog part numbers.

Code Protection

The Renesas R8C/Tiny series supports a couple of different mechanisms for flash protection. Serial programmer commands to access the flash, including erasing, are completely ignored if a custom 7-byte ID code was interleaved with entries in the interrupt vector table at offsets `0x0FFDF`, `0x0FFE3`, `0x0FFEB`, `0x0FFEF`, `0x0FFF3`, `0x0FFF7`, and `0x0FFFB`. An ID code consisting of all-ones (such as when flash cells are unprogrammed from the factory) is automatically unlocked by the boot ROM, while any other value requires manual unlocking with a successful ID code comparison to re-enable flash manipulation. Parallel programmer commands to access the flash are ignored through configuration of the Option Function Select (OFS) register located at offset `0x0FFFF` by setting `ROMCP1=0` and `ROMCR=1`.[35]

Fixed Interrupt Vector Table (Flash)

Address	Vector	ID	Offset
`0x0FFDC`	Undefined Instruction	ID1	(`0x0FFDF`)
`0x0FFE0`	Overflow	ID2	(`0x0FFE3`)
`0x0FFE4`	`BRK` Instruction		
`0x0FFE8`	Address Match	ID3	(`0x0FFEB`)
`0x0FFEC`	Single Step	ID4	(`0x0FFEF`)
`0x0FFF0`	Osc. stop, watchdog, VM2	ID5	(`0x0FFF3`)
`0x0FFF4`	Address Break	ID6	(`0x0FFF7`)
`0x0FFF8`	Reserved	ID7	(`0x0FFFB`)
`0x0FFFC`	Reset	OFS	(`0x0FFFF`)

Finding a Target

Renesas is one of the leading suppliers of microcontrollers in the world but it's not very common to see their microcontrollers used by electronics hobbyists in western countries. Mass-produced commercial designs spanning from inexpensive toys to fault-tolerant automotive engine control units are much more likely to include Renesas parts.[36]

One low-cost and readily accessible product containing an R8C/Tiny microcontroller is the SA868 radio module with integrated power amplifier. Limitations in the module's factory firmware make it an attractive target for modifications, but this first requires unlocking access to the protected contents.

[33] The documentation is confusing here, for further see `unzip pocorgtfo22.pdf m32c90-compiler.pdf` Page M-70 and `unzip pocorgtfo22.pdf m32-compiler.pdf` Page 59.

[34] `unzip pocorgtfo22.pdf r5r0c00cn.pdf`

[35] See `unzip pocorgtfo22.pdf r8c-hardware.pdf` section "ROM Code Protect Function" (Page 250)

[36] See *Bypassing the Renesas RH850/P1M-E read protection using fault injection* by Willem Melching.

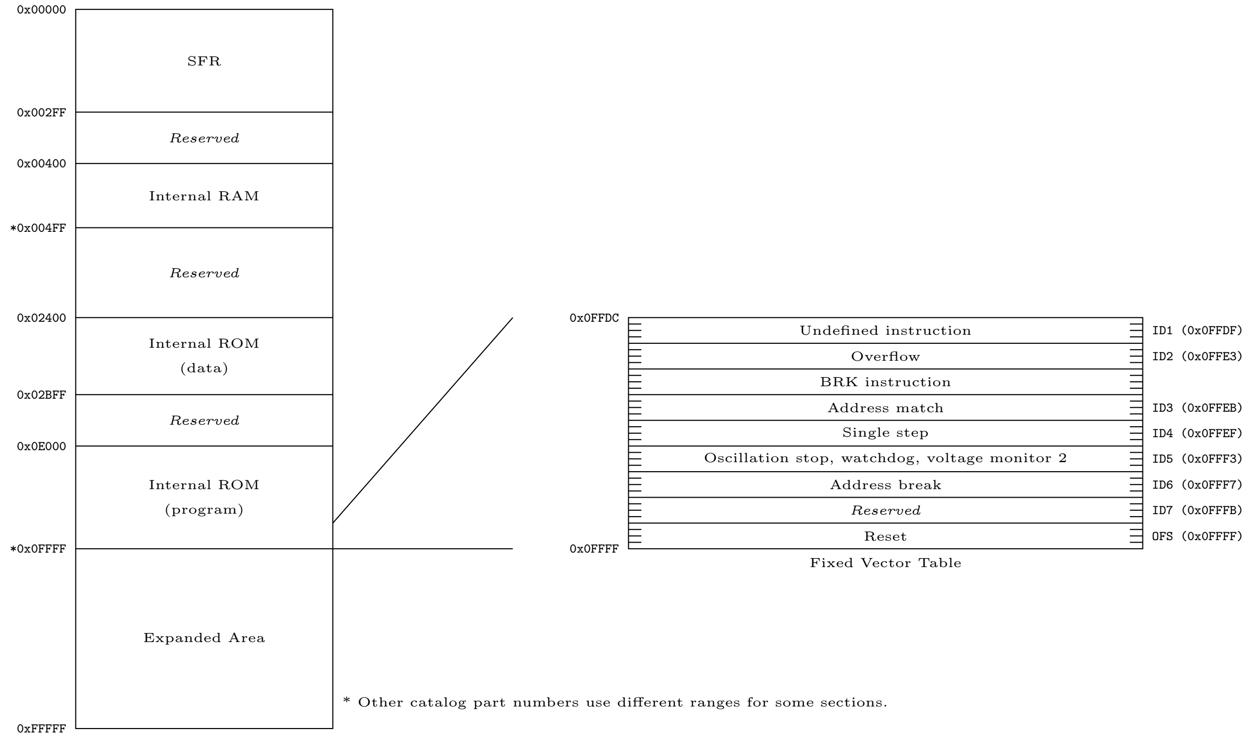
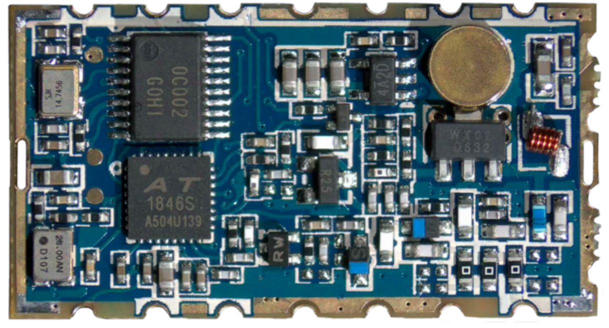
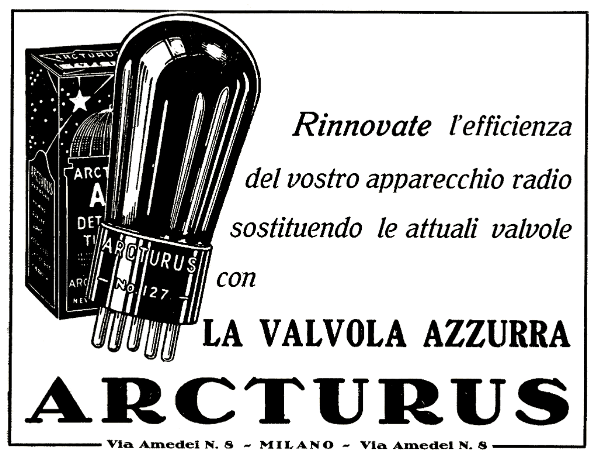


Figure 9: Memory Map Summary



Under the SA868 v1.1's metal shield sits a Renesas R5R0C002SN, an R8C/1B group compatible part that was only available to customers in Asia. A close approximation with a publicly available English language datasheet is the Renesas R5F211B2SP. The role of this microcontroller in the module is to expose a Hayes-style command set interface to control an Auctus AT1846S RF transceiver, which is the same part at the core of many low-cost amateur handheld radios including the ubiquitous UV-5R, GD-77, and MD-UV380.

Without getting too deep into radio theory, the SA868 module has a lot more potential than it was designed for. While the module is strictly marketed for use in analog FM applications, the transceiver component is used in more sophisticated digital radios using 4-FSK modulation. Once the microcontroller's protected flash can be unlocked, it is possible to dump and patch the firmware or even replace it with a custom purpose-built one to support more useful and interesting digital voice and data protocols.[37]

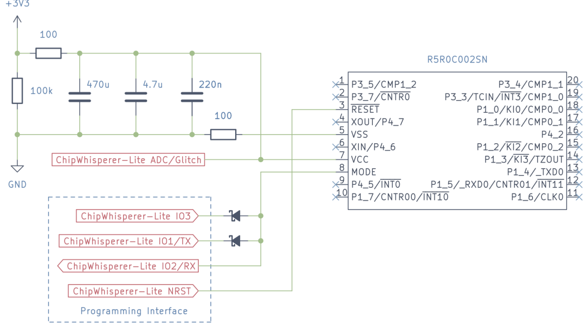


[37] See Delorie 2009 page 5, `unzip pocorgtfo22.pdf renesasflash.pdf`

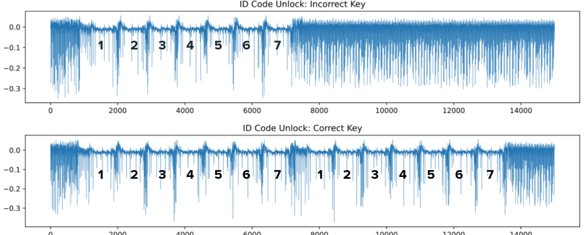
Extracting the Application

A previously published attack for a microcontroller in the M16C family described a successful timing attack against the boot ROM.[38] The authors demonstrated a measurable time delta between the last cycle of the serial programming clock (SCK) and an output pin (BUSY) asserted while servicing programmer commands that could be used to iteratively determine individual bytes of an ID code. This approach might have been viable for the R5R0C002SN since the R8C family is a close relative of the M16C family, if not for the lack of an equivalent pin indicating busy state. It is, however, still possible to demonstrate whether or not the timing attack is portable to this target by extracting the same information through power analysis of the ID code verification process.

A relationship to power consumption can be measured by removing the microcontroller from circuit and inserting a low value shunt resistor inline with the power supply. Experimentation with added capacitance or changing shunt resistor position helps establish which conditions provide the cleanest measurements. During any unsuccessful unlock attempt, voltage measurements at the supply pin expose seven evenly spaced segments, corresponding to each byte of the ID code. This observation suggests that the R5R0C002SN's boot ROM executes comparisons in constant time and is not vulnerable to the same timing attack. Brute force attempts are also discouraged by silently ignoring unlock requests after a few unsuccessful attempts. On a target with a known ID code, leakage from successful unlock attempts suggests that valid comparisons are performed twice, possibly to mitigate against power glitches.



Readers with experience in side channel analysis might be tempted to calculate Pearson correlation coefficients in order to match ID code attempts with power trace data in hopes of leaking bits or bytes from the real ID code, but the approach seemingly does not work here. Whether the result of high clock jitter, inadequate ADC resolution, or just bad luck, no apparent correlation between ID code attempts and resulting power trace can easily be identified.



Fault injection is another tool at our disposal for extracting protected flash memory contents. Long pioneered by satellite television enthusiasts exploring conditional-access modules, fault injection attacks traditionally manipulate clocks or supply voltages as a mechanism for introducing unintentional behavior to a system, such as causing instructions to be skipped or register contents to be modified. Devices like the ChipWhisperer-Lite have made these kinds of practical attacks significantly more approachable for hobbyists, but don't disregard the price point and flexibility of a simple microcontroller combined with a fast switching MOSFET to momentarily bridge a supply voltage to ground potential.

Each microcontroller in the R8C/Tiny family has an integrated ring oscillator running at roughly 8 MHz further divided by 32 for use as the system clock during boot ROM execution. This clock is not exposed externally, so clock glitching is not the most convenient approach. The high degree of jitter from this internal clock combined with the double check of the ID code observed in power analysis means that landing a voltage glitch twice during an unlock attempt, with the correct timing, might be excessively difficult. But let's consider for a moment if a successful ID code verification is even necessary prior to accessing flash programming commands. Maybe it's really only a formality intended for diligent engineers who rigidly follow the rules outlined in the hardware manual.

It is clear from the location of the fixed interrupt vector table that the ID code is stored on the last page of flash and clear from the programming guide that there is a flash page read command in the boot ROM. It's not unreasonable to at least try

[38] See *Hacking Toshiba Laptops* by Serge Bazanski and MichałKowalczyk.

repeatedly reading the final page of flash without any prior ID code verification while simultaneously sweeping glitches over the microcontroller's power supply with varied time offsets. The programming interface's serial transmit pin can even be used as a trigger to anchor glitches around the page read commands.

Some experimentation is required to find glitch pulse lengths and time offsets that don't stall the microcontroller yet still influence boot ROM behavior. Keep in mind that variations in capacitance and even temperature can easily impact results and repeatability. Since thousands of glitch attempts might be required for a single success, it's best to keep each attempt as short as possible: Skip unnecessary communication steps by directly using the boot ROM's flash programming protocol and only perform hard resets when the microcontroller is completely stalled and not responding. With a little luck, our trusty microcontroller confidently returns a full page of flash data, rather than nothing.[39]

Glitching an unknown programmed R5R0C002SN sample from AliExpress

```
[*] bootrom: VER.1.20
[*] injecting faults...
<omitted>
[?] dumped page - width: 37.890625 offset: -44.921875 ext_offset: 5420
0000FF00   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FF10   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FF20   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FF30   ff ff ff ff  53 fb 00 00  53 fb 00 00  53 fb 00 00  ....S...S...S...
0000FF40   53 fb 00 00  53 fb 00 00  b4 fa 00 00  53 fb 00 00  S...S.......S...
0000FF50   53 fb 00 00  53 fb 00 00  aa f8 00 00  53 fb 00 00  S...S.......S...
0000FF60   53 fb 00 00  53 fb 00 00  53 fb 00 00  53 fb 00 00  S...S...S...S...
0000FF70   53 fb 00 00  1b fb 00 00  ff ff ff ff  ff ff ff ff  S...............
0000FF80   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FF90   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FFA0   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FFB0   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FFC0   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FFD0   ff ff ff ff  ff ff ff ff  ff ff ff ff  53 fb 00 4e  ............S..N
0000FFE0   53 fb 00 6a  53 fb 00 00  53 fb 00 46  53 fb 00 74  S..jS...S..FS..t
0000FFF0   50 fb 00 53  53 fb 00 59  53 fb 00 54  49 e0 00 f7  P..SS..YS..TI...
[!] valid idcode - 4e6a4674535954
[*] dumping entire flash...
[*] block 0 (0x0e000 - 0x0ffff):
0000E000   7b 60 5e 7c  65 3d 3f 70  7f 7d 77 2f  1b 6e 1f 17  {`^|e=?p.}w/.n..
0000E010   f4 85 0f f4  92 0f f4 ba  0f f4 d5 0f  f4 e1 0f f4  ................
0000E020   02 10 f4 ec  0f f4 0f 10  f4 2a 10 f4  56 10 f4 43  .........*..V..C
<omitted>
[*] block 1 (0x0c000 - 0x0dfff):
0000C000   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000C010   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000C020   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
<omitted>
[*] block a (0x02400 - 0x027ff):
00002400   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
00002410   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
00002420   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
<omitted>
[*] block b (0x02800 - 0x02bff):
00002800   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
00002810   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
00002820   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
<omitted>
[*] done
```

[39] See the Jupyter notebook `unzip pocorgtfo22.pdf r8c-glitch.ipynb`

R8C/Tiny BootROM ID Code Bypass - Greetz to ShmooCon Moose Crew

RENESAS

Successful glitches don't always return meaningful data, but ID codes can be assembled from their expected offsets in the page, then verified through an unlock attempt. Eventually you'll find a match and, once successfully unlocked, the entire flash memory can be dumped or erased and reprogrammed. If it looks like some data is missing, try reading additional flash pages that aren't officially supported by the part number since there's a good chance the same internal die is used by several part numbers.[40]

Proceeding with Analysis

Once a firmware image is safely recovered, you'll almost certainly want to inspect how it works. M16C isn't as esoteric as it might seem and there are actually a few different options for analysis. IDA Pro provides a disassembler for the architecture and Binary Ninja has some support by way of a third-party plugin.[41] If you're averse to commercial software don't forget about GNU Binutils which supports M16C and R8C through the m32c-elf target.

```
;; Recovered firmware through m32-elf-
    objdump
0000fcca <.data+0xd8ca>:
  fcca: eb 40 32 06   ldc #1586,isp
  fcce: c7 02 0a 00   mov.b:s #2,0xa
  fcd2: b7 04 00      mov.b:z #0,0x4
  fcd5: b7 0a 00      mov.b:z #0,0xa
  fcd8: eb 30 80 00   ldc #128,flg
  fcdc: eb 50 b2 05   ldc #1458,sp
  fce0: eb 60 00 04   ldc #1024,sb
  fce4: eb 20 00 00   ldc #0,intbh
  fce8: eb 10 dc fe   ldc #-292,intbl
  fcec: fd 64 fc 00   jsr.a 0xfc64
  fcf0: 75 cf ba 04   mov.w:g #1586,0x4ba
  fcf4: 32 06
  fcf6: 75 cf bc 04   mov.w:g #128,0x4bc
  fcfa: 80 00
  fcfc: d9 0f be 04   mov.w:q #0,0x4be
  fd00: fd a2 fa 00   jsr.a 0xfaa2
  fd04: eb 70 00 00   ldc #0,fb
  fd08: fd 7a f5 00   jsr.a 0xf57a
  fd0c: f5 03 00      jsr.w 0xfd10
  fd0f: fb            reit
  fd10: d9 10         mov.w:q #1,r0
  fd12: 6e fd         jne 0xfd10
  fd14: f3            rts
```

Alternatively, a Ghidra third party plugin created recently is capable of disassembling most instructions and may help jumpstart new reverse engineering projects through integration with Ghidra's processor independent decompiler.[42]

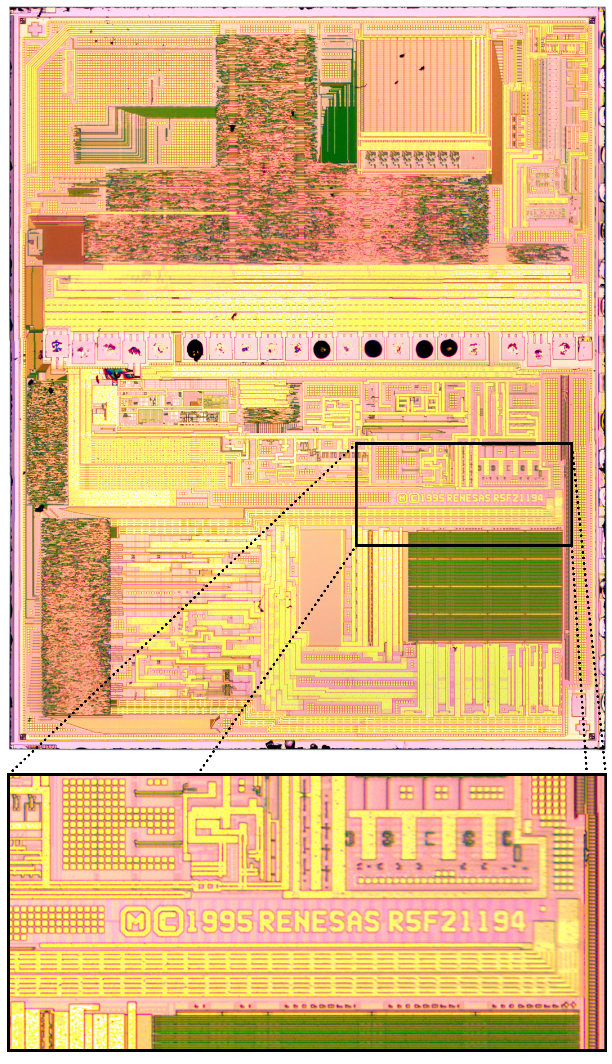
```
;---------------------------------
; after reset, this program will start
;---------------------------------
ldc     #((topof istack)+(sizeof istack)),
    isp ;set istack pointer
mov.b   #02h,0ah
mov.b   #00h,04h    ;set processor mode
mov.b   #00h,0ah
.if __STACKSIZE__ != 0
  ldc     #0080h,flg
  ; set stack pointer
  ldc     #((topof stack)+(sizeof stack)),sp
.else
  ldc     #0000h,flg
.endif
  ldc     #__SB__,sb  ;set sb register

  ; If the destination is INTBL or INTBH,
  ; make sure that bytes are sent in order
  ldc     #((topof vector)>>16)&0FFFFh,INTBH
  ldc     #(topof vector)&0FFFFh,INTBL

<omitted>

;===================================
; Call main() function
;-----------------------------------
ldc     #0h,fb; for debugger

; Remove the comment when you use
; global class object
; Sections C$INIT will be generated
;  .glb      __CALL_INIT
;  .call     __CALL_INIT,G
;  jsr.a     __CALL_INIT

.glb     _main
.call      _main,G
jsr.a      _main
```

Whichever option you pick, be sure to identify the correct entrypoint for the binary by referencing the reset vector in the fixed interrupt vector table at the very end of flash memory.

[40] See *The Secret of R8C/M11A and M12A* at the RVF/RC45 blog.

[41] `git clone https://github.com/whitequark/binja-m16c`

[42] `git clone https://github.com/silverchris/m16c`



Die photograph by Travis Goodspeed

An Unexpected Outcome

News of the effort to repurpose the SA868 with custom firmware eventually found its way to the company producing the radio modules, NiceRF Wireless Technology. An amateur radio enthusiast in China, Amo Xu, made a compelling case for the company to release an intentionally user-programmable variant of the module. Shortly after their discussion, the company began offering the SA868S Open Edition module. This variant is erased after quality assurance, guaranteeing the module is unlocked for reprogramming.

The new SA868S, version 2.0, is notably different from previous versions in that the microcontroller has been replaced with one from a different Renesas microcontroller architecture family, RL78, which is not vulnerable to the attack presented in this article. The RL78 family has, however, been explored in some detail in the context of the PlayStation 4 gaming console and several useful tools already exist for working with that platform, including an implementation of the debugging protocol and third party plugins for IDA Pro and Ghidra.[43]

While not as common as it should be, hardware reverse engineering occasionally leads to mutually beneficial outcomes with a manufacturer. A deep dive into an unfamiliar microcontroller architecture to improve a product's capabilities led to a manufacturer removing obstacles for experimentation. The availability of the SA868S Open Edition paves the way for user customizable firmware and has already motivated the creation of a free and open source alternative firmware granting complete control of all registers in the underlying transceiver chipset, enabling use of digital protocols such as APRS and M17.[44]

We hope that you've enjoyed this little guide to Renesas M16C and R8C, and that you'll keep it handy when reverse engineering firmware from those platforms.

[43] See *PS4 Aux Hax 2: Syscon* at Fail0verflow.

[44] `git clone https://github.com/OpenRTX/sa8x8-fw.git`

22:10 A Tourist's Guide to Эльбрус

by evm

In the tradition of the many high quality tourist guides that have appeared in this fine publication, let's take a magical tour around Russia's modern computer architecture, the Elbrus 2000.[45] [46] [47]

At A Glance

Common Models
Elbrus-1S+, Elbrus-4S, Elbrus-8S, Elbrus-8SV, Elbrus-16S

Architecture
Von Neumann
Very Long Instruction Word
Register Windowing (32-bit Base Registers)

Registers
`g0`–`g31`: Global Registers
`r0`–`r17`: General Purpose (Windowed)
`b0`–`b7`: Overlay Register within Window
Pred0-Pred31: Boolean Predicate Registers

Address Space
64-bit Virtual Addressing
Unknown Physical Memory Map

Background & History

Elbrus is a Russian CPU architecture that has been around in some form for over 40 years. It started at Lebedev Institute of Precision Mechanics and Computer Engineering. It was the first superscalar, out-of-order execution processor developed in the Soviet Union (when the Elbrus 1 debuted in 1979). The architecture was extended to be a very long instruction word (VLIW) architecture with Elbrus 3 in 1990. Once fully integrated as a microprocessor architecture in 2001 (previous versions had used many discrete chips), the architecture became known as Elbrus 2000, or E2K for short. Elbrus is designed in Russia but currently manufactured by TSMC in Taiwan because of a lack of Russian production facilities capable of producing chips at advanced technology nodes.[48]

In the early '90s, the Lebedev Institute spun off a joint stock company called the Moscow Center of SPARC Technologies (now shortened to just MCST). MCST currently produces new Elbrus chips and Elbrus-based PCs, laptops, and servers. Elbrus-8S and 8SV are the current top-of-the-line processor models (eight core versions for servers and desktops), and a lower-cost 1S+ (single core) is available as well. Note the transliteration from Cyrillic where the model names appear as Эльбрус-8С, Эльбрус-8СВ, and Эльбрус-1С+, respectively. Anecdotally, the 8S CPUs are about three times slower than a comparable Intel CPU,[49] but the draw of Elbrus is that it's a fully domestically designed Russian processor. The Russian military has reportedly ordered thousands of ruggedized laptops based on the Elbrus-1S+,[50] although there is no indication that the order was ever delivered.

There is currently very little public documentation on Elbrus because MCST controls most documentation under nondisclosure agreements. This means we don't have full processor documentation like we normally would for a commercial CPU.

[45] Travis Goodspeed and Ryan Speers, "A Tourist's Phrasebook for Reversing Embedded ARM in the Dialect of the Cortex M Series," PoC||GTFO 11:6

[46] Ryan Speers and Travis Goodspeed, "A Tourist's Phrasebook for Reversing MSP430," PoC||GTFO 11:08

[47] Chris Hewitt, "A Tourist's Guide to Altera NIOS," PoC||GTFO 21:7

[48] Ian Cutress, "Russia's Elbrus 8CB Microarchitecture: 8-core VLIW on TSMC 28nm," AnandTech, June 1, 2020.

[49] Anton Shilov, "Russian-Made Elbrus CPUs Fail Trials, 'A Completely Unacceptable Platform'," Tom's Hardware, December 24, 2021.

[50] Inna Sidorkova, "Цены на военные ноутбуки достигли Эльбруса." July 9, 2018, RBC.

ENGINEERS AND SCIENTISTS

Phoenix, HM-99, Syncom, Colidar Laser Range Finders, ADO - 51, Surveyor, Maverick, Early Bird, TOW, Polaris Guidance Systems, AIM-47, Advanced Technology Satellites, VATE and more...

They're all at Hughes.

(Should you be there, too?)

Send your resume to: **Mr. Robert A. Martin**/Head of Employment/Hughes Aerospace Divisions/11940 West Jefferson Boulevard/Culver City 80, California.

Requirements: Accredited Engineering degree and a minimum of three years of applicable professional experience. U. S. Citizenship.

We promise you a reply within one week.

An equal opportunity employer.

HUGHES

HUGHES AIRCRAFT COMPANY
AEROSPACE DIVISIONS

We used three sources of information for this article: (1) a Russian guide to Elbrus programming and optimization published by MCST,[51] (2) source code published by the OpenE2K group (a hobbyist group seemingly unrelated to MCST), and (3) leaked Linux kernel source code.

MCST is currently on the US sanctions list but thanks to the Reverend and friends we had access to an Elbrus-1S+ machine and used it to play around with some code examples. Our Elbrus was running a version of Linux made by MCST, but other Russian Linux distros are also available for Elbrus (e.g., Astra Linux). The Elbrus machine has a compiler called lcc, which is the MCST compiler based on gcc. It produces standard Linux ELF binary files. The options for disassembly at the moment are limited to ldis, which is part of lcc, and objdump, which is part of the binutils package put out by the OpenE2K group. ldis produces cleaner output, including resolution of symbol names, while objdump has a debug flag in the build that will prefix the output with the decoded instructions in hex. Anecdotally, ldis seems to miss some things (e.g., not disassemble all functions), although that could be due to operator error.

In order to explore the Elbrus instruction set we updated rix's Smashing C++ VPTRs from Phrack 56:8. That is a whole story for a another day, but you will find my code examples and the corresponding Elbrus disassembly attached to this PDF.[52]

Basics of Instruction Set Decoding

The first thing we needed to figure out was how the instruction format works since the official documentation left this topic out entirely. Fortunately we found that the OpenE2K binutils release has a preprocessor flag `ENABLE_E2K_ENCODINGS`, which causes objdump to print out the instruction bytes and their groupings.[53] A version of objdump with this flag was what we used to produce the disassembly for most of this article.

In Elbrus documentation, the VLIW is called a "wide command" (широкой командой). A wide command contains multiple instructions, each of which is targeted at individual execution units in the CPU pipeline. The documentation variously uses the terms "commands" (команд), "instructions" (инструкций), and "operations" (операций) for the component instructions within the instruction word.

The OpenE2K objdump code refers to the way these component instructions are encoded as "syllables." A nice feature of Elbrus is that the instruction encoding is fairly simple when compared against modern DSP architectures we've experienced. Instruction counting is an exploitation task that can be pretty complicated on some architectures, but not Elbrus. It's fairly simple to determine the length of an instruction from the initial "HS" syllable (shown on page 49).

The HS syllable determines the presence of the other instruction syllables, which appear in a particular order. The order is: SS; ALU; CS0; ALES half syllables 2 and 5; CS1; ALES half syllables 0, 1, 3, and 4; AAS half syllables; a gap check; CDS; PLS; and finally LTS (literals). Literal syllables (i.e., immediate values) occur at the end of the syllables. The OpenE2K objdump code looks for all of the syllable presence flags above, reads them in order (minding the possible gap), and then compares the number of syllables read against the size field in HS. Any extra syllables are read as literals. For syllables that contain "half syllables" (i.e., 16-bit values), the order of the syllables is flipped as they appear sequentially in memory.

```
Byte order          | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
                    ---------------------------------
Half syllable order |   1   |   0   |   3   |   2   |
```

This makes more sense if you think about the bytes being read in as 4-byte little-endian values.

```
Word order          |       0       |       1       |
                    ---------------------------------
Byte order          | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 |
                    ---------------------------------
Half syllable order |   0   |   1   |   2   |   3   |
```

Register Set

Elbrus's basic registers consist of 18 general-purpose registers (`r0`–`r17`), 32 global registers (`g0`–`g31`), and a sliding set of windowed registers (`b0`–`b7`). More will be explained about the register windowing in the next section. Registers are prefixed with an access width, similar to x86.

For example, `sr0` is single (32-bit) `r0`, and `dr0` is double (64-bit) `r0`, which is also the default. When the registers are used with floating point values,

[51] `unzip pocorgtfo22.pdf elbrusprog.pdf`
Murad Neumann-zadeh and Sergei Korolev, "Руководство по эффективному программированию на платформе «Эльбрус»"

[52] `unzip pocorgtfo22.pdf vptrs.zip`

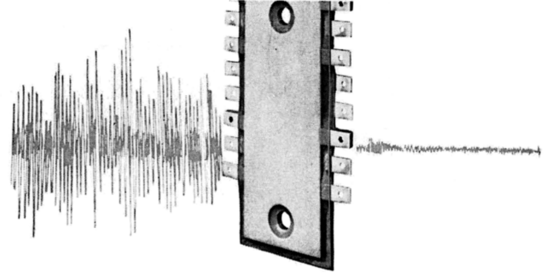
[53] `git clone https://git.mentality.rip/OpenE2K/binutils-gdb.git`

```
|31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
-------------------------------------------------------------------------------------------------
|       ALU         |        ALES        | PLS | CDS |  CS |  | C|  | L|   NOP  |    S   |      F    |
L = loop mode
NOP = number of cycles to delay (max 7)
S = size of instruction word (add 1 to get number of 8-byte fields in the instruction word)
C = presence of SS (control transfer) syllable
ALU = presence of ALU syllables (6 possible)
CS = presence of control syllables (call/return, windowing instructions) (CS0 = 14, CS1 = 15)
F = size of F1 fragment (add 1 to get number of 4-byte fields in the fragment)
ALES = presence of ALES syllables (5 possible)
CDS = presence of CDS syllables
PLS = presence of CLS syllables
```

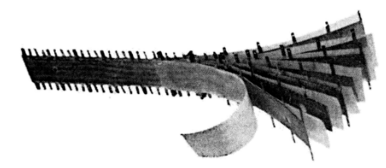
Encoding of the initial HS syllable, which determines the presence of other syllables, in an Elbrus-wide instruction word. It is unclear what the ALES, CDS, and PLS syllables are used for as we did not generate any of those instructions in the example code.

there is an `xr0`, which is an 80-bit version, presumably using the long double format from x86. According to the documentation, two double registers can be accessed as a quad register – for example `qr[i]` where `[i]` is even – but `gdb` on our box doesn't seem to be aware of this notation.

CAUTION: Elbrus has a word size of 32-bits for both registers and memory accesses, so the notion of single/double/quad on Elbrus is double what you might be used to on 64-bit x86, where the length of a word dates back to its early ancestors.



Laminated Bus Bars For Noise Reduction



Flat bus conductors laminated with Eldre's thin, rugged insulation will reduce electrical noises which cause havoc in high speed, solid state equipment. Lower the inductance and control the capacitance of your vital power distribution lines. Ground shields are interleaved with the voltage-carrying conductors so that effective shielding can be adequately provided. The terminations of each conductor, as shown, are for soldering but other types can be incorporated into the bus design. This compact and completely molded bus can replace a bulky harness and repetitive wiring.
Increase the reliability of your circuit with a bus system and obtain efficiency.

Basic Arithmetic, and Memory Operations in Elbrus

Here we show the various ALU register operations:

Integer Arithmetic Instructions	
add	Addition
sub	Subtraction
rsub	Reverse subtraction
umul/smul	(Un)signed integer multiplaction
udiv/sdiv	(Un)signed integer division
umod/smod	(Un)signed modulo
sxt	Sign Extend
Bitwise Operations	
and/andn	Boolean and/nand
or/orn	Boolean or/nor
xor/xorn	Boolean xor/xnor
shl/shr	Shift left/right
scl/scr	Shift cyclic
sar	Shift right arithmetic (signed)
insf/getf	Set/get bitfield
Floating Point Arithmetic Instructions	
fadd	Floating point addition
fsub	Floating point subtraction
frsub	Floating point rev. subtraction
fmax/fmin	Floating point maximum/minimum
fmul/fscale	FP mult. / mult. by power of 2
fdiv/frcp	Floating point division/reciprocal
fsqrt	Floating point square root

A basic ALU instruction looks like this:

```
ALS0  1181d48d  addd,0 %dr1, _f16s,_lts0hi 0xfff0, %dr13
```

This translates to "add double precision, using channel 0, the 64-bit register %dr1 to the signed 16-bit value 0xfff0, and place the result in dr13." There are six ALU channels, so you can do up to six ALU instructions in one wide instruction. There is no simple register "move," so the compiler tends to use a zero-add as a "move" instruction. The full list of

ALU register operations is shown in the table above. Notice that this is a fairly small number of operations. Outside of the VLIW construct, the Elbrus instruction set feels pretty RISC-like.

Memory operations are also pretty simple. Operations are load and store with a variety of width specifiers. Addresses can be a register plus an immediate offset, or the sum of two registers. Here is an example of a basic load operation:

```
ALS0  678dc08c  ldd,0 %dr13, 0x0, %dr12
```

ldis renders this (a bit more clearly) as:

```
    ldd,0 [ %dr13 + 0x0 ], %dr12
```

This translates as "load double word (64-bits) from memory, using channel 0, from the address dr13 + 0, and store in register dr12." There are also array memory load/store operations (ldaa/staa) that work similarly. As far as we can tell from the documentation, the array mode doesn't add any special addressing. It's still the sum of two registers or a register plus a constant; the main advantage is that there's a built-in post-increment operation.

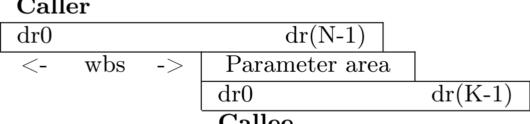
Register Windowing

Probably the simplest way to understand register windowing is that it functions similarly to local variables within a stack frame in a memory stack. In processors without windowing (which is nearly all processor families with some notable exceptions, like SPARC and Itanium), we are used to code transferring registers around between function calls, meaning that some registers need to be saved on the memory stack or transferred to nonclobbered registers (those guaranteed by the application-binary interface to not get modified by the called function) prior to a function call.

A function of reasonable complexity will save registers it's not supposed to clobber to the memory stack so that they are available for calculations and then restore the previous values from the stack at the end of the function. Register windowing aims to reduce some of this register bucket-brigading overhead by making the register set function more like a memory stack. In the same way that a function allocates a stack frame for itself, a function allocates a window of registers.

On Elbrus, this is accomplished in a function prologue with the setwd instruction. After setwd executes, the "register" r0 is really a reference to the first item in the register window. Now the function can use r0 to r<N> without having to save any registers from the calling function. How about parameter passing in registers? Just like architectures with stack-passed parameters, we need a calling function and the called function to share an overlapping area.

This is done with a wbs parameter in the call instruction. wbs indicates the start of shared function parameters within the current window. After a call, r0 in the called function now refers to the base of the shared parameter area. This is illustrated here, where a caller function has a window of size N and calls a function that allocates a window of size K:



Elbrus also offers a sliding or mobile base register (b), which a function can use within its own function window. The base register is just an overlay on the existing register window; it points to a given register within the window. Accesses to registers with the base register use an array notation—for example, db[0] means "access first double register (64-bits) at the base pointer." The instruction setbn is used to set this pointer.

The operand rbs (the offset to set b from the base of the window) is also specified in quadwords. In practice, it looks like lcc uses the base pointer to point to the parameter area, so db[0], db[1], db[2], etc. are parameter 0, parameter 1, parameter 2, etc. for functions that are about to be called.

Since functions return values in dr0, this also means that db[0] holds the return value from the calling function's point of view.

Calls and Branches in Elbrus

Calls and branches are somewhat unique in Elbrus, they occur in two phases instead of in a single instruction the way it works on most architectures. Elbrus uses the disp instruction to set up any kind of control transfer instruction. This sets the ctptr1 register to the target address. The call instruction executes the control transfer. This allows the pipeline to get a little bit of advance warning for the call, allowing it to set up state for the target function, which can be undone or ignored if the call doesn't execute. The documentation refers to the ipd portion as specifying the "swap depth," but it is unclear what this means.

Return instructions happen similarly with a `return` instruction first that sets up the return and the `ct` instruction to execute control transfer. (This is also used for branches, as we'll discuss in the next section.) Notice that the function never seems to do anything with the return address. This is because Elbrus has a completely separate call chain stack, called the Procedure Chain Stack (PCS). Architecturally this is referenced via the Procedure Chain Stack Pointer (PCSP) register. The PCSP is not accessible from user mode; rather, it is set up by the kernel similarly to how user stack memory gets set up on a per-process basis.

The Procedure Chain Stack (PCS)

The PCSP is pretty simple—it's a 128-bit register with 64-bit "lo" and "hi" parts. The "lo" part contains the base address, and the "hi" part contains an index to the current frame.[54] It is unclear at this point what the "rw" field is actually used for.

```
(gdb) info registers pcsp_lo
pcsp_lo     0x1800c2e00002b000 1729596524238909440
    base    0xc2e00002b000 214267328638976
    rw      0x3 3

(gdb) info registers pcsp_hi
pcsp_hi     0x200000000060 35184372088928
    ind     0x60 96
    size    0x2000 8192
```

The Linux kernel source code shows the format of the stack frames, in the form of the `e2k_mem_crstack` struct. Each frame is 32 bytes and consists of four saved 64-bit register values, the "lo" and "hi" parts for `cr0` and `cr1`, respectively. Again we are left without documentation on what exactly the `cr0` and `cr1` registers do, but they must be involved in control transfers. The Linux code shows that `cr0` "hi" contains the return address, and `cr1` contains a bunch of fields pertaining to the current procedure's register window.

Here is the definition of `e2k_mem_crstack` (PCSP frame structure) in the E2K Linux kernel (`arch/e2k/kernel/e2k_syswork.c`):

```
typedef struct e2k_mem_crstack {
  e2k_cr0_lo_t   cr0_lo;  //pf?
  e2k_cr0_hi_t   cr0_hi;  //return address
  e2k_cr1_lo_t   cr1_lo;  //mess of fields - includes
                          // interrupt enable flags
  e2k_cr1_hi_t   cr1_hi;  //more fields - includes register
                          // window and stuff
} e2k_mem_crs_t;
```

[54] Current frame address = base + index.

So the return `%ctpr3` instruction is essentially saying "pop the current frame off the PCS into `cr0` and `cr1` and stick the return address in `ctpr3`."

Branching

A similar construction is used for basic branches. Rather than flags or conditions registers like in x86 or ARM, VLIW processors often have a full set of condition registers called "predicate" registers. These allow the compiler to set up a sequence where multiple comparisons can happen in advance of a branch, and then a branch can be based on multiple predicates, or a sequence of branches can occur using the different predicate registers.

Here's a common design pattern seen in Elbrus branches. The following is essentially implementing `if (condition) { function(); }` in C.

```
disp %ctpr1, 0x10d48
cmpedb,0 %dr0, 0x0, %pred0
ct %ctpr1 ? %pred0
```

First the `disp` instruction indicates to the pipeline the control transfer target, the function address. Then in the `cmpedb` instruction `dr0` is compared to `0` and the result placed in `%pred0` (true or false). Finally if `%pred0` is true then the ct instruction causes a control transfer, otherwise we fall through to the next instruction.

Conclusion

Elbrus processors are pretty capable and make decent Linux machines. While the Elbrus CPU may be under powered compared with similar Intel or ARM server processors, given the Russian geopolitical situation, these guys are going to stick around for a while. Elbrus's VLIW architecture and register windowing will pose additional challenges for exploit writers. Fortunately, the Elbrus component instructions are very RISC-like, despite the wide command format.

In this article, we've explored the basics of the instruction set and the PCS using publicly available documentation. There's a lot more to learn, however. We'll need some full documentation to start plumbing the depths of things like virtual memory, interrupt and exception handling, and the boot process.

22:11 Janus Polyglot

by Harvey Phillips

Who left among you hold any faith in the empty promises of filetypes? Who is yet to accept to that beauty is in the eye of the parser? I hope that this gentle stroll through 512 harmless looking bytes will dispel any remaining myths that you, dear neighbours, continue to clutch to your collective chests.

Regular readers of this fine journal may have seen my and @netspooky's articles in PoC||GTFO 21:09 and 21:10 respectively. Those were write-ups for the Binary Golf Grand Prix back in 2020 where the challenge was to produce the smallest palindromic binary. 2021's edition of this wonderful challenge pitted competitors against one another in a battle to produce the smallest polyglot binary. There were two possible avenues of attack that were scored separately: you could either connive the smallest polyglot that was executable as a binary, or rack up points for every parser that successfully processed your entry. We decided to normalize scores by filesize for this second category, so that the emphasis was still on as small a collection of bytes as possible.

Feeling drawn towards this latter category, and seeing as the competition's name begins with the word "binary," I chose an x86 bootloader as my host binary. For those who haven't delighted in the pleasures of 16-bit real-mode assembly, a bootloader for x86 machines is a 512-byte blob that ends in `0x55aa`. Execution begins at offset `0x0`. That's it.

Ordinarily such a bootloader would be responsible for loading some more bytes into memory from a hard drive and jumping to it, maybe setting up a stack or other registers along the way. The nice thing about choosing an essentially format-less format is that I can shift around the code and data portions of the bootloader to make way for what is to come. I just have to fit in an appropriate `jmp` instruction to keep the execution flow flowing.

So, what does this bootloader do? I thought it'd be fun to have a single string in the polyglot that I could either print for executables, or extract for archive formats. With this in mind, I reused some old 16-bit real mode assembly I wrote for printing strings to the screen. Printing a string to stdout in Linux is very straightforward thanks to the blessing of syscalls. (We do it later on with just a few lines.) However, 16-bit real mode affords us no such niceties.

The *very* rough analogue of the syscall in 16-bit x86 is the interrupt. Your BIOS may be getting old now, but it still offers a wealth of prewritten routines for you to use. Calling a routine via an interrupt is strikingly similar (for good reason!) to using a syscall: set a value corresponding to the routine you want in `ah`, arguments in `bl`, `bh`, etc, and throw the interrupt. As an example, let's look at the very first of my routines that the bootloader portion will call into, `clearScreen`:

```
pusha              ; Save state
mov ah, 0x6        ; "Scroll Up Window" routine
xor al, al         ; Number of lines to scroll
                   ; (0x0 is the whole screen)
mov bh, 0x03       ; Colours: fg black, bg cyan
xor cx, cx         ; (CH,CL) = coordinates of
                   ; upper left corner
mov dx, 0x184f     ; (DH,DL) = coordinates of
                   ; lower right corner
int 0x10           ; Graphics Interrupt
popa               ; Restore state
ret
```

Pretty straightforward, right? Routine `0x6` from interrupt `0x10` is Scroll Up Window. We set some arguments in the other registers and then kick things off with `int 0x10`. The reason we need to scroll the screen at all is because it's usual for the BIOS to have left some text in the screen buffer as it loads, and we want to get rid of it.

Once we've cleared the screen, we use another BIOS routine to set the cursor position, then we store the memory location of our string in the `si` register before calling our `printString` function. (Yes — the BIOS does *not* provide a routine for printing strings!) However, it's easy enough as we are provided with a Display Character (TTY Output) routine by the Graphics Interrupt `0x10`. So, we simply loop over the bytes of our string, calling this BIOS routine each time until we hit a NULL byte. Just for added panache, I inserted a `delay` routine in between printing each character.

Running the polyglot in QEMU with `qemu-system-x86_64 janus.com` will spell out the string.[55]

[55] `unzip pocorgtfo22.pdf janus.zip`

COM Shenanigans

How different really is a bootloader to a COM file? A COM executable doesn't need that pesky 0x55aa at the end, and there isn't a hard byte count to deal with. However, if you take an x86 bootloader like I had started off with and run it in dosbox, you don't see any output. No errors either, but what has happened to my beautiful string that the BIOS prints? The answer lies in the console buffer. Despite its lowly appearance to today's behemoths, DOS is indeed an operating system (hence the letters O and S), and it does perform some slight attempts at memory management. The console buffer where we enter commands and see their output in DOS is not mapped to the same memory as the BIOS's TTY output buffer. This means that our assembly is still writing our string, but to somewhere else in memory that doesn't show on the screen!

Fortunately, one of the many blessings of DOS is interrupt 0x21. One of the routines provided by this interrupt gives the ability to write a string to the DOS equivalent of stdout. The only thing we need to be aware of is that this routine expects such strings to be $-terminated. Yet more fortune is at our door upon discovering that interrupt `0x21` isn't mapped to anything by the BIOS — `int 0x21` doesn't do anything if we aren't in DOS!

By modifying our `printString` routine in our source, we can first print the string in a DOS manner, and then in a BIOS manner. All we need to do is append a $ to our string (after the null-byte that the BIOS routine looks for so we don't see it in either output), and remember that the offset to the string in memory is different in DOS than it is in BIOS.

While I used nasm for the fine-grained byte control it gave me, I opted to use its `org` directive to tell it to compute offsets relative to `0x7c00`, the bootloader load address on x86 machines. This means that any other offsets to the string for non-bootloader sections of executable code would need to be calculated manually. For DOS, this is no hassle as binaries are loaded at address 0x100, meaning I only have to add 0x100 to whatever the file offset of my string is.

So, the `printString` routine ends up like this:

```
printString:
  ; DOS Version
  push cs          ; Set CS=DS
  pop ds
  mov dx, 0x211    ; Offset to string
  mov ah, 0x9      ; 0x9: Write to DOS stdout
  int 0x21         ; DOS Interrupt
  mov ax, 0x4c02   ; 0x4c: DOS Exit
                   ; 0x02: Return Value
  int 0x21         ; DOS Interrupt

  ; BIOS Version
  pusha
  .loop:
    lodsb           ; Load char from SI to AL
    test al, al     ; Check for null-byte
    jz .end
    call printChar ; Print it.
    call delay      ; Lazy animation effect
  jmp .loop
  .end:
    popa
    jmp waitForKeypress  ; Will loop if held
```

One final caveat for DOS fun, we need the file to have a `.com` file extension! Fortunately, none of the other parsers that *janus* supports had a file extension as a hard requirement.

We can test the COM portions work as expected by running `janus.com` under DOSBox.

ELF Shenanigans

From chaos, evolved structure — and so we shall also find that from the disarray of real-mode, we are able to find order in the ELF specification.

An ELF file contains a great deal of structure, but many fields in its various headers are ignored by the Linux kernel's ELF loader. We can play this to our advantage by populating those fields with content for other parsers! For a recent overview, I suggest that readers take a look at tmp.0ut's Issue 1:1 where I peruse the various fields in the ELF and program headers, foraging for those that we are free to do with as we like — without affecting execution. For a more fiendish appraisal of these fields, I highly recommend @netspooky's series of blogs chronicling his journey to produce an 85-byte ELF.[56]

[56] `https://n0.lol/ebm/1.html`

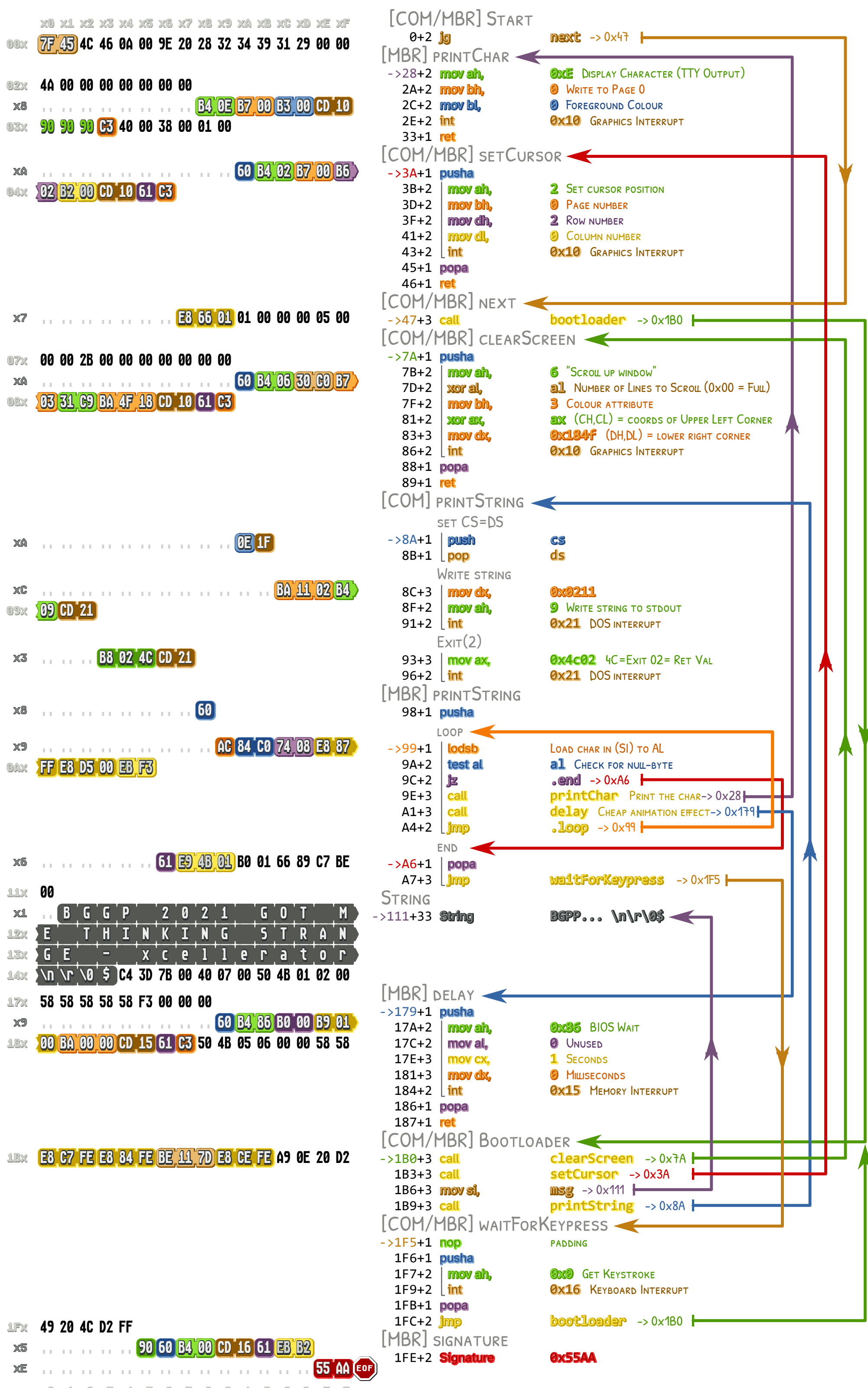


Figure 10: COM and MBR's side of Janus

For the attendees in the back who are not fully acquainted with the internals of ELF, here is very brief overview of the parts relevant to us:

- The ELF header (`\x7fELF`) must begin at offset 0x0
- The `e_phoff` field of the ELF header is a file offset to first program header
- The first (and in our case, only) program header will detail where our x64 Linux assembly can be found, and where it is to be loaded in memory

The important takeaway here is that, although our ELF header has to begin at offset 0x0, the program header can appear much later because we provide the ELF parser with an offset to it. However, we do have a potential issue: we already have something at offset 0x0, the entry point to the BIOS and COM assembly!

The first few bytes of the ELF header (and therefore any valid ELF file) are `\x7fELF`, which disassemble as 16-bit real-mode instructions to:

```
jg 0x47
dec sp
inc si
```

So, upon our dutiful BIOS loading this particular collection of bytes into memory and jumping to offset 0x0, it will immediately jump to offset `0x47`, thanks to how the EFLAGS register is initialized at boot. (At least in SeaBIOS that QEMU uses — I'd be very interested if any neighbours know of any variance in this observation!) Therefore, all we are required to do in order to overcome this calamity is move our real-mode assembly elsewhere, and place yet another `jmp` to it at offset `0x47`. This way, after bouncing around a few times, our BIOS and DOS functionality is preserved.

Populating the beginning of our file with an ELF header, and armed with a list of fields that we know are ignored by the Linux loader, we can fill in several gaps with more interesting things. At this stage of my design, I simply left these fields with X's so that I could come back later and put something fun in its place. Several of the real-mode routines are small enough that they fit in overlooked `uint64_t` fields. Can you spot them all?

Lastly, an ELF that presents itself as executable in its header requires something to execute! Running with the same theme of printing the string already present in the file, I used:

```
1 mov al, 0x1       ; SYS_WRITE
  mov di, ax        ; Write to stdout
3                   ; (file descriptor 1)
  mov esi, 0x400111 ; Virtual memory address of the
5                   ; string: 0x400000 + file offset
  mov dl, 0x32      ; String length
7 syscall
  mov al, 0x3c      ; SYS_EXIT
9 inc di            ; Return value 0x2
  syscall
```

Notice that we have to calculate the virtual address of the string manually again! The string appears at file offset `0x111`, and our ELF is loaded to address `0x400000`. Adding the two gives us the right address.

As a final touch, we can now set the size of our file to be loaded in the `p_filesz` and `p_memsz` fields of the program header, set `p_offset` to `0x0` so we load the entire 512 bytes, and at long last we can set `e_entry` so that the Linux loader knows what virtual memory address to jump to after loading our ELF into memory.

To test things are as they should be, we can run the binary in any x64 Linux distro.

RAR Shenanigans

Long time neighbours will no doubt have seen several polyglots over the years incorporating the RAR file format. It was my intention all along for each of the incorporated file formats to make use the same string over and over again, either printing it or decompressing to it. Fortunately, RAR (and as we'll see later, ZIP) supports containing files without compression, meaning we can just dress up our string with the appropriate structures and `unrar` should play fair!

For anyone looking to get a decent handle on the RAR format, Ange Albertini's poster on page 57 is an invaluable first step. Looking at this, we see a reasonably straightforward structure to the file. One of the several fun things about the RAR format is that the `Rar!` magic can appear at *any offset* in the file, which means we aren't bound to place the RAR part of the file at any particular location.

However, unlike in the executable portions of *janus*, we can't point the `unrar` parser to any location we like for our (un)compressed data. Indeed, the RAR File Header must immediately prepend the data, and the Archive End structure immediately follows it. This is one of the first hard restrictions on our binary. We have a whole `0x3d` bytes before our string, and another `0x7` bytes after it. If we

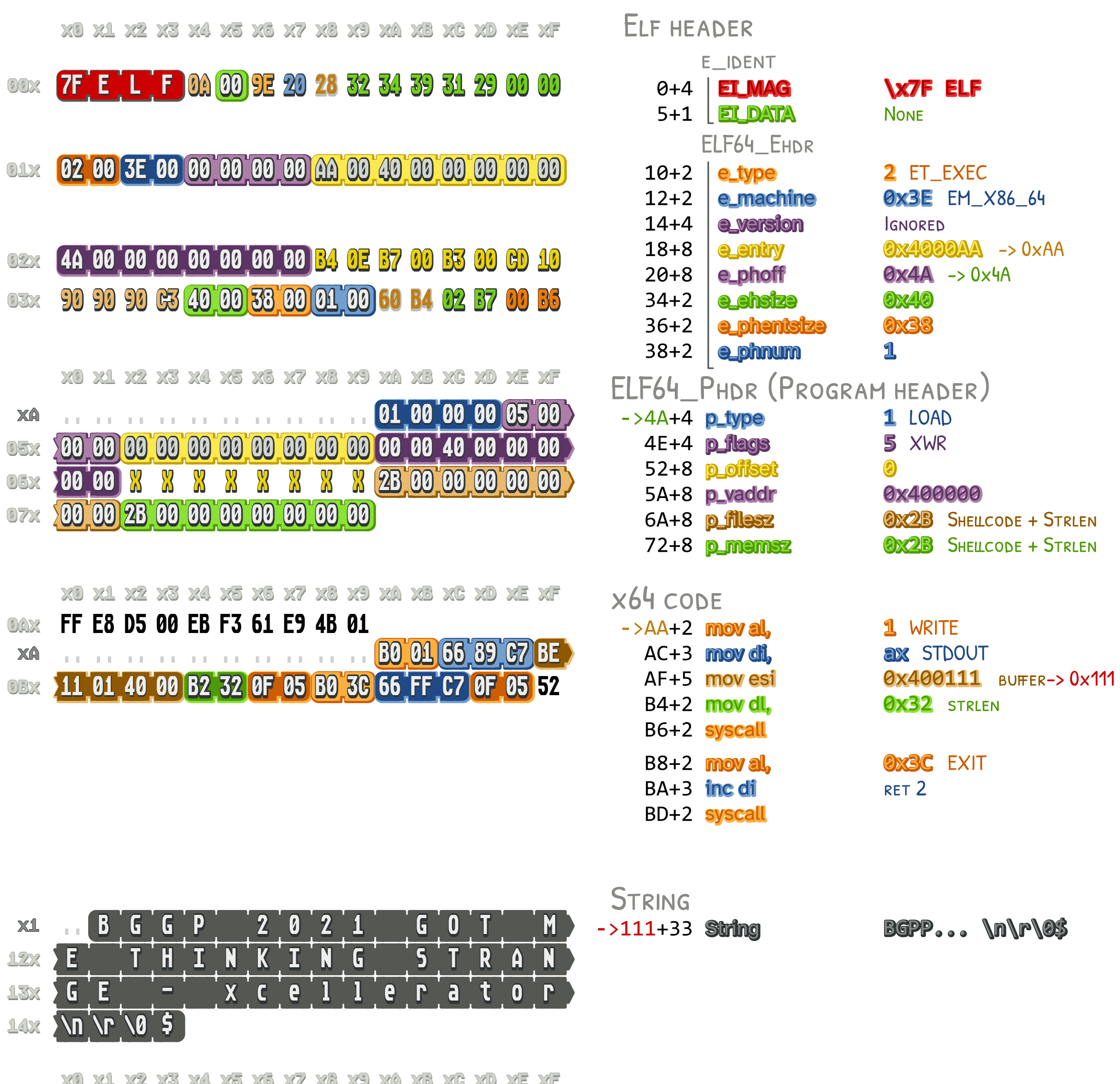


Figure 11: ELF's side of Janus

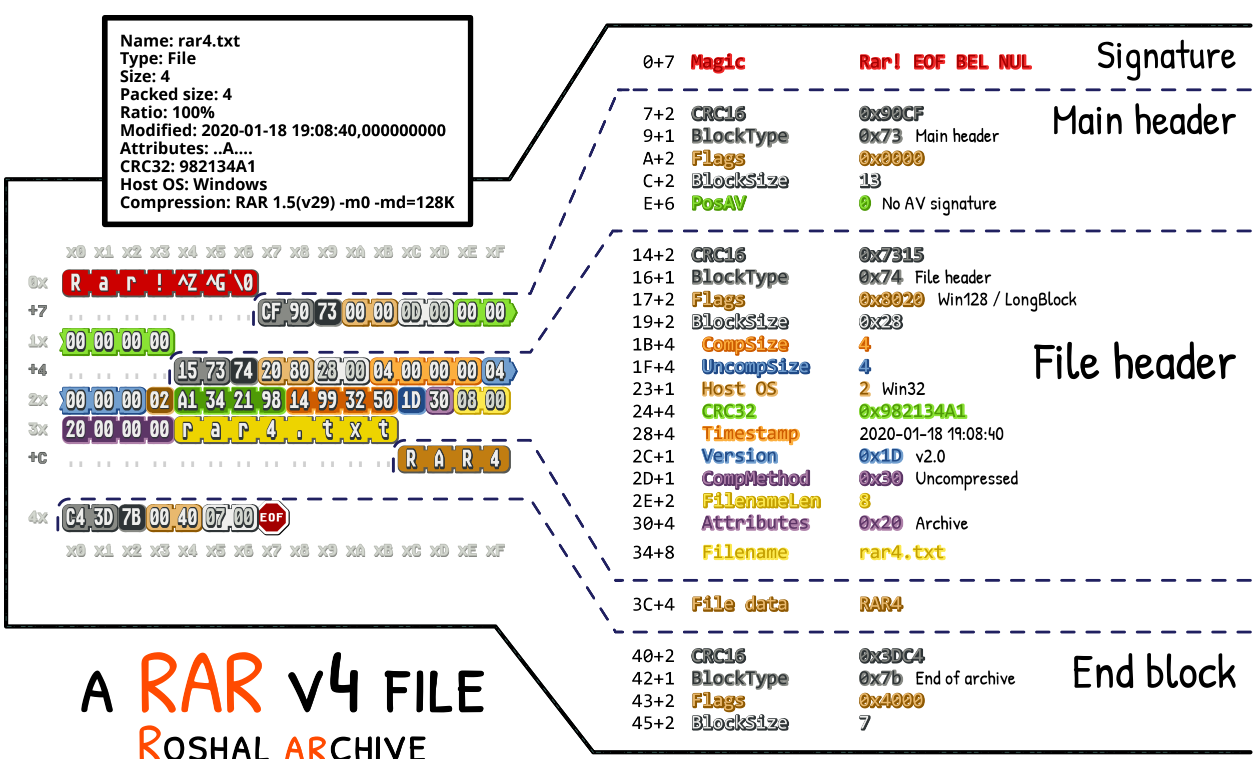


Figure 12: Ange Albertini's Poster on RAR Format

want to relocate our string later, we have to move all these surrounding bytes with it.

There is one slight loophole here that we will certainly play to our advantage when it comes to ZIP shenanigans: the field at the end of the File Header, just before our string begins, is the filename. Ordinarily, the filename would be just that, the filename. There is even a separate field for the filename length, so we don't have to null-terminate it or anything like that. It turns out that the filename can actually be anything we want—including non-printable characters!

There is a pretty big downside to all of this RAR business. Although we control the size of the data in the File Header, the `unrar` parser will not tolerate any junk between the end of the compressed data and the start of the Archive End. Therefore, extracting our string with `unrar` will include the `\n\r\0$` bytes in its output. I thought about possible ways around this due to its esthetically displeasing nature, but it seems to be a necessary evil.

There were two major stumbling blocks I found along the way. The first was the CRC. In the format specification, it occupies the top two bytes in each of the Main Header, File Header and Archive End structures. Leaving these bytes as NULLs made `unrar` complain about a CRC error, so I was reasonably confident that the rest of the bytes were correct. I had seen in various sources that the CRC was a CRC16, but after trying several times with different regions of bytes, and different polynomials, I couldn't find anything that worked.

Eventually, I resorted to RTFM'ing and I dragged up the UnRAR sourcecode. This is found in `rawread.cpp`.

```
// RAR 1.5 block CRC.
uint RawRead::GetCRC15(bool ProcessedOnly) {
  if (DataSize<=2)
    return 0;
  uint HeaderCRC=CRC32(0xffffffff,&Data[2],
      (ProcessedOnly ? ReadPos:DataSize)-2);
  return ~HeaderCRC & 0xffff;
}
```

After smacking my head against the desk a few times, I tried computing the CRC32 of the Main Header, and chopped off the top two bytes to obtain `0x90cf`—precisely the CRC of the Main Header from the reference I used. A truncated CRC32 is most certainly not the same as a CRC16! Had I

begun by looking at the unrar sourcecode instead of trying to brute force various CRC16 polynomials to find a match where there was none, I would have saved myself several evenings. Fortunately, the python zlib library offers a crc32() function which precisely computes the CRC we need:

```
>>> header = bytes.fromhex(
                '7300000d00000000000000' )
>>> hex( zlib.crc32(header) & 0xffff )
'0x90cf'
```

The second confusing feature of the RAR format was the datetime format in the timestamp field of the File Header. Eventually, I found it documented in one of the Kaitai Struct examples.[57] It's just a bitfield, common in DOS-land. Both the date and time occupy a uint16 each.

```
year   = ((date & 0b1111111000000000) >> 9) + 1980
month  =  (date & 0b0000000111100000) >> 5
day    =  (date & 0b0000000000011111)
hour   =  (time & 0b1111100000000000) >> 11
minute =  (time & 0b0000011111100000) >> 5
second =  (time & 0b0000000000011111) * 2
```

To be confident things are working properly, unrar p janus.com happily produces our string, with the unfortunate extra $ on the end.

ZIP Shenanigans

If you are not yet acquainted with the details of the PKZIP format, and felt that incorporating a RAR into our polyglot was intricate, I have bad news for you. But the PKZIP format actually lends itself very nicely to polyglots! The thing that makes it unique (at least in my experience) is that a proper PKZIP parser, will process a file *backward*. Typically, we think of parsers are looking for some magic value which indicates the start of the data it should parse. PKZIP flips everything on its head and instead looks for the End of Central Directory signature, which comes at the end of the file.

In this End of Central Directory, there is a file offset and size of the Central Directory. The Central Directory holds all the information about our (un)compressed files contained within, including their filenames, and CRCs. (This time around, it's just a CRC32.) Also included in this directory are offsets to our data, which is always prepended by a Local File Header.

Let's take a moment to ponder this last point. Our data (the string we keep re-using) must be prepended by the PKZIP Local File Header. But we've already added our RAR shenanigans which also required our data being prepended by something. (In the case, it was the similarly named File Header.) How can we reconcile these two facts? The trick lies in something I hinted at earlier! The final field of the RAR File Header, which comes immediately prior to the start of our string, is the filename of the to-be-extracted file. Seeing as we aren't too fussed by actually extracting this string to a file with unrar, we can simply use this filename field to store the PKZIP Local File Header! The downside is that we'll end up with a nasty filename in our directory if we run unrar with the x switch. (Try unrar p janus.com instead.) This seems like a small price to pay in order for RAR and PKZIP to peacefully coexist!

As other devotees of weird machines will no doubt be familiar, when a trick like smuggling binary data in filenames works with one format, we are led to ask whether it will work elsewhere? If the RAR specification outlines no consequences for unpleasantness in a filename, does the PKZIP specification also afford us this luxury? It does!

In contrast to the RAR format, the filename in PKZIP lies in the Central Directory rather than the Local File Header. This means that the filename according to PKZIP actually occurs later in the file, whereas RAR believes the filename lies just before the data begins. This trick wasn't actually needed based on the file formats that I selected for inclusion in my polyglot, but it may well be useful to you in future endeavours. In my case, I opted to place one of the 16-bit real mode routines into the PKZIP filename, namely the delay routine. When was the last time one of your binaries executed a filename as machine code?

GNU Multiboot2 Shenanigans

At what point do we call something a file format? How much *format* does there have to be to a *file*? I ask because I have trouble identifying this next inclusion with an actual file format. Indeed, the GNU Multiboot2 format has a specification and a parser (grub-file from the grub2 project).[58] But… well, read on and see for yourself if you agree with my

[57] rar.ksy, near line 151.
[58] https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
1Ax	D6	50	52	E8	00	00	00	00	00	01	00	00	2A	AE	AD	17

[GNU] Multiboot 2.02

1A0+4	Magic	0xE85250D6
1A4+4	Architecture	0 i386
1A8+4	Header length	0x100
1AC+4	Checksum	0x17ADAE2A

Figure 13: Multiboot's side of Janus

feeling of cheekiness in including it in my polyglot.

The GNU Multiboot2 is a pretty straightforward specification that allows a bootloader like GRUB to boot a file without having to go via the BIOS. GRUB will parse a file top-to-bottom looking for the magic (`0xE85250D6`), so we can have anything we like both before and after the relevant bytes. In total, we require four uint32's worth of bytes, but we have to be 64-bit aligned, so I ended up with an additional four bytes of padding to round off the PKZIP End of Central Directory.

The format is as follows: Magic, Architecture, Header Length, Checksum. That's it. I already mentioned that the magic is `0xE85250D6`. The architecture value corresponding to i386 is simply `0x0` and the header length is self-explanatory. The only thing worth commenting on here is the checksum. It's possibly the simplest checksum I've ever encountered: the unsigned 32-bit sum of the magic, architecture, header length and checksum is `0x0`. Simple!

So, all that was required to be able to claim another file format in my polyglot was to find room for 20 bytes, including four bytes of padding! Cheeky? Absolutely. Technically correct? Absolutely.

If you have GRUB installed on your machine, you can test the validity of the polyglot as a GNU Multiboot2 image with `grub-file -is-x86-multiboot2 janus.com`. There should be no output, but `echo $?` will inform you that `grub-file` returned `0`.

Commodore 64 Shenanigans

Up until this point, we've been playing around with well trodden parsers and specifications. It was certainly a lot of fun getting to this point, but when I looked back at my in-progress polyglot in a hex editor, I saw lots of empty space. This displeased me. A certain idea had been bugging me for a while as I was working on this project: could I incorporate support for an 8-bit computer? Back in the 80s, when 8-bit machines reigned supreme, hard drives were prohibitively expensive for most people, so programs were typically stored on floppies and cassettes. My initial approach was to explore the tape format of the ZX Spectrum—falsely expecting it to be reasonably malleable to the kinds of distortions that are suitable for polyglotting. A week goes by and I realised that it wasn't going to work. (For those interested: Kaitai Struct already has excellent support for this format.)

The next thing to try on my list was the Commodore 64 PRG format, which turned out to *only just* be possible! As you'll see further down, we end up having part of our ELF header form lines of BASIC, and we make use of 75% of a uint32. This was my first time playing with machines and architectures from this era, and it was a lot of fun!

(Note to the reader: in keeping with 8-bit tradition, hexadecimal values in this section are prepended by '`$`'.)

For any neighbour unacquainted with the wonders of the Commodore 64, it is an 8-bit computer first released in 1982. It's powered by an 8-bit 6502 CPU and sports 64k of RAM. All pointers are two bytes long. The primary way to interface with the machine is the BASIC interpreter, which it boots to. There are several different file formats that can be loaded into memory from either floppy, cassette or even cartridge. (The cartridge was a distinctly North American luxury that my European ancestors were seemingly deprived of.) In my case, I went for the most common file format: PRG, short for "program."

Before we even begin looking at the structure of these files, we need to know something about how they are loaded into memory. Indeed, confusingly enough there are two different ways: absolute and non-absolute. The difference is whether the Commodore 64 will load the PRG file where it wants to be loaded, or just ignore it and load it to the start of BASIC RAM at `$0800`. This was important because of the lack of dynamic linking at the time; many programs had hard-coded offsets that required being loaded to a particular address in order to make any sense.

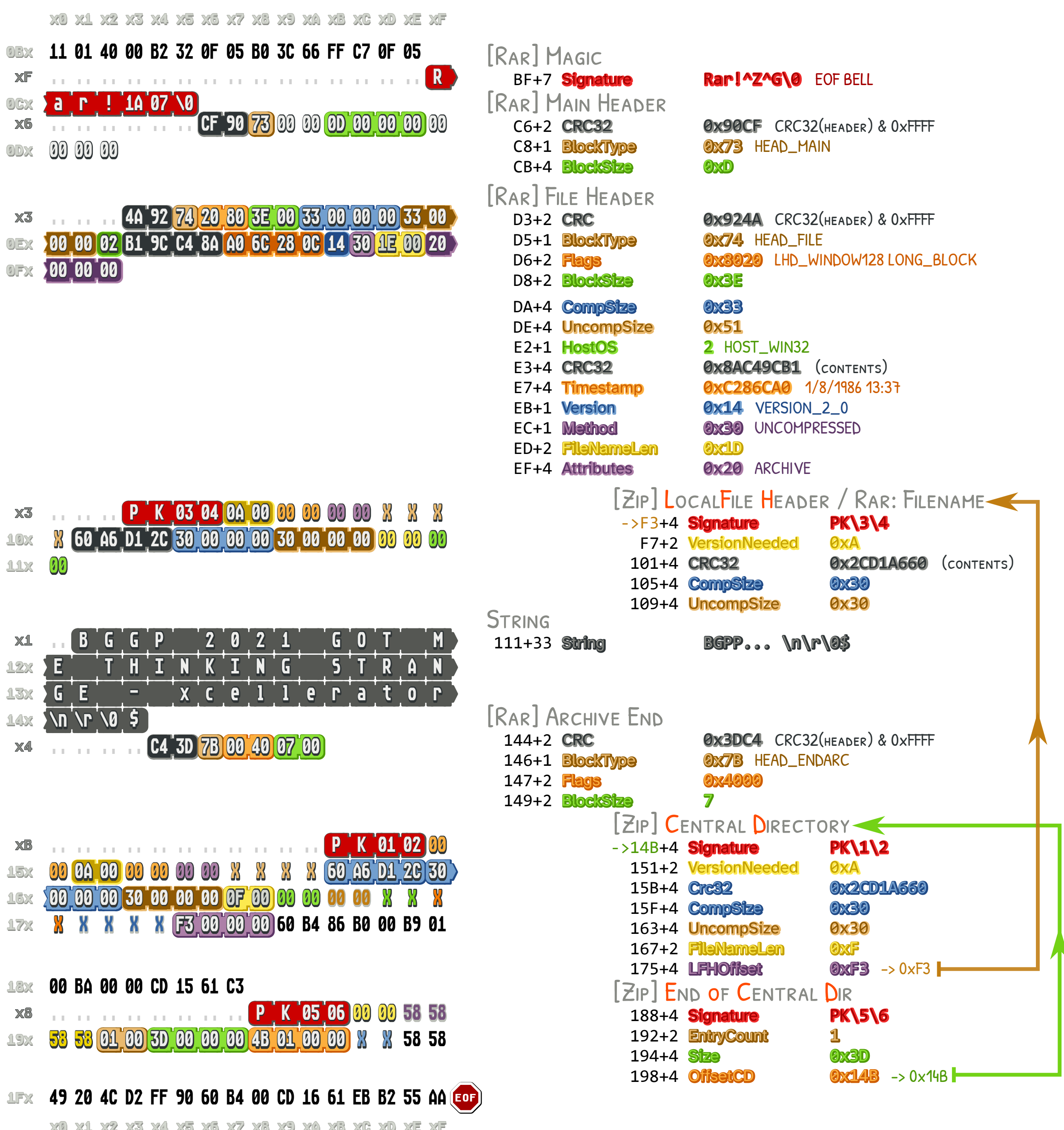


Figure 14: Rar and Zip's sides of Janus

We are very lucky that this is the case! The first two bytes of a PRG file are a pointer to where in memory the PRG is *supposed* to be loaded. In our case, this is `$7f45` (the start of the ELF magic), which is *not* a valid location for a BASIC program to be loaded to. However, by loading our PRG in non-absolute mode, these bytes are ignored, although they must still be present.

The next two bytes are *supposed* to be a pointer to the first line of BASIC. We are stuck with this being `$4c46`. (This is the '`LF`' of the ELF magic.) Non-absolute mode to the rescue! Our file is going to just be parsed sequentially instead of hopping around for lines of BASIC to interpret.

What comes next is a line of BASIC. I'm sure many readers will have written some BASIC before, even those like myself who are too young to have lived through BASIC's heyday. But what does a line of BASIC look like *on disk*? Disk space was a premium back in the 80s and it didn't make sense to store entire words like `PRINT`, `PEEK` and `POKE` when a single byte could accomplish the same job. Fortunately for the programmers, commands like `LIST` automatically converted the tokenized BASIC on disk and in memory to the much more familiar and verbose form that we all know.

So, according to a PRG file, a line of BASIC is composed of: a two-byte little-endian line number, a single byte BASIC token, arguments in PETSCII (kinda like ASCII, as we'll see in a bit), and a NULL terminator. Here we are at offset +`0x4` into our ELF header, writing BASIC! Out of respect and deference to the old ways, our first line number is going to be 10, but what are we going to actually do?

As we don't have a whole lot of room to do much of anything in before the ELF header starts getting picky with us, we have to move our execution somewhere else as soon as possible. The easiest thing to do is to make our BASIC program simply jump to some 6502 machine code with the `SYS` instruction and then terminate. That sounds easy enough, apart from having to write 6502 assembly. Let's focus on cramming our minimal BASIC program into what little space we have first, then we can figure out where to pass execution to later.

On page 62, we have the first 24 bytes of `janus.com`, with both the ELF and Commodore 64 interpretations of each byte. Let's take it from the top:

As already mentioned, the first `$7f45` pointer would be the load address of the PRG if we loaded in absolute mode, so these bytes are ignored, as are the next two bytes `$4c46`, which completes the ELF magic.

Now comes `$0a00`, or "10", which is our first BASIC line number. The ELF parser believes this to be `EI_CLASS` and `EI_DATA`. Next up we have `$9e` which is the BASIC token for the `SYS` instruction, which will jump to executing 6502 instructions at the *decimal* address we provide it. ELF parsers believe this byte to be `EI_VERSION`. Asking `readelf`, we are informed that the version is 158, or `0x9e` in hex. So far so good!

Next up is the argument to the SYS instruction: "`(2491)`". The actual number is variable, and for a long time I left this as 1234 until I knew exactly where in memory my 6502 instructions would be. These bytes occupy the region that the ELF spec identifies as `EI_PAD`. (The elf man page is a terrific quick reference for all these structs. In this case we're looking at `Elf64_Ehdr`.)

Assuming our 6502 instructions do what we want and culminate with a `rts` instruction, we will end up back in BASIC and we should be good? But no, our BASIC program will continue running, and we need to gracefully finish it. Unfortunately, the next few bytes form the `e_type` and `e_machine` fields of the ELF header, which we cannot mess around with. Any deviation from their current state will result in the ELF not running under Linux.

So, what does the Commodore 64 think these bytes mean if we just leave them alone? First, notice that we're actually off-by-one between the ELF and Commodore 64 interpretations now: the final byte of `EI_PAD` is `0x00`, but forms part of the `$0002` pointer to the next line of BASIC. Similarly, the `0x02` byte is the *start* of the `0x0200` `e_type` field of the ELF header!

We have `$0002` as a pointer to a line of BASIC, but that gets ignored unless we're in absolute mode (we aren't). The bytes that follow, `$003e`, is the BASIC line number, in little-endian! `0x3e00` is 15,872 in decimal, and indeed, if we run `LIST` on the Commodore 64 after loading this PRG, we see:

```
10 SYS (2491)
15872
```

So, in other words, the second byte of `e_type` and first byte of `e_machine` are interpreted as a BASIC line number! Pretty cool! To finish up our BASIC program, we have an instant null byte which ends line 15872 of BASIC, which is also the second

```
EI_MAGIC
|           EI_CLASS
|           |  EI_DATA
|           |  |  EI_VERSION
|           |  |  |  EI_OSABI
|           |  |  |  |  EI_PAD
|           |  |  |  |  |                       e_type
|           |  |  |  |  |                       |     e_machine
|           |  |  |  |  |                       |     |     e_version
|           |  |  |  |  |                       |     |     |           |
 7F 45 4C 46 0A 00 9E 20 28 32 34 39 31 29 00 00 02 00 3E 00 00 00 00 00
|     |     |     |  |                       |     |     |  |     |  |
|     |     |     |  |                       |     |     |  |     Empty line, FIN
|     |     |     |  |                       |     |     |  Pointer to next line(ignored)
|     |     |     |  |                       |     |     Empty line
|     |     |     |  |                       |     Line Number (15872)
|     |     |     |  |                       Pointer to next line (ignored)
|     |     |     |  " (2491)"
|     |     |     SYS Token
|     |     Line Number (10)
|     Pointer to first line (ignored)
Load address (ignored)
```

Figure 15: First 24 Bytes of Polyglot, labeled for ELF and Commodore 64 parsing

byte of `e_machine` (`0x3e00`). Finally, we have a null pointer to the next line of BASIC, followed by another null. This indicates the end of the BASIC program. But here we have the fun observation I mentioned earlier: our BASIC program ends 3 bytes into the `e_version` `uint32`. The last byte only matters to the ELF parser! And with just one byte to spare! The next field in the ELF header is `e_entry`, which is the entrypoint of the program and something we definitely can't screw with. Phew!

Okay great, we can just about smuggle a very small BASIC program into the ELF header which will jump to any location we specify and start executing 6502 instructions. To top it off, as long as we finish our machine code with an `rts` instruction, our program will gracefully terminate and return to the BASIC prompt! So, what are we going to do? Print our string of course!

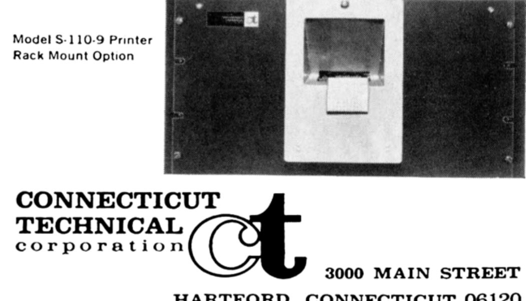
If, like me, you have never written or even looked at 6502 assembly before, you'll see that it's not too hard, but does have a couple of quirks. For one, the 6502 is an 8-bit CPU, but the Commodore 64 has 64k of memory. This means that every pointer takes two store operations to store in memory: one for the low byte, and another for the high byte.

Secondly, the first page (256 bytes) of memory is called the zero page. (The first byte of any pointer indicates which page it's in.) The zero page is special because we can index structures within it with just a single byte, i.e. we only need one register to be an index into our string if we store it in the zero-page.

TAPE LISTING PRINTERS

Price is all we have to offer! Our tape listing printers are slow. They accept and print up to a nine character data line, and it takes almost a whole second. We don't know how long they will continue to do this, because we've only been selling them for six years and nobody tells us when they stop. This is why we have to sell the darn thing to OEM's at a starting price of $275. If you just want the insides you can buy it for less. For a Bingo card circle you can have a data and price sheet. Please forgive the cheap ad, the mark-up on the printer isn't sufficient to pay for a fancy campaign. It's just the plain, unadulterated truth. Send for facts today—Bingo—letter or phone. You'll receive a happy surprise, maybe, we hope.

Model S-110-9 Printer
Rack Mount Option



CONNECTICUT TECHNICAL corporation
3000 MAIN STREET
HARTFORD, CONNECTICUT 06120

Despite ASCII being nearly twenty years old when the C64 was first released, it instead uses PETSCII, which supports two slightly different layouts. At boot, it has the first character set loaded with only has capital letters. Our string has lowercase letters too, but if we try printing it now, we'll see it all caps. We can load the alternative character set (which *does* include lowercase) by "printing" the byte `0x0e`. We do this using the C64 CHROUT routine which lives at `$ffd2` in the Commodore's KERNAL ROM. All we have to do is put `0x0e` in the `A` register and jump to the right address (`$ffd2`):

```
lda #0x0e
jsr $ffd2
```

Next we have to store a pointer to our string in the zero-page. I chose `$0020` for this, so we'll be storing bytes at `$0020` and `$0021`. Instead of working out manually where my string would be, I just loaded the binary in the VICE emulator and used the built-in monitor (debugger to you and me), to see where it ended up. It turns out the string lives at `$0910`. (BASIC RAM starts at `$0800`, so this feels about right.) Storing the pointer simply looks like:

```
lda #0x09        ; Load 0x09 in A
sta $21          ; Store byte in A in address $0021
lda #0x10        ; Load 0x10 in A
sta $20          ; Store byte in A in address $0020
```

A little unusual to modern eyes, but still pretty straightforward. Lastly, we just need to write some logic to loop over our string, checking for a null-byte terminator, and then return control to the BASIC interpreter with `rts`.

There are two final quirks to consider. First, the Commodore 64 has a 40-character wide display, but my string is longer than that. I opted to include a manual line break after 33 characters have been printed just so things wrap in a nice way. Similarly, I also print another line break when we're done so that the BASIC prompt appears neatly on the next line.

The other quirk deals with PETSCII again. The string in memory is ASCII because that's what every other format that uses it expects. Is converting from ASCII to PETSCII going to be a royal pain? As fortune would have it, in this second PETSCII character set, the byte representations of the alphanumeric characters differ only in the sixth most significant bit! The alphanumeric characters begin at `0x40` onwards, so we only need to make the conversion for bytes larger than that. Therefore in our character printing routine that the string printing routine calls each loop, we can simply do the following (the ASCII byte to print is in the `A` register):

```
cmp #0x40  ; Compare byte in A to 0x40
bcc +$2    ; Branch if Carry Clear to the jmp
           ; instruction (i.e. if A < 0x40)
eor #0x20  ; Toggle 6th bit..
jmp $ffd2  ; Jump to CHROUT in KERNAL ROM
```

We check to see if the byte is greater than `0x40` ('a' in PETSCII character set 2), if it is, we bitwise-or it with `0b00100000` to flip the 6th bit, and then jump to the CHROUT routine in ROM.

Putting everything together, our 6502 assembly looks like this:

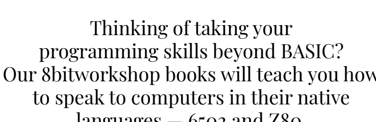
```
lda #0x0e          ; Full Character Set
jsr $ffd2          ; CHROUT

lda #0x09
sta $21            ; High Byte of String
lda #0x10
sta $20            ; Low Byte of String

jsr $09cc          ; Call PRINTSTR
rts                ; Return to BASIC

PRINTSTR:
  ldy #0x0         ; Reset Y register to 0
  LOOP:
    lda ($20),y    ; Read char from zero-page
    cpy #$21       ; Past 33 characters?
    beq +$b        ; If so, jump to EXTRACR
    cmp #$00       ; Null-terminator?
    beq +$d        ; If so, jump to DONE
    jsr $09eb      ; Jump to PRINTCHAR
    iny            ; Increment Y
    jmp $09ce      ; Jump to LOOP
  EXTRACR:
    jsr $09e6      ; Jump to PRINTCR
    jmp $09d4      ; Return to LOOP
  DONE:
    rts            ; Return

PRINTCR:
  lda #13          ; Store CR in A
  jmp $09eb        ; Jump to PRINTCHAR

PRINTCHAR:
  cmp #0x40        ; Greater than 0x40?
  bcc +$2          ; If so, jump to DONE
  eor #0x20        ; Convert ASCII to PETSCII
  DONE:
    jmp $ffd2      ; CHROUT Routine
```

As you can see, it's pretty similar to any other string printing routine in assembly. (For example, the one we wrote for the 16-bit real mode portion of this polyglot.) Sure, there are a couple of extra

quirks in there, but nothing too hazardous. Notice how we were able to use the `Y` register to index our string in the zero-page.

The final part to this Commodore 64 addition is how load this thing? I've mentioned that it's vital to load this PRG in non-absolute mode so that the ELF header can coexist with our BASIC program. This is simple, and can be specified when we use the `LOAD` BASIC instruction: `LOAD "janus.com",8` is all it takes. Notice the lack of an extra `,1` which is usually seen with the `LOAD` command. This extra argument is used to specify whether we are loading in absolute mode or not! Alternatively, if using the VICE emulator like I was, the `-basicload` argument does this for us.

"I Speak 6502. Do You?"



Thinking of taking your programming skills beyond BASIC? Our 8bitworkshop books will teach you how to speak to computers in their native languages — 6502 and Z80.

Use our development tools from the comfort of your home. Write source code in C and watch as we translate it to machine code, then run it on a simulated microcomputer.

To access, enter into your data terminal:
8BITWORKSHOP.COM

Summary

Thank you for joining me on this journey, fellow computer-enjoyers. This whole process was a wild ride of mixed emotions. These 512 bytes took me a few months to assemble into their final form. Like 2020's inaugural Binary Golf Grand Prix, I was convinced that I wouldn't be able to produce an entry, but just kept working on it until something started to come together. Like many readers of this fine journal, I had read the many prior articles on polyglot techniques, but had yet to attempt one of my own.

If you think that this sounds like fun, then you're in luck! The Binary Golf Grand Prix has run now for four years and rumours have it that there are already plans for 2024.

```
000: 7f  E  L  F 0a 00 9e 20  (  2  4  9  1  ) 00 00
010: 02 00 3e 00 00 00 00 00 aa 00 40 00 00 00 00 00
020: 4a 00 00 00 00 00 00 00 b4 0e b7 00 b3 00 cd 10
030: 90 90 90 c3 40 00 38 00 01 00 60 b4 02 b7 00 b6
040: 02 b2 00 cd 10 61 c3 e8 66 01 01 00 00 00 05 00
050: 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00
060: 00 00 58 58 58 58 58 58 58 58 2b 00 00 00 00 00
070: 00 00 2b 00 00 00 00 00 00 00 60 b4 06 30 c0 b7
080: 03 31 c9 ba 4f 18 cd 10 61 c3 0e 1f ba 11 02 b4
090: 09 cd 21 b8 02 4c cd 21 60 ac 84 c0 74 08 e8 87
0a0: ff e8 d5 00 eb f3 61 e9 4b 01 b0 01 66 89 c7 be
0b0: 11 01 40 00 b2 32 0f 05 b0 3c 66 ff c7 0f 05 52
0c0: 61 72 21 1a 07 00 cf 90 73 00 00 0d 00 00 00 00
0d0: 00 00 00 4a 92 74 20 80 3e 00 33 00 00 00 33 00
0e0: 00 00 02 b1 9c c4 8a a0 6c 28 0c 14 30 1e 00 20
0f0: 00 00 00  P  K 03 04 0a 00 00 00 00 00 58 58 58
100: 58 60 a6 d1 2c 30 00 00 00 30 00 00 00 00 00 00
110: 00  B  G  G  P     2  0  2  1     G  O  T     M
120:  E     T  H  I  N  K  I  N  G     S  T  R  A  N
130:  G  E     -     x  c  e  l  l  e  r  a  t  o  r
140: 0a 0d 00 24 c4 3d 7b 00 40 07 00  P  K 01 02 00
150: 00 0a 00 00 00 00 00 58 58 58 58 60 a6 d1 2c 30
160: 00 00 00 30 00 00 00 0f 00 00 00 00 00 58 58 58
170: 58 58 58 58 58 f3 00 00 00 60 b4 86 b0 00 b9 01
180: 00 ba 00 00 cd 15 61 c3  P  K 05 06 00 00 58 58
190: 58 58 01 00 3d 00 00 00 4b 01 00 00 58 58 58 58
1a0: d6 50 52 e8 00 00 00 00 00 01 00 00 2a ae ad 17
1b0: e8 c7 fe e8 84 fe be 11 7d e8 ce fe a9 0e 20 d2
1c0: ff a9 09 85 21 a9 10 85 20 20 cc 09 60 a0 00 b1
1d0: 20 c0 21 f0 0b c9 00 f0 0d 20 eb 09 c8 4c ce 09
1e0: 20 e6 09 4c d4 09 60 a9 0d 4c eb 09 c9 40 90 02
1f0: 49 20 4c d2 ff 90 60 b4 00 cd 16 61 eb b2 55 aa
```

Thanks go to @netspooky for creating and masterminding this competition. Thanks also to everyone who submitted entries last year, as well as the Binary Golf Association for comprehending and scoring them all.

So this is my submission in all its glory: an x86 bootloader, ELF, COM, RAR, ZIP, GNU Multiboot2 Image, and Commodore 64 PRG hybrid. You can find this project with a full nasm listing on GitHub.[59]

Until next time!

[59] `git clone https://github.com/xcellerator/janus`

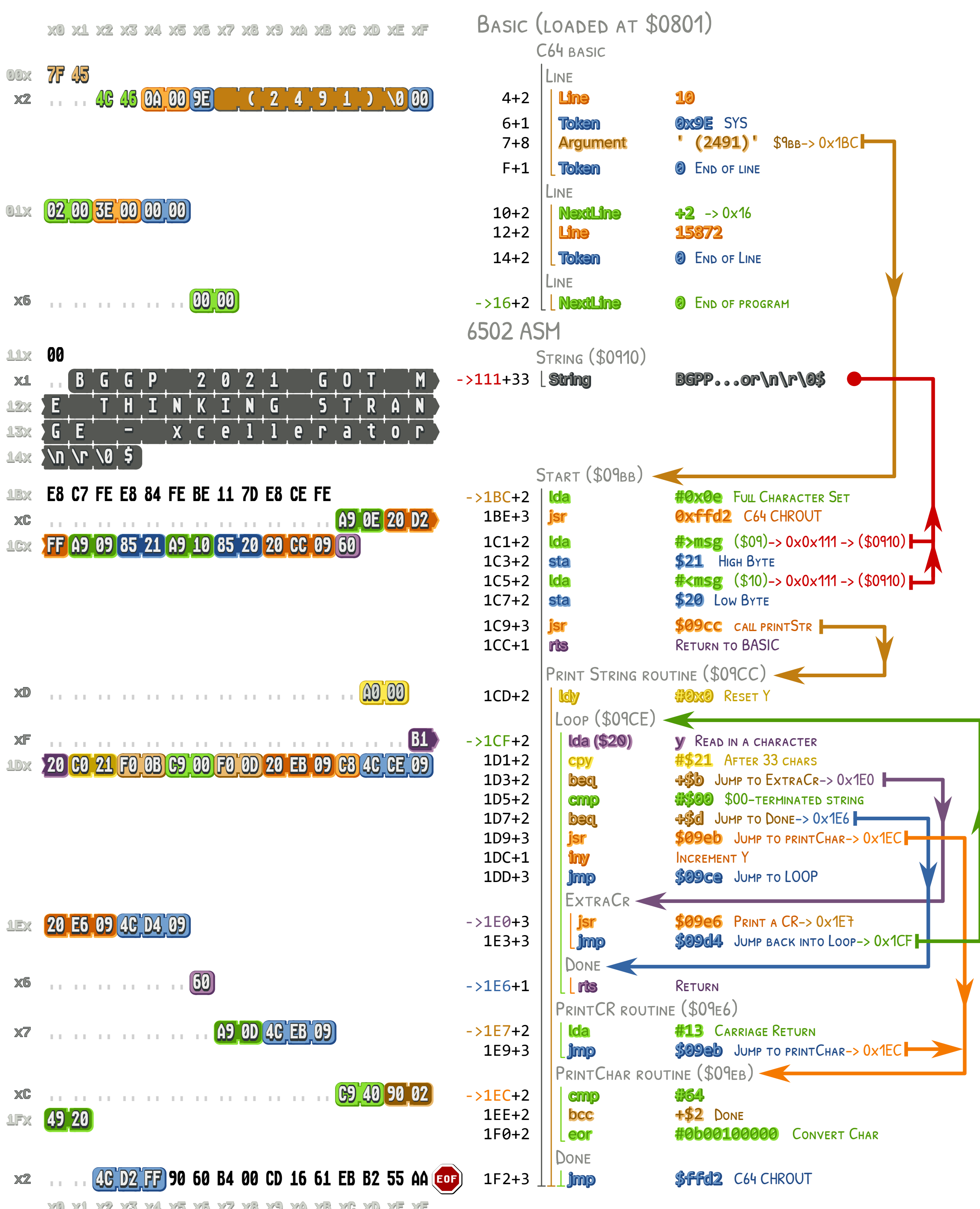


Figure 16: PRG's side of Janus

CIRCUIT DESIGN
Computers are realized in the mind of the Circuit Design Engineer.

at
Honeywell
ELECTRONIC DATA PROCESSING

His goal is always the same — the indiscernible point of least compromise between what is proposed and what is possible. This is electronics engineering at its most basic — designing memories, power systems, logic circuits for proposed systems, investigating experimental innovations and probing application of new knowledge of existing circuitry.

His responsibilities encompass every aspect of the computer system. Working in a purposely informal environment, he and his technicians may design linear circuits for wide band feedback amplifiers; or logical building block circuits that switch milliamperes in nanoseconds; or control element circuits that switch amperes in milliseconds.

At times the work requires only his technician's breadboarding of a relatively elementary circuit. More often, it tests all the Circuit Designer's knowledge and all his design ingenuity and skill; requiring entirely new techniques, new component usage or radical departure from accepted circuit design practices.

He must keep pace with every pertinent development — or face technological obsolescence. His awareness of this fact is reflected in the high ratio of Honeywell Circuit Designers who take full advantage of Honeywell's tuition-paid program at many of the world renowned universities in the Boston-Cambridge area.

Qualified individuals, interested in discussing Circuit Design at Honeywell, should forward their qualifications to the address below. Positions of equal significance exist for engineers with experience in Logic Design . . . Systems Design . . . Mechanical Engineering . . . Microelectronics.

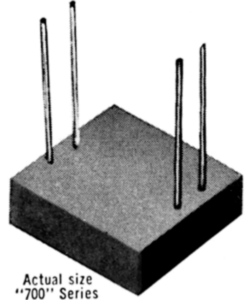
Address your resume to:
Mr. Edwin Barr, Employment Supervisor
HONEYWELL EDP
200 Smith Street, Dept. CD05
Waltham, Massachusetts

Honeywell
ELECTRONIC DATA PROCESSING

Opportunities exist in other Honeywell Divisions. Send resumes to F. E. Laing, Honeywell, Minneapolis, Minnesota 55408. An equal opportunity employer, M&F.

Leedawl COMPASS
Make Your Boy a Leader
Give him a Leedawl Compass for Christmas and let him lead "the boys" through the woods, over a trail or on a tramp.
It's the only Guaranteed Jeweled Compass for $1.00.
If your dealer does not have them, write us for folder C-12.
Taylor Instrument Companies, Rochester, N. Y.
Makers of Scientific Instruments of Superiority.

evenvolt™
complete voltage reference units
in modular component size



Actual size
"700" Series

small ■ printed circuit or chassis mounting ■ simplifies circuit design ■ no temperature compensation needed ■ low price — unlimited variations of voltages, current, temperature coefficient, regulation, output impedance, case and mounting.

NEW!
Fully guarded line operated
DC Current Supplies
Write for Spec. 200-18

For information—write, wire or phone
INSTRULAB, INC.
1205 LAMAR STREET
DAYTON, OHIO 45404

Corrections

Those more familiar with Commodore BASIC than I might know that the brackets around the argument to the SYS instruction are not required. The KERNAL will simply ignore them when parsing the line. Perhaps without the minimum size limitation brought about by the bootloader, there might be a way to save more space in a PRG/ELF hybrid.

As Janus began to take form, I needed to know how many bytes were left that didn't impact my tests. I kept setting all the null bytes (excluding padding for things like integers) to `58` while making sure the functionality was unaffected. This makes them stand out nicely in a hexdump so that I could find the large unused chunks. However, as pointed out by my editors, there was an unintended consequence! All the way down at offset `0x19c` are four bytes of `58` and are labeled as padding to properly align the GNU Multiboot 2 image to 64 bits. The first two of these bytes are also the length of the comment of the PKZip file. It pains me that I missed the opportunity for some added neatness by setting these two bytes back to 00, but the SHA256 hashes have me stuck in a bind.

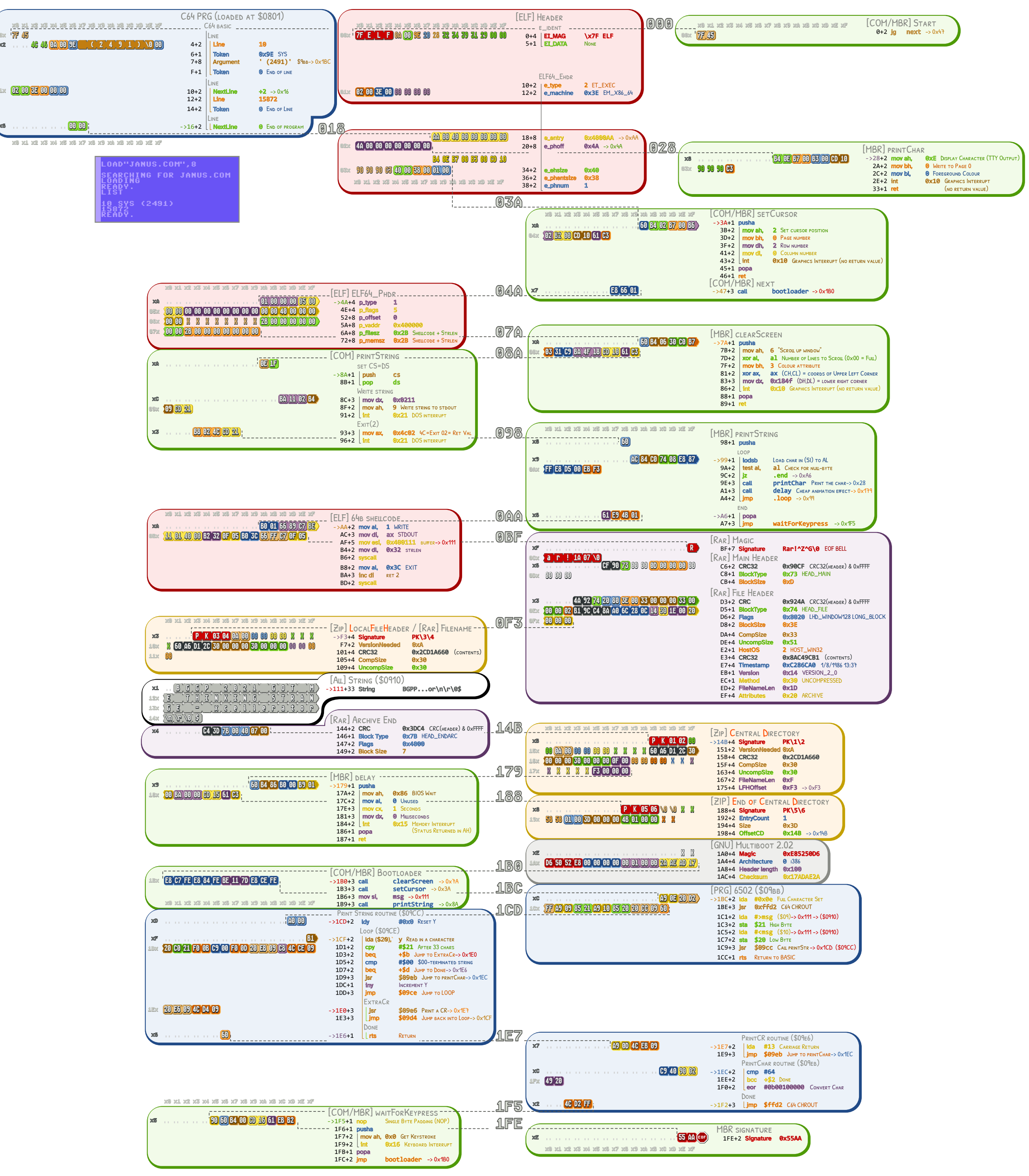


Figure 17: All sides of Janus

22:12 Let's bide our time for a brand new adventure!

from the desk of Pastor Manul Laphroaig,
Tract Association of PoC||GTFO.

Dearest neighbor,

Our scruffy little gang started this самиздат journal a few years back because we didn't much like the academic ones, but also because we wanted to learn new tricks for reverse engineering. We wanted to publish the methods that make exploits and polyglots possible, so that folks could learn from each other. Over the years, we've been blessed with the privilege of editing these tricks, of seeing them early, and of seeing them through to print.

So today, in that spirit of exploration and wonder, I pass around the collection plate and ask you, neither for paper money nor pocket change, but for nifty projects and the clever tricks that make them possible.

Blessed be the hackers who seek to share their knowledge with the world. Maybe share a technical story from the good old days. Maybe share a clever trick from the modern day, such as how to improve TMS320 support in your favorite disassembler or how to reverse engineer a binary format using custom visualizations.

May you be blessed with the generosity to share your runnable source code and buildable hardware schematics, so that others may build upon your work. May you be blessed with the patience to explain how you got to your result, so that others may learn from your experience. Teach me to identify those things that only look intimidating without context, and arm me with the tools to conquer those problems.

Give me these tricks and techniques in an ASCII textfile, or UTF-8 if your language insists, including high resolution figures as separate PNG or PDF files as an email to `pastor@phrack.org`. We've taken submissions hand drawn on napkins, but we'd like to avoid that when possible. My gang and I will clean it up, typeset it in TeX, index it and print it for the world. We'll happily translate from French, Spanish, Portuguese, German, Ukrainian, Hungarian, Hebrew, Serbo-Croation, and Southern Appalachian.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, T∙G∙ S∙B∙