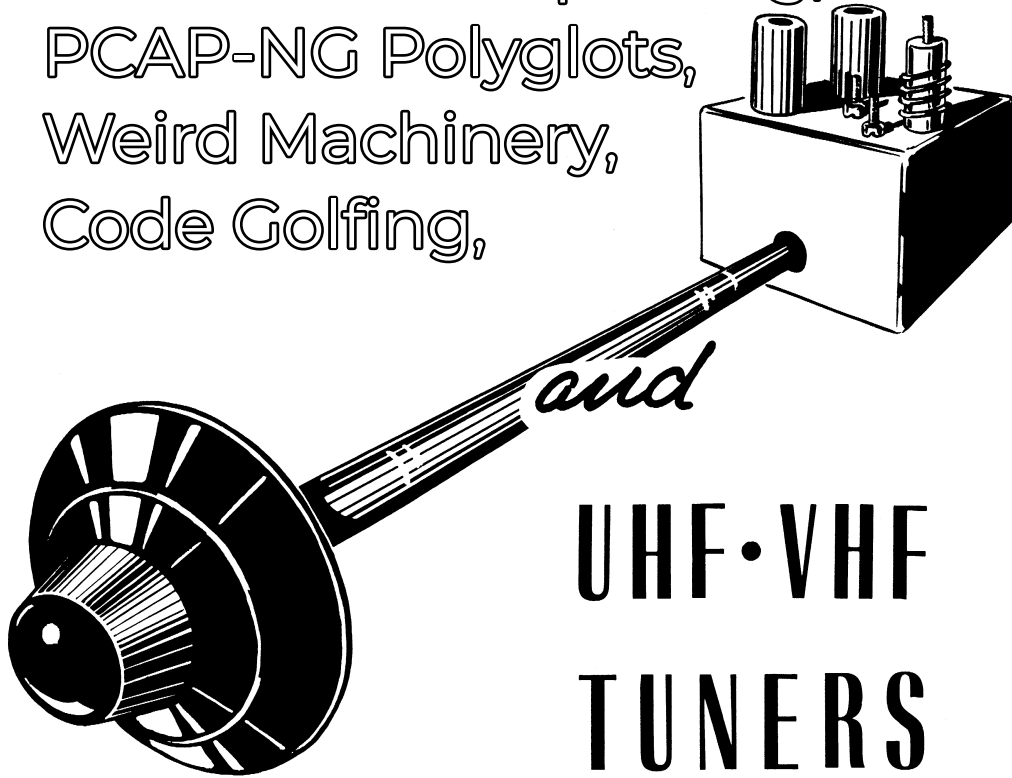


21:02 (p. 6) Glavlit and Samizdat 21:03 (p. 7) Spoofing IP with IPIP 21:04 (p. 8) Antidebugging in Cortex-M
21:05 (p. 11) A Web API for Thumb2 Symbols 21:06 (p. 17) Reversing a Baseball Scoreboard

Notebook

of Altera NIOS Disassembly,
Routable IPIP Spoofing,
PCAP-NG Polyglots,
Weird Machinery,
Code Golfing,



UHF·VHF TUNERS

PoC || GTFEO
LECTURE BUREAU

21:07 (p. 24) Altera NIOS 21:08 (p. 31) An Electromechanical Telephone Exchange 21:09 (p. 42) ELF Palindrome
21:10 (p. 50) A Bootloader Palindrome 21:11 (p. 55) Windrose Fingerprinting of Code Architecture
21:12 (p. 59) NSA's Backdoor of the PX1000-Cr 21:13 (p. 67) Solving the Load Address 21:14 (p. 71) Counting words with a state machine.

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 12, 2022.
€ 0, \$0 USD, \$0 AUD, 0 RSD, 0 SEK, \$50 CAD, 6×10^{29} Pengó, 100 JPC. Это не Госкомиздат.

Legal Note: This magazine is a labor of love, written so that you fine folks might read it and share it. If you have access to a halfway decent laserjet, please make a paper copy for a friend.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo21.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 PCAP-NG packet capture of an experiment by Yannay Livneh. See page 7.

Cover Art: The cover art for this issue is adapted from a 1951 television repair manual by Edward M. Noll, a lecturer at Temple University in Philadelphia.

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, 79 and 80 should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.  
sudo apt-get install pdftjam  
pdftbook --short-edge --vanilla --paper a3paper pocorgtfo21.pdf -o pocorgtfo21-book.pdf
```



Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
T _E Xnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funkmaster of File Formats	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Scooby Bus Driver	Ryan Speers
Stunts (Uncredited)	Alexei Bulazel
with the good assistance of	
Tree Killer	EVM

21:01 Don't give up on your library card!

Neighbors, please join me in reading this twenty-second 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 D.C., Berlin, and Fort Washington, Pennsylvania.

If you are missing the first twenty one 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, or the twenty-first in Leipzig or Washington, D.C. Three collected volumes are available from No Starch Press, wherever fine books are sold.

The old joke goes that Stalin's 1936 Constitution of the USSR guaranteed freedom of speech, but that rat bastard never made any promises about freedom *after* speech. On page 6, our own Pastor Manul Laphroaig takes a step back from the tragedy of so many brilliant works being unpublished or unwritten, to consider another question: If you had the power to censor just the bad stuff, would you use it?

A long time ago in a fancy bar in Tel Aviv, Yannay Livneh told us of a bug in the IPIP tunneling protocol that might allow for convenient injection of an IP frame into a remote network, that this might be nested to create a very complicated route, and that a large number of machines on the Internet were possibly vulnerable. His proof of concept took just a week or two, but the coordinated disclosure dragged on for many months, and the only way you can repay this blood debt is by reading his fine article on page 7.

Suppose that you have some firmware, but you don't want us meddlesome reverse engineers to run that same firmware under a debugger, even after they've defeated the readout protection. On page 8, Balda describes a grab bag of these anti-debugging tricks.

Travis Goodspeed and EVM have spent the past year collecting three hundred gigabytes of microcontroller SDKs, which they've parsed and blinded into a SQL database. Accessible through a JSON API, this database allows such nifty queries as function name recovery and I/O port naming. See page 11.

EVM has also been playing with a baseball scoreboard, the FairPlay 710. On page 17, he describes his method for attaching this ancient artifact to a modern network, allowing parents in his town to display the scores from the comfort of the bleachers.

Our fine journal frequently runs tourist guides to strange CPU architectures. On page 24, Christopher Hewitt introduces us to Altera's original Nios architecture, a soft CPU from the year 2000 that was largely forgotten after Intel's acquisition of the FPGA company in 2015. This particular Nios machine was used in a GPS-disciplined oscillator that Chris wanted to repurpose for other uses.



**ГЛАВНОЕ УПРАВЛЕНИЕ
ПО ОХРАНЕ ВОЕННЫХ
И ГОСУДАРСТВЕННЫХ
ТАЙН В ПЕЧАТИ
при Совете Министров СССР**

„_____“ _____ 1960 г.

№ _____

Москва, Ж-74, Китайский пр., 7
Телефон К 4-46-63

Those of our readers old enough to have used a rotary telephone might have explored the network with a blue box or at least made free payphone calls with a red box. Younger readers might've made their own small telephone networks with VoIP and other newfangled technologies. An anonymous article on page 31 goes beyond those tricks, to show you how a rotary telephone network might be built from scratch with vintage technology.

Netspooky is a damned clever code golpher who is new to these pages. On page 42, you'll find a technique for producing palindrome ELF files. Harvey Phillips also describes his own techniques for machine code palindromes, applied to an X86 boot-loader on page 50. These were both written for 2020's Binary Golf Grand Prix, and we eagerly await what clever things they'll do this year.

Suppose that you have a bit of raw firmware that you're pretty sure is executable code, but you don't yet know the architecture. You might try looking for common sequences, or you might check that relative function calls match entry points. EVM has a simpler method, which is to draw a windrose diagram of byte frequencies, skipping universally common ones like 0x00. Page 55.

In the early eighties, a gizmo called the Text Lite PX-1000 allowed folks to encrypt short messages with DES, then transmit them by audio coupler modem. At some point the NSA got nervous about this, purchased all outstanding units, and convinced the manufacturer to update the ROM to support a unique and proprietary encryption protocol, rather than the standard for which it was made. On page 59, Stefan Marsiske explains how he reverse engineered the backdoored algorithm and cracked it with modern tooling.

It's not so uncommon to find a firmware image, but not a load address. On page 67, EVM describes a generalized solution to this problem, first defining function entry points as a function of the load address and then solving for the load address that matches a strong majority of any absolute calls.

Robert Graham has often lectured our editors on the virtues of state machine implementations of software, as a leaner and meaner alternative to the object oriented monstrosities that might be more organized, but are undeniably more computationally expensive. On page 71, he applies his highfalutin performance optimizations to the wc command.

On page 80, we pass the collection plate, not for bitcoins or wooden nickles, but for nifty stories.



LASER PRINTER with in-built PHOTOCOPIER

RANK XEROX 4045R

- ★ Personal photocopying facility
- ★ 10 ppm fast Laser Printing
- ★ HP LaserJet+ Diablo, Xerox 2700 (FX80/IBM Pro. text)
- ★ 512KB & 1MB RAM versions - RAM memory expandable.
- ★ Resident, cartridge & Downloadable fonts - portrait and landscape.
- ★ Parallel and Serial Interface
- ★ 250 sheet Paper Tray A4
- ★ Cheap toner refills (4 supplied) £10 each, approx 5000 copies.

WARRANTY - 3 months on-site Service/Maintenance by Rank Xerox.

£650	£750
512KB RAM	1MB RAM

NEXT DAY DELIVERY/ INSURANCE £30

TELEPHONE ORDERS,   ACCEPTED

All prices exclude VAT

**Factory Refurbished/Exdemo stock
Current model, normally £3,995 + VAT**

PHONE: 081-330 7533

FAX: 081 - 330 4838

**COMMONSIDE
HARDWARE SERVICES LTD**

**Unit 13, 193 Garth Rd,
Morden, Surrey, SM4 4LZ**

DON'T BUY SOFTWARE THAT'S LOCKED UP!



Beagle Bros
MICRO SOFTWARE

All Beagle Bros Apple Utilities are **BACKUP-ABLE, LISTABLE, CUSTOMIZABLE** and fully compatible with normal Apple DOS.

APPLE MECHANIC

SHAPE WRITER/ZAP UTILITY by Bert Kersey

SHAPE EDITOR: Add professional hi-res animation to your programs. Design shapes & custom type characters, automatically written into shape tables. Many type fonts on disk & listable demo programs showing how to use shape tables for games & impressive hi-res **CHARTS & GRAPHS.** A valuable time-saving utility and Apple learning tool.

BYTE ZAP: A MUST utility. Rewrite any byte on a disk. Optional Hex/Decimal/Ascii display and input. Create illegal file names. Restore deleted files. Inspect, repair and protect disks. Change DOS. Clear illustrated instructions show how data is stored and how to access it.

MORE: A disk PACKED with useful music, text & hi-res tricks for use in **YOUR PROGRAMS.**

\$2950 Includes Apple Tip Book #5 and Peeks & Pokes Chart

UTILITY CITY

21 UTILITIES ON ONE DISK

by Bert Kersey

LIST FORMATTER makes custom listings with page breaks; each statement on new line, if then's called out and loops indented. **MULTI-COLUMN CATALOG** in any page-width. Put invisible commands in programs. Alphabetize & store info. Make trick & invisible file names. Append programs. Convert hex. Dump text to printer. Auto-post Run Number/Date in programs. More: 21 LISTABLE PROGRAMS Total!

\$2950 Includes Apple Tip Book #3 and Peeks & Pokes Chart

ALPHA PLOT

HI-RES GRAPHICS/TEXT UTILITY

by Bert Kersey & Jack Cassidy

HI-RES DRAWING: Create pictures and charts on both hi-res pages; all appendable to **YOUR PROGRAMS.** Relocate any portion of a picture. Compress hi-res; store images in 1/3 DISK SPACE. Superimpose images too.

HI-RES TEXT: Upper/lower case with descenders. **PROPORTIONAL SPACING.** No tab limitations. Adjustable letter height, spacing & color. Multi-directional typing for graphs.

\$3950 Includes Apple Tip Book #4 and Peeks & Pokes Chart

DOS BOSS

DISK COMMAND EDITOR

by Bert Kersey & Jack Cassidy

A classic utility you will ENJOY. Rename commands/error messages. **PROTECT PROGRAMS** (unauthorized save-attempt produces "Not Copyable" message). **LIST-PREVENTION** too. One-key program-run from catalog. Change Disk Volume heading to your title with or without volume number. Fascinating documentation. Hours of good reading & experiments.

All changes may be appended to your programs, so that anyone using your disks (booted or not) will be using DOS the way YOU formatted it.

\$2400 Includes Apple Tip Book #2 and Peeks & Pokes Chart

TIP DISK #1

100 programs from Beagle Bros Tip Books 1, 2, 3 & 4 - Hi-Res/Lo-Res/Text/Sound. All listable, copyable and changeable; each teaches another fascinating Apple programming trick!

\$2000 With Peeks & Pokes Chart

GOTO your Apple Dealer.

Most Apple Dealers carry our software. If yours doesn't, he can have it in his store for you within just a few days through **Beagle Bros** or **Softtel.**



Or Order by Phone:

24-hour **TOLL-FREE** Order Desk:
Visa/MasterCard/COD* Orders, call
Nationwide: **800-854-2003** ext. 827

California: **800-822-1500** ext. 827

Alaska/Hawaii: **800-854-2622** ext. 827

(ORDERS ONLY PLEASE) *COD, add \$3.00

Or Mail us a check:

(or Visa/MC No. & Exp. Date)

- | | |
|-------------------------------------|---|
| <input type="checkbox"/> Alpha Plot | <input type="checkbox"/> Utility City |
| <input type="checkbox"/> Dos Boss | <input type="checkbox"/> Apple Mechanic |
| <input type="checkbox"/> Tip Disk | <input type="checkbox"/> Game Pack 1-4 |

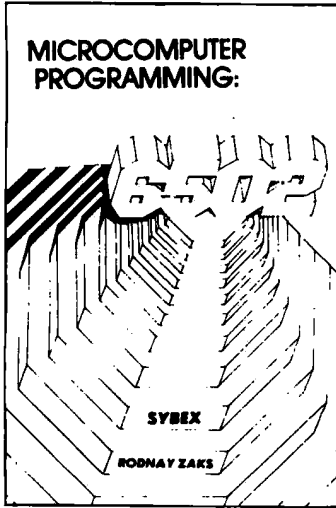
Add \$1.50 shipping (on any size order)

and 6% tax if California. MAIL TO:

BEAGLE BROS, Dept. O
4318 Sierra Vista
San Diego, CA 92103

Please add 44* if outside North America.

LEARN TO PROGRAM WITH THE 6502



● MICROCOMPUTER PROGRAMMING: 6502

By Rodney Zaks, ref C202 **\$9.95**

This text will teach you how to program with the 6502, from ground zero if necessary: arithmetic, input-output, including polling and interrupts, addressing techniques. Completely self-contained, it can be used by the novice to learn programming or by anyone who wants to learn about basic techniques, using the 6502.

(The author has taught programming to more than 1000 persons).

● 6502 APPLICATIONS BOOK

(For SYM and KIM), ref D302 **\$12.95**

A series of practical (hardware and software) applications for a 6502 board (SYM preferred or KIM) which can be used as experiments, or implemented at minimal cost. Examples are: morse generator, electronic piano, digital clock, home alarm system, traffic controller.

● WITH SYM-MICROCOMPUTER BOARD (COMPLETE SELF-STUDY)

C202 + D302 + SYM Board + cassette **\$330**
(shipping add'l)

TO ORDER

- **BY PHONE:** call (415)848-8233
BankAmericard/Mastercharge accepted
- **SHIPPING:** no charge when payment included.
ADD: \$1.50/book for fast shipping.
- **TAX:** in California, add sales tax.
- **OVERSEAS:**
SYBEX-EUROPE, 313 rue Lecourbe,
75015 - PARIS, France Tel: (1)8282502



2020 Milvia St.
Berkeley,
Calif 94704
(Dept B)

NAME _____ POSITION _____
 COMPANY _____
 ADDRESS _____
 CITY _____ STATE/ZIP _____
 C201 C207 C200 C202 D302 Other _____
 Payment enclosed C.O.D.
 ADD \$1.50/BOOK FOR FAST SHIPPING
 charge my Visa Master charge American Express
 Number _____ Exp date _____
 Signature _____ Send catalog.
 FREE CATALOG/ ORDER FORM



21:02 A Tale of Glavlit and Samizdat

by PML

Gather around, neighbors. You’ve all heard the tale of Samizdat, and of course these pages *are* Samizdat. But now it’s time for a darker tale: that of *Glavlit*, Samizdat’s opposite, nemesis, and chief reason for existence. As many things in the history of humanity, it was thought to be a thing of the past—and yet it rises again.

Glavlit (ГЛАВЛИТ) started as a part of the Commissariat of Enlightenment, to prevent printing of harmful things: religious, anti-science, anti-Communist, pornographic, liable to stoke hate or spread rumors, and otherwise misleading¹ or contrary to the public enlightenment as the Commissars saw it.² Glavlit was also charged with removal of previously printed harmful materials from libraries, bookstores, and any other places they could be found. It grew and grew, until it came to report directly to the Party’s Central Committee, and its purview included any posters, note pads, and theater tickets. Nothing could be printed without its approval, and nothing printed could persist without its approval.

This sounds Orwellian, and it sure was, but that’s not my point today. The point is that, despite employing an army of well-educated human censors, Glavlit was *incredibly dumb*. Anyone whose occupation involved printed words knew it, no matter how loyal to Soviet ideology they were. Glavlit would reject the most loyally written books because it creatively imagined some allusion critical of the Soviet system, but would pass things it should have obviously caught, and then confer on them the special quality of suppressed truth by chasing them down.

In short, Glavlit was a disaster. It was the disaster that begat Samizdat.

It turns out that a lot of people who were made to learn to read actually *want* to read—and they want to read stuff that interests them. Scaring them away from reading unapproved things works to a point, until the difference between what’s approved and what’s not stops making sense. At that point, the folks who don’t care whether their words make sense beyond their own career advancement take over the printed word, with Glavlit’s blessing. The folks who care about the actual subject-matter—

the fools, the nerds, and those obsessed with their profession—try and try, and then reinvent Samizdat, as the official print becomes synonymous with hypocrisy.

It really didn’t matter then if the official press had any redeeming value or not. In the end, Soviet society became almost entirely rumor-driven and cultish, despite science being its official highest value. Worse, it erupted in ethnic cleansing wars as soon as it could, despite decades of “peace and friendship between the peoples” lessons dominating the school curricula, the news, the TV, and any other mass medium. Glavlit absolutely succeeded in its mission of control—and it totally failed at every good thing it was meant to achieve. It even failed at pushing the Party line, because the press it shaped was so stupid and boring that it wasn’t even good for propaganda, let alone persuasion.

These days, there seem to be quite a few folks trying to reinvent Glavlit with technology. Wouldn’t it be nice if computers could tell what’s harmful and block it before it could do any harm, or at least unprint it soon afterwards? Lately it’s been looking like the “Web 2.0” is having a giant Glavlit cosplay party. Even nerdy news some days look like there’s been a record-smashing new manga about Glavlit-senpai who is part AI and part superhero.

It might behoove us all to remember how the actual continent-scale 100% expert human baseline effort turned out. As they say, “Play Soviet games, win Soviet prizes.”

There’s an old Soviet joke about a bicycle factory worker who wanted a bicycle but couldn’t afford one. So he started sneaking out parts, one at a time, to assemble them at home. But no matter how hard he tried, he’d get not a bicycle but a machine gun. Well, Glavlit was that kind of a factory. It wouldn’t produce enlightenment no matter how hard they tried.

For what it’s worth, here’s a prediction: there is no such thing as a smart Glavlit, and the larger it gets, the dumber it will be. So we might as well work on building a better Samizdat, neighbors, for it was the past and it will be the future. Amen.

¹Also state secrets, which included any kinds of statistics that contradicted the official Party line. So it goes.

²At the start of the XXth century the idea that the generally uncouth population was there to be forcibly enlightened by their betters was shared by imperial elites and socialists alike. Judging by how the first half of the century worked out, it was not the best of ideas, but the meeting of minds that refined British intellectuals had with Bolsheviks on this was truly touching.

21:03 Spoofing IP with IPIP

by Yannay Livneh

On the Internet, nobody knows you're a dog. Or so they said in 1993. IP, the most fundamental protocol of the Internet, does not enforce or verify the validity of the `source` field specified in the header of an IP packet. Anyone could just send packets spoofing whichever origin address as they liked. It was as easy as executing this Python code. (The `/` operator in the `scapy` package is used to stack the latter layer over the former.)

```
1 from scapy.all import *
  packet = IP(src='13.37.13.37',
3         dst='8.8.8.8')/"some data"
  send(packet)
```

This made a lot of people very angry and been widely regarded as a bad move. So the elders of the Internet, the IETF, sat together in May 2000. They decided to drop packets they deemed fishy, and thus BCP 38 was born.³ This fine document requires ISPs, the moderators of the Internet, to filter packets that originate from their customers with source IPs which were not assigned by the ISP.

Fast forward to 2020: many ISPs implemented this policy and cloud providers followed suit. Nowadays, the average Internet user can't really spoof IP packets. However, some machines in the Internet don't suffer from these policies. So if a user wants to spoof a packet, all they need to do is to ask one of these machines nicely to send a spoofed packet on the user's behalf. How does one ask a friendly machine to send a packet? Just send it over IP and the remote machine will do the rest. To illustrate it with Python code:

```
1 from scapy.all import *
  packet = IP(src='13.37.13.37',
2         dst='8.8.8.8')/"some data"
4 friendly_ip = '1.2.3.4'
  send(IP(dst=friendly_ip, proto='ipip')/packet)
```

And this is it: all you need to do is find such a friendly machine and send it a spoofed packet to send using the somewhat forgotten "IP over IP" protocol (protocol number 4). This protocol was an early implementation for VPN. It's dead simple, just encapsulate another IP in an IP packet and send it. The receiver simply decapsulates the outer packet

and sends the inner IP packet. No authentication, no filters, and no hassles. The Internet has evolved since those naïve days, but operating systems still implement this protocol. And sometimes, if you are lucky, some vendor opens it to the Internet for one reason or another. Surprisingly, this scenario happens quite more often than you might imagine. In fact, this is how I found it. I imagined the bug and then tried to scan the Internet to find such a machine.

This issue has more uses than simply spoofing, and some are worse than others (perhaps the subject for a future article). However, I find one of the uses rather amusing. Packet encapsulation is not limited and can be done multiple times in a recursive manner. The only limitation is the IP packet maximum length which is $2^{16} - 1$. As every IP header size is at least twenty bytes, the limit for IPIP encapsulation is 3,276 layers. This is way more than the classic limitation of maximal network hops (TTL) a packet is allowed to in the IP protocol: 255. So using our new technique, we can craft the longest Pass-The-Parcel game in the history of the Internet. We can craft a single packet that would bounce around for a really long time, way more than you might have expected. I really like this idea.

Scanning code is attached to the PDF of this fine journal.⁴ As for a proof that this technique works? Simply open `pocorgtfo21.pdf` in Wireshark!⁵

IMPORTANT NOTICE

There are thought to be approximately 20 virus programs circulating in the Atari ST community worldwide

Protect your ST with

THE VIRUS DESTRUCTION UTILITY 3.1

ONLY £6.95 INC P&P

Excel Software are the sole U.K. Agents for the above product (Dealer enquiries welcome)

Excel Software also operate a large public domain software library with guaranteed virus free software!

Send a 19p stamp or call us today for our latest catalogue

EXCEL SOFTWARE, PO BOX 159, STOCKPORT SK2 6HN
TELEPHONE: 061-456 9587 (After 6pm)

³BCP38: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing

⁴`unzip pocorgtfo21.pdf zmap-ipip.patch`

⁵`wireshark pocorgtfo21.pdf`

21:04 Anti-debugging tips and tricks for Cortex-M microcontrollers

by Balda

ARM-based microcontrollers are ubiquitous in the so-called *smart* devices we all live around. If you take the time to open these up, you will most certainly find an accessible JTAG interface, or more often now using SWD. As a security-aware person you might say that these interfaces should be disabled at the factory, but they most of the times are not for multiple and often non-relevant reasons like failure analysis and such. As a firmware developer, this interface is also a nightmare as any curious person with the right tools would be able to access the internal secrets held inside the flash memory.

The purpose of this article is to provide fellow firmware developers some ways to detect a debug access from the firmware itself and react to such undesirable intrusion. We also will focus on ARM's Cortex-M family of microcontrollers.

Debugging a Cortex-M core

For nearly all of the Cortex-M cores out there, the most used way to access the debug interface is through the SWD port. The SWD protocol itself is extensively described in ARM's Debug Interface Architecture Specification or ADI, which is freely available from ARM's website. From that document, we know that the interface uses a memory access controller which can read and write to arbitrary locations called the MEM-AP. This means that all subsequent debug operations are performed using memory reads and writes to specific memory-mapped registers.

One of these registers is the Debug Halting Control and Status Register, DHCSR for short. This 32-bit register is located at address 0xE000EDF0 and is used by debuggers to control the core execution state and contains several control bits. Two of them are very interesting: C_HALT[1] halts the core execution, and C_DEBUGEN[0] enables core debug.

To set C_HALT and stop the core, C_DEBUGEN must already be set to 1. This means that a debugger has to perform two writes to this register in order to stop the core. As this register can also be read from the core itself, it is possible to detect if a debugger is trying to connect by looking at the C_DEBUGEN bit value.

```
1 uint8_t detect_debug(void) {
2     uint32_t *DHCSR=(uint32_t *) 0xE000EDF0;
3     if(*DHCSR&1) { // Detect C_DEBUGEN bit
4         // debugger detected
5         return 1;
6     } else {
7         return 0;
8     }
9 }
```

In practice, we used this simple detection method in a CTF challenge by placing the detection routine inside a FreeRTOS thread to clear a secret key from RAM whenever a debug interface tries to connect. If the action is simple enough like in this example, it will complete before the core halts and protect the secret key. Note that this technique cannot be used on Cortex-M0 cores because DHCSR is not reachable from the CPU on this architecture.

Hardware breakpoints

Like their x86 cousins, ARM cores have two kinds of breakpoints: software and hardware. Hardware breakpoints use a dedicated core component, called the BreakPoint Unit (BPU) on Cortex M0 and M1, or the Flash Patch BreakPoint Unit (FPB) on Cortex M3 and later.

The BPU uses a control register BP_CTRL and up to four comparator registers BP_COMPx. If the PC register matches the value of one of the BP_COMP registers and the BPU is enabled, the core will halt the execution. By default, OpenOCD will enable the BPU when connecting to a Cortex-M0 core, it is therefore possible to look for this value in the same way as with the DHCSR register above:

```
1 uint8_t detect_debug(void){
2     uint32_t *BP_CTRL=(uint32_t *) 0xE0002000;
3     if(*BP_CTRL&1) { // detect ENABLE bit
4         // debugger detected
5         return 1;
6     } else {
7         return 0;
8     }
9 }
```


The FPB replaces the BPU and has the same functionality and conveniently uses the same address for its control register `FP_CTRL` as for `BP_CTRL`, so the detection and breakpoint features work the same way. However, there is an added functionality called the Flash Patch, which allows to redirect the execution flow to a different path based on a comparator and a destination address. Instead of breaking when the PC register matches the comparator, the core will update the PC value with the value stored in a remap table located in RAM. The remap table is a pointer array, and if comparator x matches and is enabled, the x th entry of the table replaces the PC value.

In the following example, we use the FPB remap to call the `return_zero()` function instead of `return_one()`. This would produce a valid binary and headaches to any reverse engineer trying to understand what the code does.

```

1  uint8_t return_one(void)
   { return 1; }
3
   uint8_t return_zero(void)
5  { return 0; }
7
   void* FP_REMAP_TABLE[6] = {
9   (void *)&return_zero
   };
11 void setup_fpb(void) {
    // Point FP_REMAP register to our remap table
13  uint32_t * FP_REMAP = (uint32_t *)0xE0002004;
    *FP_REMAP = (uint32_t)FP_REMAP_TABLE;
15
    // Setup the comparator
17  uint32_t * FP_COMP0 = (uint32_t *)0xE0002008;
    uint32_t comp_value = 0;
19  // Set comparison address
    comp_value |= (uint32_t)&return_one;
21  // Enable comparator
    comp_value |= 1;
23  *FP_COMP0 = comp_value;
25
    // Enable FPB unit
    uint32_t * FP_CTRL = (uint32_t *)0xE0002000;
27  *FP_CTRL |= *FP_CTRL | 0b11;
   }
29
   int main(void) {
31   [...]
    setup_fpb();
33
    while (1) {
35     if (return_one()) {
        // This branch will NEVER execute
37     } else {
        // This branch will ALWAYS execute
39     }
41  }

```

Another nice feature of the FPB remap for obfuscation is that OpenOCD resets all FPB registers when connecting to a target. This means that as soon as the debugger is connected to a target running the previous example, the core will execute the first branch instead of the second one, effectively hiding the correct code flow from unauthorized eyes.

Software breakpoints

Software breakpoints halt execution when the CPU executes a `bkpt` instruction which is really useful when debugging your firmware. The instruction also takes a byte-sized parameter to further help the developer manage multiple breakpoints.

An interesting property of the `bkpt` instruction is that if it is executed while the core has no debug enabled, it will generate a `HardFault`. As a developer, we can leverage this property within the firmware and create a dedicated `HardFault` handler. The plan is to detect if the fault happened because of a `bkpt` instruction, restore the registers and resume execution to the next instruction.

Looking at the ARM documentation, we can find that a register contains information about the type of fault that happened. On Cortex-M0, it's the `DFSR` register and on Cortex-M3 and later the register is called the `HFSR` (Hard Fault Status Register). On both of these, a bit is set when the fault occurred because of an untrapped debug event (ie. a `bkpt` instruction with no debugger): the `BKPT[1]` and `DEBUGEVT[31]` respectively.

Now that we know how to detect the debugger, we need to resume execution. Upon entering a fault, some registers are saved on the stack for further analysis. This process is automatically managed by the core, and the following registers are saved (from top to bottom): `r0`, `r1`, `r2`, `r3`, `r12`, `lr`, `pc`, `xPSR`.

When entering the fault handler, the execution context changes to *handler mode*. It is possible to get back into *thread mode* by linking to a special address of `0xFFFFFFF9`, which coincidentally is the value of the link register set when entering the fault handler. Jumping to that address will automatically restore the register values and resume execution.

The only thing left is to increment the saved PC value in the stack by 2 to point to the instruction following the `bkpt` instruction and resume execution. In the following example, we update a global variable containing the detection status.

```

1 uint8_t DEBUGGER_DETECTED = 1;
3 void HardFault_Handler(void) {
4     uint32_t *HFSR=(uint32_t *) 0xE00ED2C;
5     if(HFSR & 0x80000000) { // DEBUGEVT bit
6         // reset detection variable
7         DEBUGGER_DETECTED = 0;
8         asm(
9             "push {r0}\n"
10            "ldr r0, [sp, #28]\n"
11            "add r0, r0, #2\n" // increment saved pc
12            "str r0, [sp, #28]\n"
13            "pop {r0}\n"
14            "bx lr\n" // resume execution
15        );
16    } else {
17        while(1){ // other fault
18        }
19    }
21 int main(void) {
22     while(1) {
23         DEBUGGER_DETECTED = 1;
24         asm("bkpt 8\n");
25         if(DEBUGGER_DETECTED) {
26             // debugger is present
27         } else {
28             // debugger not present
29         }
30     }
31 }

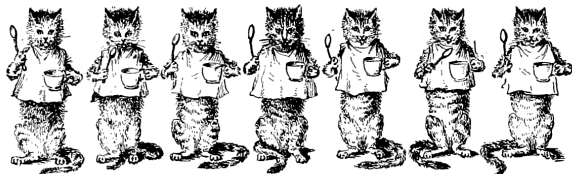
```

Semihosting

Messing with reverse engineers and people trying to debug your firmware isn't enough? Let's take a look at another ARM debugging feature: semihosting.

Semihosting is a way for the target firmware to access data on the debugger side by using syscall-like operations like `open`, `read`, and `write`. It is typically used to allow functions like `printf` to be used in the firmware, with the output being printed in the debugger console on the host. It uses a clever mechanism to work. If the firmware halts on a `bkpt` instruction while being debugged, the debugger will fetch the argument to the `bkpt` instruction. If the argument value is `0xAB`, the debugger will fetch the operation to be performed in `r0`, and the arguments at a location pointed to by `r1`.

The following code implements semihosting to perform a `SYS_WRITE` operation (semihosting call 5) to the host's `stdout`, file descriptor 1.



```

1 void print_semihosting(char * data, size) {
2     /* use SYS_WRITE to STDOUT */
3     uint32_t args[3];
4     args[0] = 1; // FD 1 = STDOUT
5     args[1] = (uint32_t) data;
6     args[2] = size;
7     asm(
8         "mov r0, #5\n" // Op #5 - SYS_WRITE
9         "mov r1, %0\n"
10        "bkpt 0x00AB" : : "r"(args) : "r0", "r1");
11 }

```

The same applies to the other semihosting operations, but one in particular is interesting: `SYS_SYSTEM`. As the name implies, this operation asks the debugger to fetch a command from the target and pass it to the `system()` function on the host. It is therefore possible to use any if the debugging detection routines shown in this article to call this function if a debugger is detected. As a mandatory example, this function will spawn the `xcalc` binary on the debugger host:

```

1 void spawn_calc(void) {
2     const char * cmd = "xcalc";
3     uint32_t args[2];
4     args[1] = (uint32_t) cmd;
5     args[2] = 6;
6     asm(
7         "mov r0, #18\n" // Op #18 - SYS_SYSTEM
8         "mov r1, %0\n"
9         "bkpt 0x00AB" : : "r"(args) : "r0", "r1");
10 }

```

Many variations of this trick exist, and can easily mess with the debugger host. Fortunately, semihosting is not enabled by default in OpenOCD. The command `arm semihosting enable` must be entered in OpenOCD's console to activate semihosting support.

Conclusion

ARM microcontrollers are wonderful devices packing lots of hidden gems like the ones I briefly presented to you today. There surely are more of them hidden deep in the documentation or in an obscure corner of a dumped firmware. I hope that this small introduction will trigger your curiosity and help you find other clever ways to practice firmware self-defence. Code is attached.⁶

⁶git clone <https://github.com/Baldanos/cortex-m-antidebug> || unzip pocorgtfo21.pdf cortex-m-antidebug.zip

21:05 Symgrate: A Web API for Thumb2 Symbol Recovery

by Travis Goodspeed and EVM

Hey folks!

Today we'd like to share with you our nifty little summer project, a publicly accessible server for recovering Thumb2 symbols. You send us the first 18 bytes of a function as a hex-encoded string, and if that function exists in our collection of hundreds of thousands of embedded SDK libraries, we'll give you back the function's name and the library's filename in easily parsed JSON. Client plugins for most interactive disassemblers have already been written.

The first popular symbol recovery tool was IDA Pro's FLIRT technology, which debuted in 1996. FLIRT matches are performed by exact equality between the first 32 bytes of a function, except for those bytes which a linker would relocate. These are stored in a tree structure, to take advantage of overlaps between similar functions to minimize the file size.

FLIRT does handle some of the stranger linking choices of X86, but it does not try to solve the general problem of false positives and collisions that arise from related functions having identical signatures despite different behavior. It won't be putting us reverse engineers out of business anytime soon!

In this article, we'll be implementing a function matcher similar to FLIRT, in that we'll produce blinded signatures of functions which demand that most of the code exactly match, while forgiving differences in the bytes that will be adjusted during linking. We'll implement this both as C code that compares functions within its own address space, and as a PostgreSQL table that can be queried to quickly recover function names.



Thumb2 Instruction Blinding

Before we can build a database, or even compare two functions locally, we need to learn a little bit about the Thumb2 instruction set. We'll do this to create a sort of optimized disassembler, whose only job is to blind out the bytes that don't matter.

Thumb2 is the denser of two instruction sets in 32-bit ARM, and it's the one that's most commonly found in embedded systems. Instructions are either one or two 16-bit instruction words in length; unlike the original Thumb and MIPS16, the 32-bit wide instructions cannot be treated as two independent 16-bit wide instructions.

Thumb2 code uses relative addressing for branches for reasons of efficiency, but it has the nice side effect of making much code accidentally position independent.

This applies to branches and function calls, but not to explicit pointers, such as immediate values. Those are stored in something called a constant pool, which is a group of 32-bit constants that are placed after the `ret` instruction at the end of a function and before the entry point of the next function. These constant pools exist because Thumb2 instructions are too short to include long immediate values, so rather than include these values inside of the instruction, they are referenced by a PC-relative offset to the pool.

So a Thumb2 linker is mostly adjusting (1) relative calls between functions, and (2) absolute addresses held in literal pools at the end of a function. Relative branches within a function aren't adjusted, because they remain the same wherever that function might be loaded. Our basic strategy will be to count from the beginning of a function, enforcing that a minimum number of instructions are *either* identical *or* function calls. And those absolute addresses which might change in the constant pool? Those we don't worry about, because they come late

☆☆☆ SPECIAL OFFER ☆☆☆

CERTIFIED DISKETTES INCLUDING SLEEVES, LABELS AND A 100% LIFETIME GUARANTEE!

QTY.	51/4" 48 TPI DD/DS	31/2" 135 TPI DD/DS
	£	£
25	9.40	22.00
50	17.00	40.00
100	30.00	75.00
1000	280.00	700.00

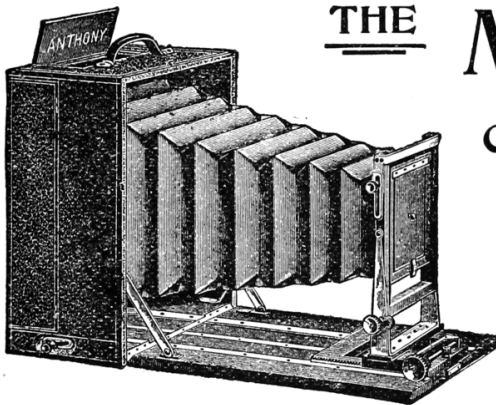
Prices include carriage and VAT
ALL DISKETTES CARRY A NO-QUibble
REPLACEMENT GUARANTEE FOR LIFE!
SAME DAY DESPATCH

Call our order desk
0525 718668
quoting your Access number, or send cheque to:-
WESTONING SOFTWARE,
DEPT NCE, 12 SANDERSON ROAD, WESTONING, BEDFORD, MK45 5JY

**WE ACCEPT
EXPRESS
VOUCHERS**

	hw1																hw2																									
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
General format	1	1	1	1	0												1																									
Branch	1	1	1	1	0	S	offset[21:12]						1	0	J1	1	J2	offset[11:1]																								
Branch with link	1	1	1	1	0	S	offset[21:12]						1	1	J1	1	J2	offset[11:1]																								
Branch with link, change to ARM	1	1	1	1	0	S	offset[21:12]						1	1	J1	0	J2	offset[11:2]				0																				
Reserved	1	1	1	1	0												1	1					0							1												
Conditional branch	1	1	1	1	0	S	cond	offset[17:12]						1	0	J1	0	J2	offset[11:1]																							
Secure Monitor Interrupt	1	1	1	1	0	1	1	1	1	1	1	1	1	1	imm[3:0]	1	0	0	0	imm[11:4]				imm[15:12]																		
Move to status from register	1	1	1	1	0	0	1	1	1	0	0	R	Rn	1	0	S	B	Z	0	field_mask	SBZ																					
Change processor state (imod, M != 00,0)	1	1	1	1	0	0	1	1	1	0	1	0	SBO	1	0	S	B	Z	0	S	B	Z	imod	M	A	I	F	mode														
No operation, hints	1	1	1	1	0	0	1	1	1	0	1	0	SBO	1	0	S	B	Z	0	S	B	Z	0	0	0	hint																
Special control operations	1	1	1	1	0	0	1	1	1	0	1	1	SBO	1	0	S	B	Z	0	SBO	OP		option																			
Branch, Change to Java	1	1	1	1	0	0	1	1	1	1	0	0	Rn	1	0	S	B	Z	0	SBO	SBZ																					
Exception return	1	1	1	1	0	0	1	1	1	1	0	1	(1)(1)(1)(0)	1	0	S	B	Z	0	SBO	imm8																					
Move to register from status	1	1	1	1	0	0	1	1	1	1	R	SBO	1	0	S	B	Z	0	Rd	SBZ																						
RESERVED	1	1	1	1	0	1	1	1	1	1	1	1												1	0	1	0															
Permanently UNDEFINED	1	1	1	1	0	1	1	1	1	1	1	1												1	0	1	0					1	1	1	1							

Subset of Thumb2 instruction formats from ARM DDI 0308D.
 Note well that branches begin with 1111.



THE MARLBOROUGH

Combined { DETECTIVE TRIPOD } Camera

Handsomely Finished in Leather

RISING FRONT SWING FRONT
 REVERSING BACK SWING BACK

"A Perfect Model of Ingenuity"

8x10 \$50.00 | 5x7 \$35.00
 6½x8½ 45.00 | 5x7, with lens and shutter 60.00

SEND FOR ILLUSTRATED BOOKLET

E. & H. T. ANTHONY & CO., = 591 Broadway, New York

in the function, after enough instructions that we'll have a match or not.

The machine language format for branches are described in Section 3.3.6 of the Thumb2 Supplement Reference Manual, ARM DDI 0308D, reproduced here on page 12. J1 and J2 are inverted by the sign-extension bit for reasons of backward compatibility, being 1 when the branch is short. From this table, we can see that a Branch with Link (b1) instruction always begins with F in its first word, and will also begin with an F in its second word except when the target is very far away.

Comparing Instructions from C

If you need to quickly compare short functions for similarity in C, where `src16()` and `dst16` grab 16-bit words from your source and destination address spaces, you might do something like the following, counting 16-bit words which are either exactly equal or are short branches.

```
2  //! How similar are two functions?
3  int scorematch(int sadr, int dadr){
4      int i=0;
5
6      // Comparing half-words.
7      do{
8          i+=2;
9      }while(
10         (
11             //Halfwords exactly agree
12             src16(sadr+i)==dst16(dadr+i)
13
14             //or halfwords might both be a BL.
15             || ( (src16(sadr+i)&0xF000)==0xF000 &&
16                 (dst16(dadr+i)&0xF000)==0xF000 )
17         )
18         //stop after a while.
19         && i<1024
20     );
21     return i;
22 }
```

This scruffy little example is missing range checks, and it will fail to recognize the second word of longer branches, when J1 or J2 might be zero, but it is brutally effective at moving symbols between minor revisions of small firmware images.

We used this code, complete with an embarrassing bug or two, in the MD380Tools project for years. See PoC||GTFO 10:8 and 13:5.

Comparing Instructions from SQL

Having some C code that can quickly compare one function to another is great for porting symbols from one executable to another, but we'd rather have a giant database of functions, on a central server, than any friend or stranger can query freely when useful. For this, we needed to convert our rag tag algorithm into one that was just as scruffy, but could be expressed in terms of SQL for convenient querying.

We decided to implement this as a string that is wildcarded in the style of a SQL LIKE clause, ASCII-armed and with two underscores (__) to replace any byte which might be changed by the linker.

Our table schema is roughly like this,

```
1  drop table if exists functions;
2  create table functions(
3      id serial primary key,
4      arch varchar(10) not null,
5      — C++ names can be very long.
6      name varchar(2048) not null,
7      filename varchar(2048) not null,
8      raw varchar(100) not null,
9      blinded varchar(100) not null,
10     unique(blinded) —saves pruning later
11 );
```

SQL Optimizations

When this scheme was mentioned in passing to a mainframe old-timer by the name of Jim, he panicked! “Why in hell are you traversing every table on every query?” We're not, of course, as that would be much too slow.

The naïve version of this, the one that scared Jim so much, is easily read but even after indexing it is rather slow.

```
1  — 70ms. Slow and naive, but easy to read.
2  select name from functions
3  where $1 like blinded;
```

If we ask Postgres to `explain analyze` this query, it takes nearly 70ms because every row of our functions table must be scanned directly, and even split into parallel threads that's a lot of overhead. As our database grows, the overhead will only get worse.

We can speed things up a little more by doing the barest minimum of parsing on the start of the string. See how 10b50446 (0xb510 0x4604) does not begin with an f, and is not a branch function that might be wild-carded in our database by the

```

2 % curl -X POST -d 8000beef=02780b78012a28bf9a42f5d16de9044540ea \
3   https://symgrate.com/jfns | jq
4 {
5   "8000beef": {
6     "Name": "strcmp",
7     "Filename": "iccv9cortex/GnuARM/arm-none-eabi/lib/thumb/v7e-m+dp/hard/libg.a"
8   }
9 }

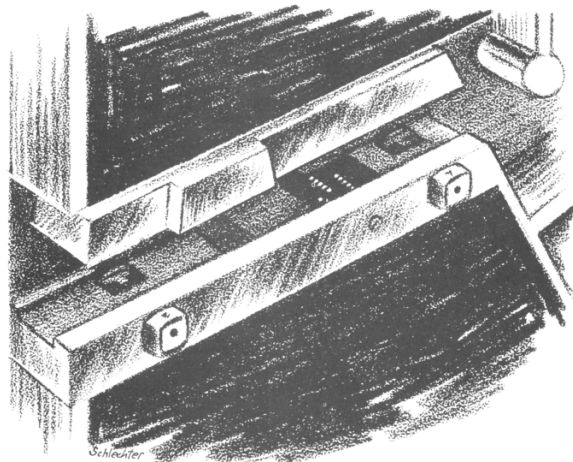
```

instruction format on page 12? We can add a little piece to the **where** clause, such that the first eight characters must *exactly* match our unknown function. With this addition, the **like** operation will only be calculated against a small subset of the total table.

```

1 — 7ms when first two are not branches.
2 select name from functions where
3   ( substr($1,3,1)='f'
4     or substr($1,7,1)='f'
5     or substr($1,1,8)=substr(blinded,1,8)
6   ) and $1 like blinded;

```



300 Gigs of Object Files

The one big shortcoming of this technique is that while it is rather robust against changes made by the linker, it is terribly fragile to changes in compiler optimization.

We counter this by recognizing that much device firmware is compiled from static libraries distributed with compiler toolchains as part of an Integrated Development Environment (IDE) from the chip vendor. We've taken to collecting every one of these we can publicly find online, buying the really old ones on eBay.

All told, we're now well above 300GB of `.a`, `.o` and other object files, which are crunched into a SQL table by a bunch of Binary Ninja scripts running in parallel. While Binja offers excellent scripting support that is a joy to use, we're ashamed to admit that we use it here only as a glorified ELF parser, to quickly give us the function prefixes and names.

All told, we have a couple hundred different IDE versions that supply us with 41 unique fingerprints for `strcpy`, 43 for `strlen`, 134 for `strncpy` and 50 for `sprintf`.

Clients and Server

Our server is written in Golang, presenting a few simple API pages that return `json` describing every function that matches our collection. For those in a hurry, results can also be requested in ASCII.

There's some overhead to the HTTPS connection, and some overhead to the searching, so we recommend sending requests in batches of a hundred or so functions.

Let's walk through how the IDA script works, using a fragment in Figure 1. We're going to iterate over the whole program on every function that IDA found in auto-analysis. We'll either grab the boundaries of the `.text` section (if we're in an ELF) with `ida_segment.get_segm_by_name` or we'll just start at memory address 0, get the next function with `idc.get_next_func(0)` (which will always be the first function in the binary), and work forward to the end of the binary.

The script calls `ida_getfunctionprefix`, a little helper function we wrote to grab the first bytes of the function used as the Symgrate signature, which is currently 18 bytes. We add (`address`, `function bytes`) pairs to our query string up to 64 times. This allows us to query 64 signatures at a time, with

```

# Iterate over all the functions, querying from the database and printing them.
2 fnhandled=0;

4 qstr="";

6 start=0
end=0
8 t = ida_segment.get_segm_by_name(".text")
if (t and t.start_ea != ida_idaapi.BADADDR):
10     start = t.start_ea
    end = t.end_ea
12 else:
    start = idc.get_next_func(0)
14     end = ida_idaapi.BADADDR

16 f=start

18 while (f != ida_idaapi.BADADDR) and (f <= end):
    iname=idc.get_func_name(f)
20     adr=f
    adrstr="%x"%f
22     res=None

24     bstr = ida_functionprefix(f)
    # We query the server in batches of 64 functions to reduce HTTP overhead.
26     qstr+="%s-%s&"%(adrstr, bstr)
    f = idc.get_next_func(f)

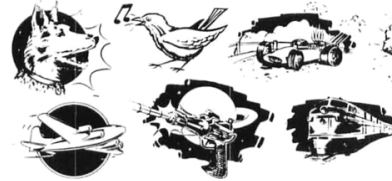
28

30     if fnhandled&0x3F==0 or f is None:
        res=Symgrate2.queryjfns(qstr)
        qstr=""
32         if res!=None:
            Symgrate2.jprint(res)
34             #optionally rename functions to the values found in the query
            #ida_renamefunctions(res)


36     fnhandled+=1

```

Figure 1: Fragment of Symgrate2Query.py from the IDA Pro plugin.

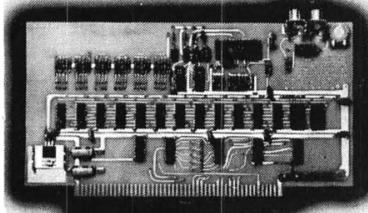



**New and Unusual SOUNDS
for your Computer \$149.95**



The Microsunder is an S-100 compatible sound generating card that can be programmed in BASIC or assembly language. Three to five lines of code generates such sounds as: organ music, sirens, phasers, shotguns, explosions, trains, bird calls, helicopters, race cars, airplanes, machine guns, barking dogs, and many thousands more. Only a few minutes of time is needed to patch the sound code into existing programs.

The Microsunder is assembled and tested, and comes complete with sample code, two game programs, and two utility programs for creating almost any sound.

BOOTSTRAP ENTERPRISES INC.
100 North Central Expwy., Richardson, TX 75080
(214) 238-9262

Name _____
Address _____
City _____ State _____ Zip _____

Add \$4.95 for Postage & Handling
 Check Enclosed Texas Residents add 5% Sales Tax
 VISA # _____
 MASTERCARD # _____
Exp. Date _____

much less network overhead than making a separate HTTPS transaction for each function.

Once we get to 64 functions we submit the query with `Symgrate2.queryjfns`. This function converts marshals the query string into a JSON object and submits the JSON object over HTTP to the server. The server returns (`address`, `function name`) pairs in a JSON object. By default, the script prints the pairs, but there is also a line that if uncommented will rename functions to the names found by Symgrate. We find it's usually better to see what kind of results you get back first before committing the names to your database.

A Database of SVDs

By this point, we hope to have already convinced you of the value that a Web API server can have for firmware reverse engineering. If you can use our server to recover missing symbols from a firmware image, why not query it for other useful things?

Among our collection of object files, we also have nearly twenty thousand `.svd` files. Each `.svd` file contains an XML description of a Device, the base address of each I/O Peripheral, and the offset after that base address for each I/O Register in that Peripheral.

Querying our server quickly gives all of the registers for the STM32F407. (There's no support for hexadecimal numbers before JSON5; please forgive us if that makes your eyes bleed.)

```
1 % curl -d STM32F407=STM32F407 -X POST \
  https://symgrate.com/jsvd | jq
3 {
5   "STM32F407": [
7     {
9       "PeripheralName": "TIM2",
10      "Name": "CR1",
11      "Adr": 1073741824
12    },
13    {
14      "PeripheralName": "TIM2",
15      "Name": "CR2",
16      "Adr": 1073741828
17    },
18    {
19      "PeripheralName": "TIM2",
20      "Name": "SMCR",
21      "Adr": 1073741832
22    }
23  ],
24 }
```

But what if you don't yet know that you are using an STM32F407? Simply send a list of the registers and their access mode, or `u` for undefined, to the server, and it'll give you a list of potentially compatible chips.

```
1 % curl -d 0x58000148=u -X POST \
  https://symgrate.com/jregs | jq
3 [
5   {
6     "Name": "STM32WBxx_CM4",
7     "Count": 1
8   },
9   {
10    "Name": "STM32MP1_v0r3",
11    "Count": 1
12  },
13 ]
```

We hope to have convinced you find folks that these new-fangled Web APIs are useful, not just for dancing babies and hamster dances, but also to expose valuable databases to otherwise slim plugins of reverse engineering frameworks. Expect some new database functions in the near future, and kindly buy us a beer if the database gives you some useful results.

**Yes, thanks,
I'm quite well.**

"Wouldn't know me? Well, I hardly know myself when I realize the superb comfort of well-balanced nerves and perfect health."

"The change began when I quit coffee and tea, and started drinking

POSTUM

"I don't give a rap about the theories; the comfortable, healthy facts are sufficient."

"There's a Reason" for Postum

Postum Cereal Company, Limited,
Battle Creek, Mich., U.S.A.

Canadian Postum Cereal Co., Ltd.
Windsor, Ontario, Canada



21:06 Reversing the Fairplay 710 Baseball Scoreboard

by EVM

The local baseball league where my kids play has some old electronic FairPlay 710 scoreboards that needed rehabilitation. FairPlay is a line of scoreboards made since 1975 by the Fairtron Corporation, which is still around in some fashion. The boards in our league date to 1992 and have become disused because of the way they were originally wired. At a league meeting over the summer, somebody asked what it would take to make them WiFi controlled. In this article, I'll walk through my RE process and my WiFi controller implementation, in the off chance that any of you fine neighbors want to rig up something similar.

At installation the boards were wired up to 110V AC power and a low voltage signal line. The processor box inside the board takes a 1/4 inch audio cable, and controls the bulbs. The board is comprised of standard E26/A19 bulb sockets (for the Ball/Strike/Out/Hit/Error lights) and E26/A15 bulb sockets (for all of the digit displays). The signal line is usually terminated indoors with a corresponding audio jack. The controller box can then be plugged into the jack to control the board.



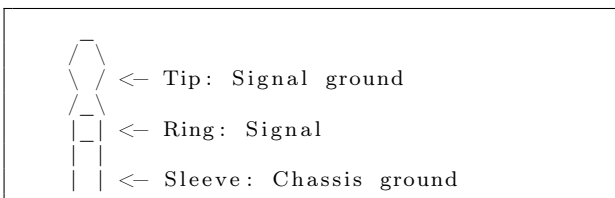
On the two primary fields of play these jacks were put in the top level of a snack shed—a sort of score-keeping booth. The problem is that no parent wants to be banished to the booth, so the boards don't get used. We wanted to make it so that parents could easily operate the boards from their phone, sitting comfortably in the bleachers.

The Controller

Since it was going to be logistically difficult to haul an oscilloscope out to the field, I decided to attack the controller box. After popping open the case I saw a beautiful little old lady of a board, featuring a Motorola 68HC11 and a 128K EPROM. Normally I would be all about popping that EPROM into a reader and dropping the image into a disassembler, but I figured that would be a long path to getting results, since the signal was probably pretty straightforward. And like a batting practice fastball, it sure was.

Stealing Signals (like an Astro)

I could easily see the red (signal) wire hooked up to the “ring” part of the 1/4 inch audio jack, and the black (ground) wire hooked up to the “tip” part. When I clipped my oscilloscope probes onto these parts of the connector, I could immediately see the data pulse train output by the controller and the encoding was very clear. (Figure 5.)



The FairPlay signal uses RS-232 signal levels ($\pm 5V$), but uses a proprietary protocol. In RS-232, each bit takes the same amount of time, with a 1 being a logical high (+5V) and a 0 being a logical low (-5V). The length of each bit is determined by chosen baud rate. In the FairPlay protocol each symbol contains both a high part and low part, and the difference between a 1 and a 0 is the length of the high part. Each symbol is 30 microseconds long,



Figure 2: FairPlay 710 scoreboard and internal processor box.

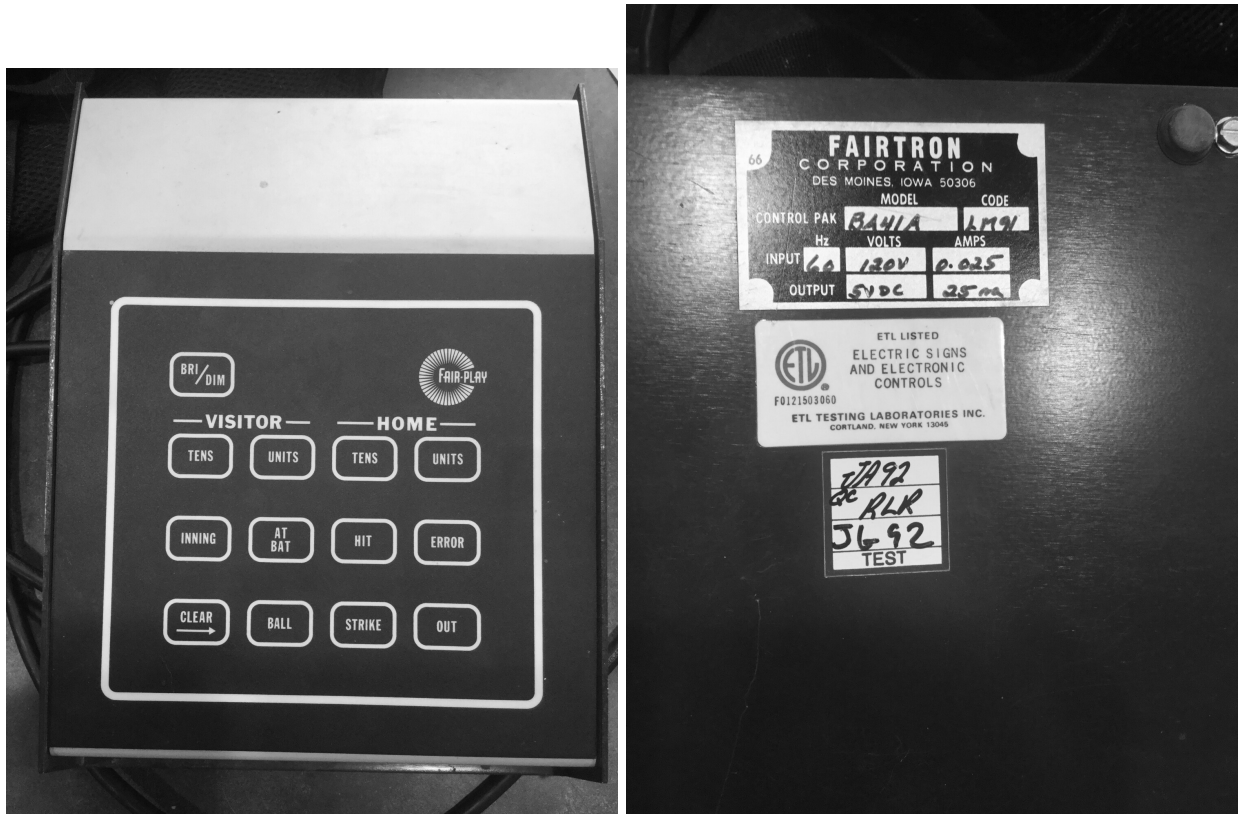


Figure 3: FairPlay controller buttons and label.

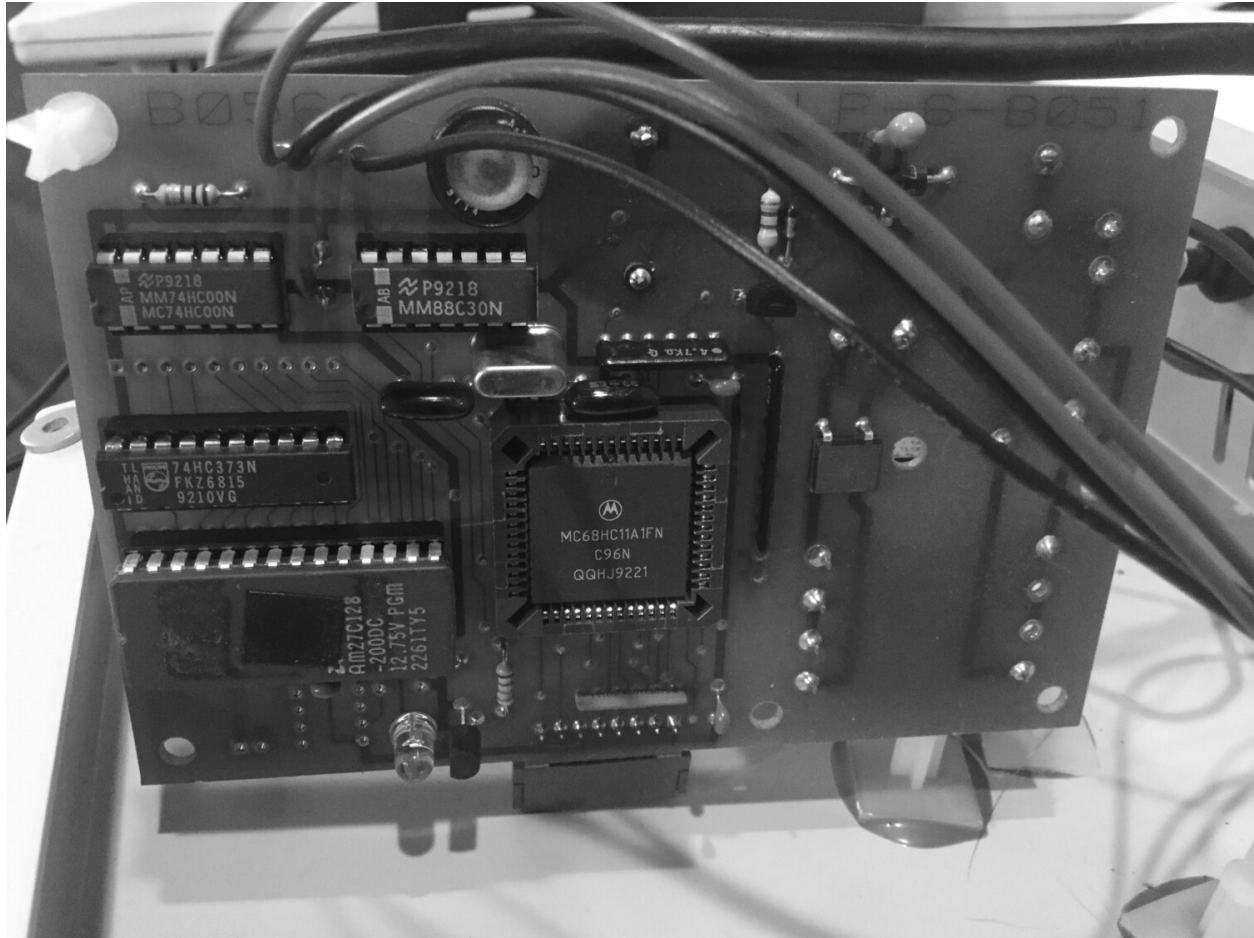
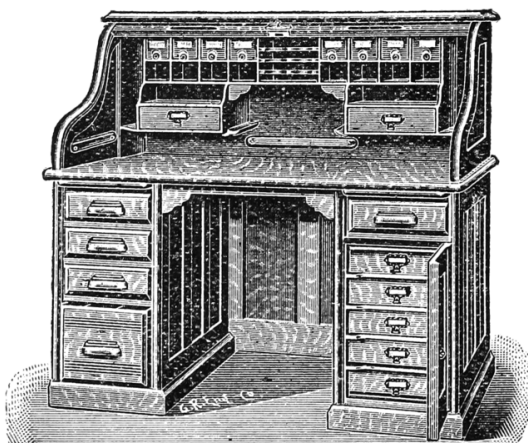


Figure 4: FairPlay BA41A controller board.



MACEY'S No. 10

is the Best high roll top Desk in the World for the Money—none excepted

OUR LIBERAL OFFER.—We will ship you this Desk—you can examine it CAREFULLY—make a note of its many points of merit, go to your furniture dealer, look at the best Desk HE will sell you for the SAME PRICE, and if you are not fully satisfied that "Macey's No. 10" is the best Desk for the price you were ever offered by ANY dealer, at ANY time, for ANY purpose, you may return the Desk, at OUR expense, and we will send your MONEY BACK.

Size, 54 in. long; 33 in. wide; 51 in. high. Quarter-sawed Oak. Antique Finish. Piano Polish. Strictly high grade throughout. Sectional Construction, permitting the Desk to be taken through the narrowest doorways.

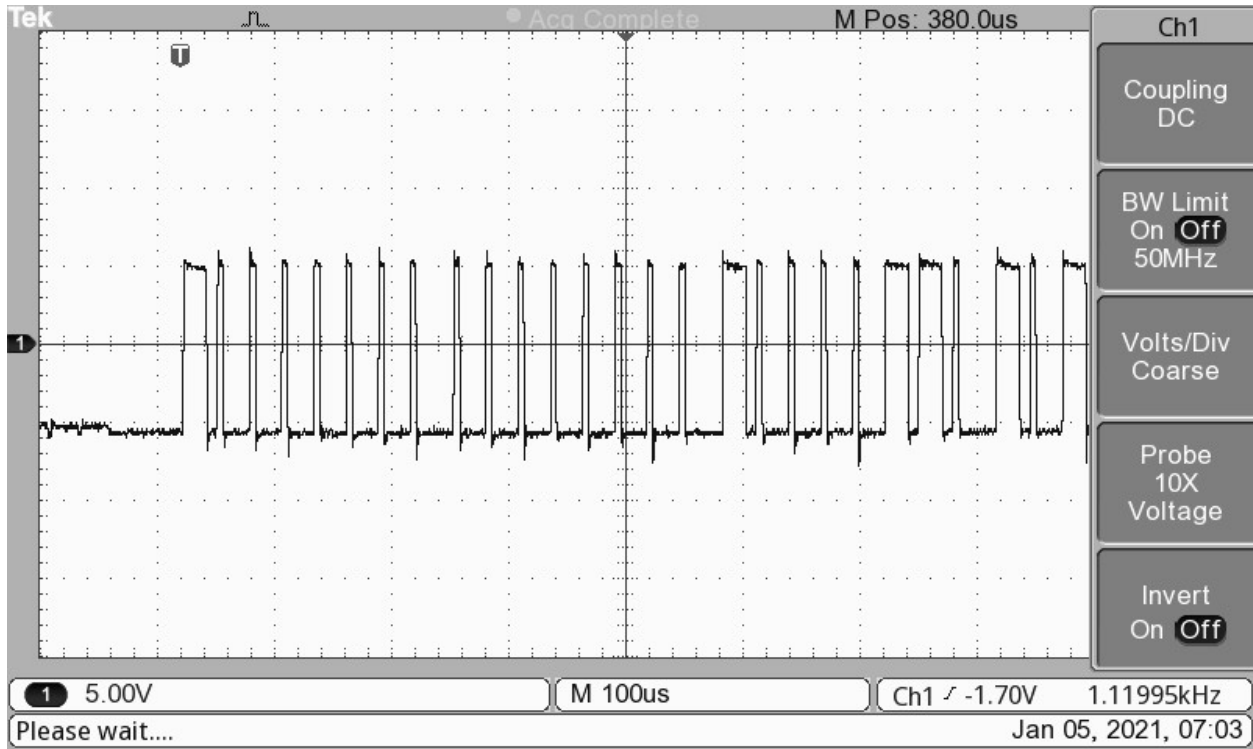


Figure 5: Scope capture of FairPlay signal.

a 0 symbol is 5 microseconds high and 25 microseconds low, a 1 symbol is 20 microseconds high and 10 microseconds low. The messages go from controller to board, there is no path for a response from the scoreboard.

This particular model uses a 56 bit message word that is repeated every 50 milliseconds. (See Figure 6.) I determined the fields by pressing controller buttons while it was hooked up to the oscilloscope and watching which bits change. For the digit encoding I cycled through all possibilities once I had a proof-of-concept implementation running on a Raspberry Pi. See Figure 7 for an explanation of the bit-fields in the message. Notice that this is how it is transmitted on the wire.

Overkill: The Correct Amount of Kill

You might be doing the math in your head and thinking that there are precisely zero things that happen in a baseball game that require 50ms timing in a scoreboard. But I think it's likely that this same protocol is used in FairPlay scoreboards for sports like basketball or hockey that have a game clock. (Such a clock needs to be accurate to tenths of seconds.) My guess is that other FairPlay boards of similar vintage for other sports probably use the same encoding and timing, with different message words.

You might expect a protocol like this to have the controller transmit numerical values and then the scoreboard would figure out which bulbs to turn on, but it doesn't work that way. For the Ball, Strike, and Out fields, each bulb directly maps to bits in the message. The score and inning digits are controlled by a single byte in the message, but each digit is made up of 13 bulbs. This means not every bulb can be directly controlled. Nor does it work like a seven-segment display.



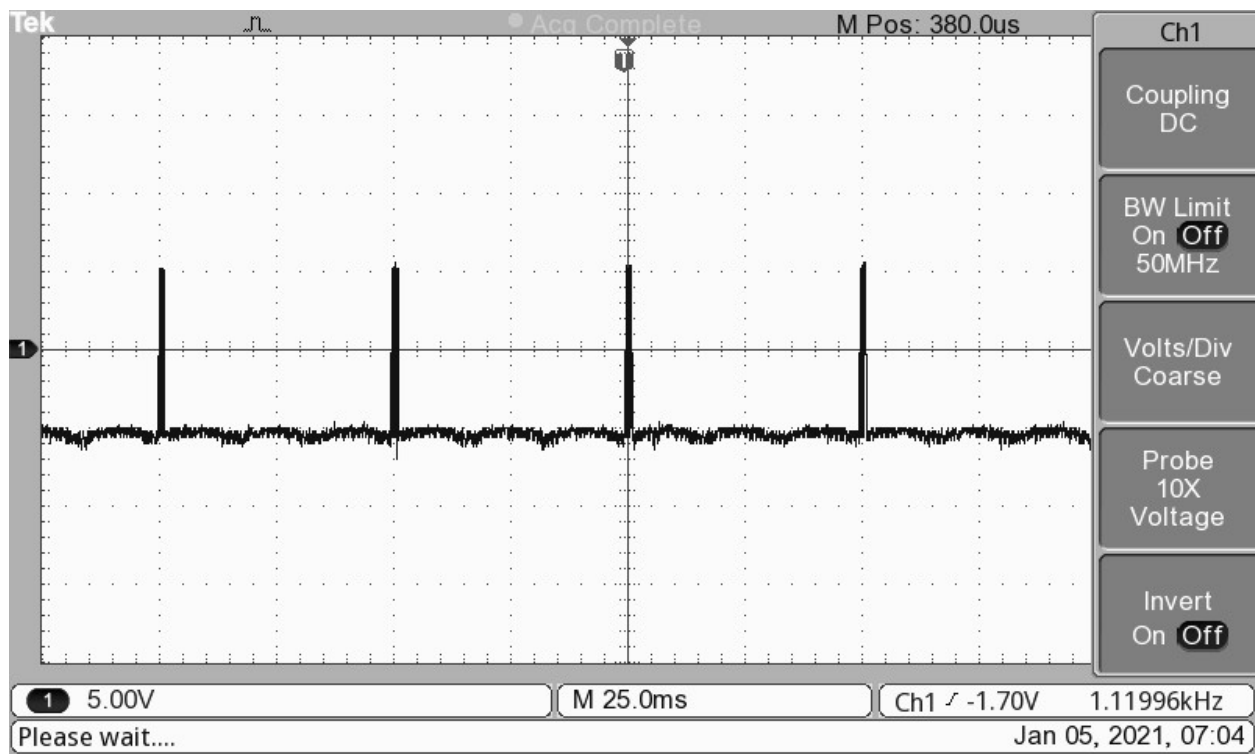


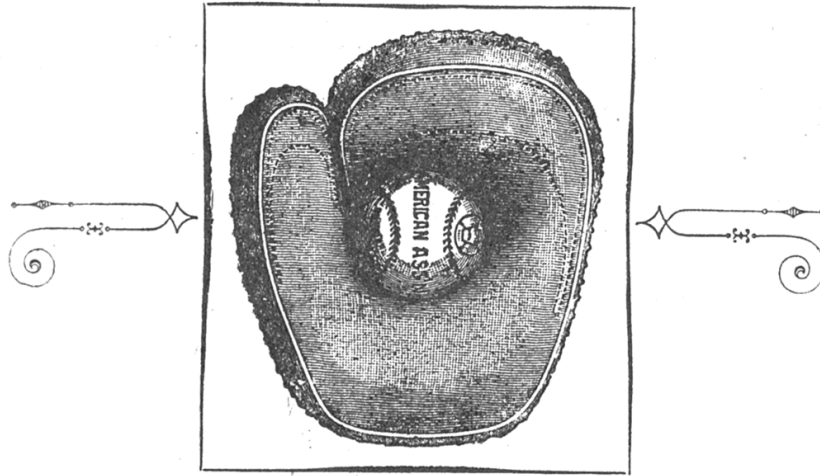
Figure 6: Repeating messages in FairPlay protocol.

Byte #	Bitfield	Key
0	br x x x e it(3)	br – bright (1) / dim (0)
3 1	h o o st st b b b	h – hit e – error
2	inning ones digit	it – inning tens digit (3 bits)
5 3	home ones digit	x – unused
4	home tens digit	o – out
7 5	guest ones digit	st – strike
6	guest tens digit	b – ball

Figure 7: Bitfields in the FairPlay protocol.

REACH'S BASE * BALL * GOODS.

The **Reach American Association Ball** is the best made and **guaranteed** to give satisfaction.



The **Reach Special Catchers' Mitts** used by all **League Catchers**, made in either **Buckskin** or **Calfskin**, with **Patent Lacing** and **Deep Pocket**.

*Reach's
Special*

PRICE,
\$7.50
EACH.

OTHER GRADES DOWN TO 25c. EACH.

We also make the famous **Irwin** line of **Catchers' Mitts** and **Fielders' Gloves**.

A. J. REACH CO.,
Tulip and Palmer Sts., PHILADELPHIA, PA.

The bulbs map to the 8 bits of the byte in the following format:

	007
2	5 1
	567
4	4 2
	337

For instance you can render the digit 3 in two ways, with either the pattern 0x4F 0xCF.

1	XX	XXX
	X	X
3	X	or like this: XX
	X	X
5	XX	XXX

Here are the mappings for every digit:

```
1 unsigned char pattern [] = {
2 // 0 1 2 3 4
3 0xBF, 0x86, 0xDB, 0x4F, 0xE6,
4 // 5 6 7 8 9
5 0xED, 0xFD, 0x87, 0xFF, 0xEF
};
```

To fully implement the WiFi control, I hooked up a Raspberry Pi Zero to the new Pi Pico board via UART and then I have a Pi Pico GPIO output hooked up to a MAX3232. (Thanks to good neighbor Goodspeed for that tip.) I have the Pi serve up a pretty simple PHP script that writes the current settings to a file, and a little server program that converts these settings into the proper 56-bit message word. The Pico program just reads the current 56-bit message and generates the signal which is converted to $\pm 5V$ by the MAX3232. Code is available, of course.⁷

Educate Your Child at Home

Under the direction of
CALVERT SCHOOL, Inc.
(Established 1897)

A unique system by means of which children from kindergarten to 12 years of age may be educated entirely at home by the best modern methods and under the guidance and supervision of a school with a national reputation for training young children. For information write, stating age of child, to

THE CALVERT SCHOOL, 14 Chase St., Baltimore, Md.
V. M. HILLYER, A.B. (Harvard), Headmaster.



*Why there must be an Official Ball, and
Why the Spalding Ball is Official*

Every condition as to a base ball must be uniform if match games are to be played and compared with accuracy and justice. The Spalding Official National League Cork Center Base Ball establishes uniformity around the world. The length of base lines is fixed, the limitations of the bat are regulated, and the Spalding Base Balls, in weight, circumference and materials, are perfect. When A. G. Spalding submitted to the governing body of base ball the first sample of a uniform base ball that had been devised, he made it possible that, no matter where a game might be played—Maine or California—all players would know their achievements would have a value because the Spalding Base Ball was standard for batting and fielding. From the first, the Spalding Base Ball was one of the most vital factors in spreading active base ball around the world. The wiry Filipino, the agile Japanese, the athletic Australian and the "Doughboy" on the Rhine play with the Spalding Official National League Base Ball, knowing it is just like the base ball which is being used by the "big leaguers" on the Polo Grounds.

**\$100
A WEEK**

CLOTHES THE FAMILY

Summer Clothing for You

We will sell you a complete outfit on the kind of terms that will suit you best.

We Guarantee Every Garment Sold

Our up-to-date styles, right from our factory, are sold to you at lower prices than other stores can possibly sell them.

Complete Outfits for Men and Women

Low, Plain Prices. Everything new for you.

FARLEY & ASKIN
16 N. Pack Square

⁷git clone <https://github.com/evm-apl/FairPlay> || unzip pocorgtfo21.pdf FairPlay.zip

21:07 A Tourist's Guide to Altera NIOS

by Christopher Hewitt

Sziasztok, szomszédok!

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 Nios family of embedded soft processors as quickly as possible, with a minimum of fuss and formality.

Those of you who have already worked with Nios might find this to be a useful reference, while those of you new to the architecture will find that it isn't really all that strange. If you've already reverse engineered binaries for any platform, even x86, I hope that you'll soon feel right at home.

We've written this guide to broadly cover various configurations of Nios processors. These processors are generally implemented in configuration bitstreams for various Altera programmable logic devices or system on programmable chips, but may also be found in custom silicon. A minimalist configuration of Nios might be used to execute a simple control sequence out of ROM, while a complex design might make use of several fully-featured Nios processors and external memory to process a complex workload.⁸ Even though Nios was quickly superseded by a complete redesign, the architecture can still be found in the occasional embedded system. Readers interested in the newer Nios II family of processors may find significant differences in the original Nios architecture and may benefit from a different introduction.



⁸unzip pocorgtfo21.pdf phrack6317.txt # Phrack 63:17 by Cawan

⁹unzip pocorgtfo21.pdf packratgps.pdf # Packrat GPS by WA2OMY and WA3YUE

Some Historical Context

Altera introduced Nios in June of 2000 as a reconfigurable embedded design platform tailored to the company's FPGA product offerings. Building from its commercial success, Altera was quick to develop and release a successor, a 32-bit redesign called Nios II, by 2003. Having vastly improved performance and resource utilization over the original Nios platform, Altera deprecated Nios and urged developers to migrate to the new platform. After Intel acquired Altera in 2015, it became particularly difficult to find Nios-related design resources as Altera's website eventually went offline causing most references to seemingly vanish. Without having encountered a device developed during this narrow window of time it's easy to have missed out on ever seeing this architecture, though there are still some traces of Nios in the wild.

An Unexpected Rediscovery

GPS disciplined oscillators are a great way to provide a stable frequency-locked reference for test and measurement equipment found on electronics workbenches, but commercial products can be out of reach for the hobbyist on a tight budget. Fortunately, amateur radio operators have already solved this problem by repurposing the TruePosition LMU300, a nifty piece of telecommunications equipment recently decommissioned in bulk.⁹ These devices were originally installed to provide caller location to emergency services in North America in accordance with federal E911 mandates. Each rack-mount unit contains a separate smaller board containing a GPS receiver and a disciplined 10 MHz reference output, which can be operated independently with some modifications.

Ordinarily, the board's GPS function is initialized by another component within the chassis sending a \$PROCEED command via RS-232. Without this command, the firmware is stuck in a loop constantly transmitting its firmware version number and device serial number. A common workaround is to have an external device send this command to the board automatically when powered on, but it's preferable for

Introducing the Excalibur Development Kit Featuring Nios



The Development Kit that Gets You on the Cutting Edge

The Nios™ soft core embedded processor, the first of Altera's new Excalibur™ embedded processor solutions to ship, delivers just what you need to create system-on-a-programmable-chip (SOPC) designs.



EXCALIBUR™

This new flexible embedded processor solution offers a 32-bit configuration, up to 50 MIPS performance, and an equivalent volume price point of \$5. The development kit is available now with everything you need to get started.

A Complete Solution for Only \$995

This Excalibur Development Kit contains:

- Nios Configurable RISC Embedded Processor Core and Peripherals
- Quartus™ Programmable Logic Development Software
- GNUPro® C/C++ Compiler and Debugger from Cygnus®, a Red Hat® Company
- ByteBlasterMV™ Download Cable
- Development Board Including an APEX™ EP20K200E Device
- Reference Design and Documentation

Free Hands-on Workshops

Intensive three-hour workshops, starting in June, will teach you how to create an SOPC design using the Nios soft core embedded processor in an APEX device. You will develop and compile C code, then execute and troubleshoot it on the development board. You will also learn about the GNUPro Compiler and Debugger from Cygnus, a Red Hat company, included in the Excalibur Development Kit.

Register Now!

To reserve your space at the FREE Excalibur workshop nearest you, or to find out more about this revolutionary development system, visit Altera's web site at <http://www.altera.com/workshop>.

Win a Free Excalibur Development Kit!

Each workshop will feature a drawing for a free Excalibur Development Kit. You must be present to win, so sign up today.

ALTERA®

The Programmable Solutions Company®

Copyright © 2000 Altera Corporation. Altera, APEX, APEX 20K, APEX 20KE, ByteBlasterMV, Excalibur, Nios, Quartus, and specific designations are trademarks and/or service marks of Altera Corporation in the United States and other countries. Other brands or products are trademarks of their respective holders. The specifications contained herein are subject to change without notice. All rights reserved.

these kinds of problems to be solved in software. Since all logic is handled by an Altera APEX 20KE FPGA and MAX3000A PLD sharing 1 MB external parallel flash, that task is somewhat more challenging.

It's often a good idea to check what's stored in flash memory first. Having neither the appropriate TSOP-48 hardware programmer adapter nor the patience to wait for one to arrive in the mail presents a ripe opportunity to explore boundary scan techniques for extracting data. Simply load the relevant Altera-provided BSDL files for the FPGA and PLDs into UrJTAG and it's possible to intercept control over all I/O lines. Without access to schematics, however, it's necessary to first probe out device interconnects in order to determine which pins could be used to bit-bang data from the external flash memory. Then it's just a matter of exercising the JTAG commands SAMPLE and PRELOAD in proper sequence or, better yet, just use UrJTAG's prototype external memory bus type to automate the process. If anything goes wrong, make sure to check the boundary scan definitions for helpful hints left for hardware hackers in the distant future.

```

2 --- *****
3 --- *          DESIGN WARNING          *
4 --- *****
6 attribute DESIGN_WARNING of EP20K160ET144 : entity is
8 "The APEX 20KE devices support IEEE 1149.1 testing "&
9 "before and after device configuration; however, "&
10 "the devices do not support this testing during "&
11 "device configuration. The easiest way to avoid "&
12 "device configuration is to hold the nCONFIG pin low "&
   "during power-up and testing.";
```

After waiting a brief eternity for data to shuffle back and forth from the boundary scan register, a complete dump of the external flash memory is finally available for analysis. One quick observation is that there are four binary chunks, each at evenly-spaced offsets and surrounded by empty space. Two of the chunks are the same size and similarly don't look particularly like any kind of program data. Since there is an FPGA on board, it's entirely reasonable to suspect that these are configuration bitstreams. The other two binary chunks, however, contain meaningful character sequences relating to flash programming and GPS operation. Even better, there's a signature near the beginning of both binaries spelling out "Nios." Finally, something that we can work with!

Basics of the Instruction Set

Even though Nios processors come in 16 and 32-bit variants, the instruction set is strictly 16-bit. Instructions are always half-word aligned, so the low-bit of the Program Counter (PC) is always zero.

Data and address bus size, as well as register and ALU width, are determined by the variant used. Most instructions are shared between both variants, but the 32-bit instruction set includes extra register manipulation functions and optional support for hardware multiply. This table highlights the instruction differences between the two variants.

Opcode	32-bit Name	16-bit Name
011001	STS16S	
011010	EXT16D	ADDC
011011	MOVHI	SUBC
011101101	ST16S	
01111100100	SEXT16	
01111101000	SWAP	
01111110001	ST16D	
01111110011	FILL16	
01111110100	MSTEP	
01111110101	MUL	
10010	PFXIO	

Code targeting the 32-bit variant is easy to recognize, as jumps require an extra register load.

```

1 ; Global so we can see it in dumps.
2 .global nr_jumptostart
3
4 nr_jumptostart:
5   PFX   %hi(_start@h)      ; 0x00
6   MOVI  %g0,%lo(_start@h)  ; 0x02
7   .ifdef __nios32__
8     PFX   %xhi(_start@h)    ; 0x04
9     MOVHI %g0,%xlo(_start@h) ; 0x06
10  .endif
11  JMP   %g0      ; 0x08 / 0x04 on Nios 16
12  NOP                    ; 0x0a / 0x06 on Nios 16
13  ; 0x0c / 0x08 on Nios 16 Signature.
   .byte  'N','i','o','s'
```

Five user-defined instructions, USR0 to USR4, facilitate accelerated data processing through additional logic placed in the hardware design. It might take some experimentation, or at least sufficient context, to determine the purpose of these types of instructions when no source is available.

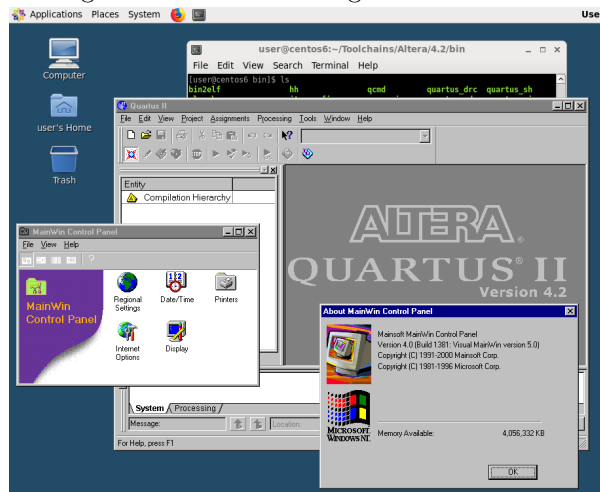
Registers and Calling Convention

If you have prior experience with SPARC or other Berkeley RISC descendants, you might enjoy seeing a familiar register layout as well as sliding register windows for stack cache and the use of branch delay slots.

	Inputs:	%r24 – %r31 (or %i0 – %i7)
2	Locals:	%r16 – %r23 (or %L0 – %L7)
	Outputs:	%r8 – %r15 (or %o0 – %o7)
4	Globals:	%r0 – %r7 (or %g0 – %g7)
	Saved return address:	%r31 (or %i7)
	Current return address:	%r15 (or %o7)
8	Frame pointer (%fp):	%r30 (or %i6)
10	Stack pointer (%sp):	%r14 (or %o6)

A Nios processor’s overall register file might span 128, 256, or 512 registers, depending on configuration. As the register window slides around, CWP is compared with the WVALID register (%ct12) to determine if a register underflow or overflow has occurred, which generates an internal exception. Unless specifically disabled, Nios designs include custom exception handlers which extend the register file with extra stack memory.

Nios programs lacking any kind of register window manipulation instructions might have been compiled with the `-mflat` option. This option was intended to improve timing predictability at the expense of overall context-switching time. As a result, only a fixed 32 registers are available to the application and register contents must be saved to stack memory during interrupts since register windows are no longer available for caching.



¹⁰unzip pocorgtfo21.pdf nios-epp-mmap.txt # Memory maps of reference designs.

Memory Map

A Nios processor’s memory map depends entirely on how it was configured. Assuming an implementation hasn’t strayed too far from one of the many original reference designs, Altera’s Embedded Processor Portfolio¹⁰ can serve as a convenient reference for correlating various peripherals to their base addresses or locating the exception vector table. Since a primary selling point of Nios (and soft processors in general) is reconfigurability, it’s possible that a complete understanding will require significantly more time and effort than with a conventional hard processor.

Interrupts and Exceptions

The exception vector table can reside in either RAM or ROM at a configurable offset specified in the processor design. The table holds up to 64 exception handler addresses, depending on configuration, with each entry occupying four bytes. Exceptions can be triggered by external hardware interrupts, internal exceptions, or software instructions. The first entry in the table is a non-maskable interrupt with priority 0 only intended for use by an optional on-chip instrumentation debug module.

If the exception vector table resides in RAM and consequently generated at run-time, try tracking down the initialization code, which might resemble the following instructions:

```
2 ;
3 ; Set up us the vector table
4 ; to catch any spurious interrupt
5 ; for great justice.
6 ;
7 .if __nios_catch_irqs__
8 .ifndef nasys_printf_uart
9     MOVIA %o0,r_spurious_irq_handler@h
10    MOVIP %o1,nasys_vector_table
11    MOVIP %o2,64
12    _init_vector_table_loop:
13        ST [%o1],%o0
14    .ifdef __nios32__
15        ADDI %o1,4
16    .else
17        ADDI %o1,2
18    .endif
19    SUBI %o2,1
20    IFRnz %o2
21    BR _init_vector_table_loop
22    NOP
23 .endif ; nasys_printf_uart
24 .endif ; __nios_catch_irqs__
```

Memory and Peripheral Access

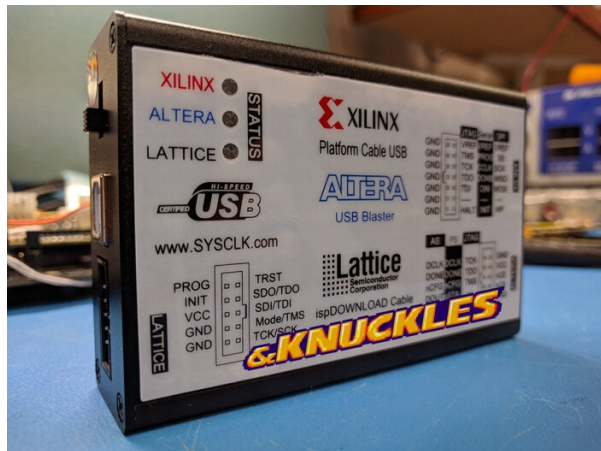
Nios has three address modes: (1) Full-width register-indirect, (2) Partial-width register-indirect, (3) and 5/16-bit immediate. Both of the register-indirect modes support an optional offset.

Nios requires the use of aligned memory accesses, so operations are performed on addresses which are multiples of two (16-bit variant) or multiples of four (32-bit variant). The lowest bit or two bits of the address are always treated as 0, respectively.

Partial-width memory reads require the combination of a full-width register-indirect read instruction with an extra EXT-prefixed extraction instruction. Partial-width memory writes, however, can be accomplished with a single dedicated ST-prefixed instruction. The additional FILL-prefixed instructions are helpful for meeting alignment requirements.

Disassembly

Don't worry if the Hex-Rays sales team stopped returning your phone calls. IDA Pro and other popular commercial tools don't currently support the Nios architecture anyway. Fortunately, some of the original components of the GNUPro Toolkit for Nios by Cygnus are still currently available on Sourceforge through the CDK4NIO project. At the very least, its Nios target support for GNU `binutils` is enough to get started with analyzing binaries.



Those familiar with Radare2 might recognize that its plugin infrastructure is well-suited to adding architectures already supported by `binutils`. Even if you enjoy leafing through actual pages of `objdump` output, consider the added value of Radare2's visual mode with colorized output, call graphs, integrated hex editor, and instruction emulation.

Implementing support for a new target architecture isn't as difficult as it might sound. The existing in-tree `nios2` arch support served as a convenient reference and starting point for implementing a `nios` arch plugin. After painstakingly modernizing the relevant code for contemporary compilers from the vintage `binutils` release, it was a quick process to write the required wrapper to hand off a byte sequence for disassembly.

Although this article only covers disassembly, complete target plugins implement an assembler, disassembler, code analysis, and a representation of each opcode using the Evaluable Strings Intermediate Language (ESIL) to enable emulation.

Support for uncommon architectures like Nios tends to end up in the `radare2-extras` repository,¹¹ otherwise known as the source graveyard, but Radare2 also includes a package manager which can conveniently download and build the plugin from source.

```
1 $ r2pm -i nios
...
3 $ r2 -a nios ./hello_world.out
```

As always, build Radare2 from Git master and rebuild often to take advantage of the latest improvements. If you happen to stumble across another rare or otherwise unusual architecture in the course of your hardware adventures, please consider taking a moment to implement your own plugin to keep the architecture alive in all of our hearts and minds.

I hope that you've enjoyed this friendly little guide to Nios, and that you'll keep it handy when reverse engineering firmware from that platform.

¹¹`git clone https://github.com/radareorg/radare2-extras.git`

SCORE WITH THE ALTERA POWERPLAY



DAC '97
June 9-11, 1997
Anaheim, CA
Booth 1574



```

1      ;-- strlen:
2      0x000809fe      1778      save sp,0x17
3      0x00080a00      1033      mov i0 , i0
4      0x00080a02      0132      mov g1 , l0
5      0x00080a04      0098      pfx hi(0x0)
6      0x00080a06      6138      and g1 , g3
7      0x00080a08      c17e      skprz g1
8      /-< 0x00080a0a      1280      br 0x00080a30
9      | 0x00080a0c      0332      mov g3 , l0
10     /-> 0x00080a0e      02b0      ldp g2 , [l0 ,0x0]
11     || 0x00080a10      4130      mov g1 , g2
12     || 0x00080a12      f79f      pfx hi(0xfee0)
13     || 0x00080a14      e437      movi g4 ,0x1f
14     || 0x00080a16      f79f      pfx hi(0xfee0)
15     || 0x00080a18      c46f      movhi g4 ,0x1e
16     || 0x00080a1a      8100      add g1 , g4
17     || 0x00080a1c      413c      andn g1 , g2
18     || 0x00080a1e      049c      pfx hi(0x8080)
19     || 0x00080a20      0234      movi g2 ,0x0
20     || 0x00080a22      049c      pfx hi(0x8080)
21     || 0x00080a24      026c      movhi g2 ,0x0
22     || 0x00080a26      4138      and g1 , g2
23     || 0x00080a28      417f      skprnz g1
24     /-< 0x00080a2a      f187      br 0x00080a0e
25     | 0x00080a2c      9004      addi l0 ,0x4
26     | 0x00080a2e      900c      subi l0 ,0x4
27     /-> 0x00080a30      04b0      ldp g4 , [l0 ,0x0]
28     || 0x00080a32      044e      ext8d g4 , l0
29     || 0x00080a34      447f      skprnz g4
30     /-< 0x00080a36      0980      br 0x00080a4a
31     | 0x00080a38      1832      mov i0 , l0
32     | 0x00080a3a      3004      inc l0
33     /-> 0x00080a3c      01b0      ldp g1 , [l0 ,0x0]
34     || 0x00080a3e      014e      ext8d g1 , l0
35     || 0x00080a40      c17e      skprz g1
36     /-< 0x00080a42      fc87      br 0x00080a3c
37     | 0x00080a44      3004      inc l0
38     | 0x00080a46      300c      dec l0
39     /-> 0x00080a48      1832      mov i0 , l0
40     /-> 0x00080a4a      7808      sub i0 , g3
41     || 0x00080a4c      df7f      ret
42     || 0x00080a4e      a07d      restore

```

Disassembly of `strlen` on Nios.



zines that teach cs concepts via cute drawings!
shop.bubblesort.io

```
static int disassemble(RAsm *a, RAsmOp *op, const ut8 *buf, int len) {
2   if (len < 2) {
      return -1;
4   }

6   buf_global = &op->buf_asm;
      memcpy(bytes, buf, 2);
8

      struct disassemble_info info = {0};
10

      info.disassembler_options = "";
12     info.mach = a->bits == 16 ? MACH_NIOS16 : MACH_NIOS32;
      info.buffer = bytes;
14     info.read_memory_func = &nios_buffer_read_memory;
      info.symbol_at_address_func = &nios_symbol_at_address;
16     info.memory_error_func = &nios_memory_error;
      info.print_address_func = &nios_print_address;
18     info.endian = !a->big_endian;
      info.fprintf_func = &nios_fprintf;
20     info.stream = stdout;

22     op->size = print_insn_nios((bfd_vma) a->pc, &info);

24     if (op->size == -1) {
          r_strbuf_set(&op->buf_asm, " (data)");
26     }

28     return op->size;
}
```

Radare2 plugin for disassembling Nios.

21:08 An Electromechanical Telephone Exchange

by Anonymous

Dear PML,

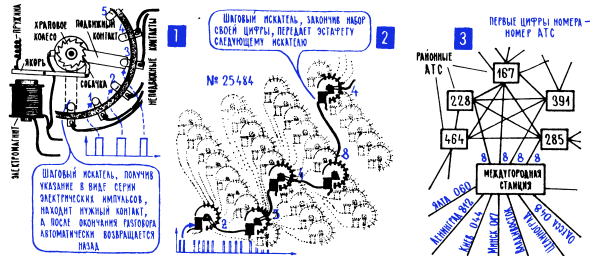
I'm sure you will enjoy reading this article and you will remember the old times when rotary dial phones were in common use. You will also remember the pioneering phreaks who explored telephone exchange equipment so they could make calls for free. One could implement special switch that, when activated, put a phone in high impedance mode, allowing the phone owner to accept incoming calls without sending an off-hook signal. The buzzer current would still rush in every four seconds, but the voice circuit was also connected so that callers might talk to each other, free-of-charge. In order to share the knowledge, the phreaks started the first hacker magazines as we know them today, *2600: The Hacker Quarterly* and *Phrack*.

Mechanical step-by-step dial telephone exchanges are now obsolete, replaced by electronic switching solutions which have themselves been replaced by TCP/IP.

Many people have asked me how to build small telephone exchanges with old mechanical parts, so here I am to share some working solutions. It's always fun to read about technology which appears to be functional as far back as 1891.

For easy circuits with telephones skip to page 32. For more complex design with two rotary stepping switches skip to page 37. Reprinted on page 41, this work was also described in Issue 3 of Paged.Out.

Before we begin, we must recognize the contributions of Svoryen Rudolf Antolievich and his book, *Electronics Step-by-Step*,¹² illustrated by S. Velitchkin and N. Frolov. This book first taught me about early telephone exchanges and stepping switches. Here's how a stepping relay is depicted in the book:



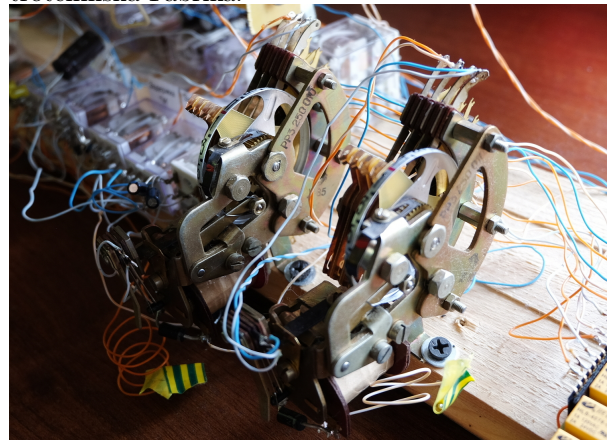
¹²<http://library.lol/main/8a7d6e726175e2679823293d58848f27> # Электроника шаг за шагом: Практическая энциклопедия юного радиолюбителя.

This mechanism can count pulses coming from a phone and advances its moving contacts to the next position. Most stepping switches are unable to step backward; when reset is needed, they continue to step in full circle until they get home.

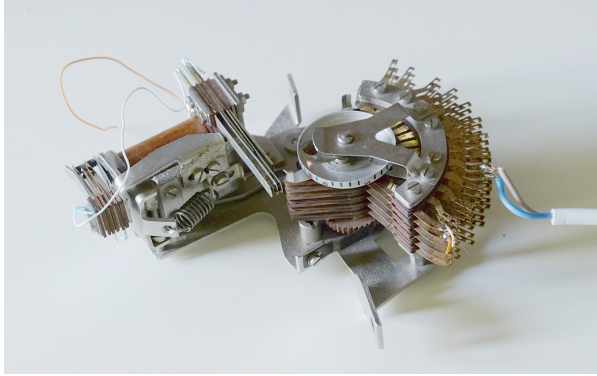
A local telephone exchange office is described in the book as a set of stepping relays. A destination number is sent into the line as a series of pulses. Each relay on its stage processes one digit and then passes the remaining pulses to the next stage. Pauses between each pulse series serve as delimiters between digits. The last stage relay performs the connection to the destination phone. Easy, isn't it?

I was wondering how many stepping relays are actually needed to process each call, and how they can be shared, to keep the number needed. It is obvious that stepping relays require additional circuitry to distinguish between long and short pulses, handle reset, avoid busy lines and apply buzzer current to the target phone to make it ring.

One day I got some stepping switches, namely two RR3 250 010 (one motion, 11-pole, 3-plane) switches manufactured by VEF Valsts Elektrotehniskā Fabrika.

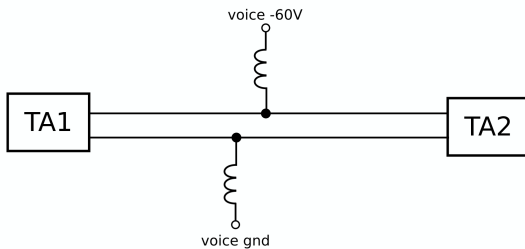


As well as a Tesla FN 935. Tesla was an electronic components manufacturer in Czechoslovakia, which made fine audio amplifiers and other audio equipment. It also developed special hardware for military use, including phone-line multiplexers, transmitters, and this 6-plane, 12-pole stepping switch.



Connecting Two Phones

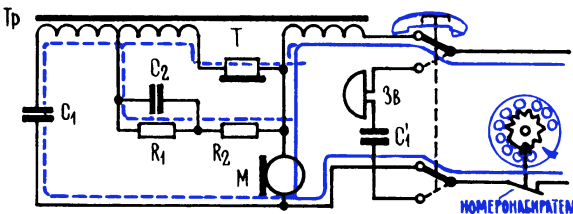
A simple two-way phone line may be built like this:



Coils are actually part of a relay. A special telephone relay with two coils and a single anchor can be used, but for now, it's okay to use two separate 24V relays. T coils pass direct current to power the phones, but resist alternating voice current and force it to go through another phone, rather than back through the power supply.

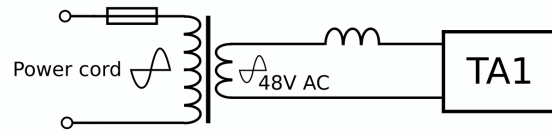
Ringling the Phone

Here is a simplified schematic of an old landline phone.

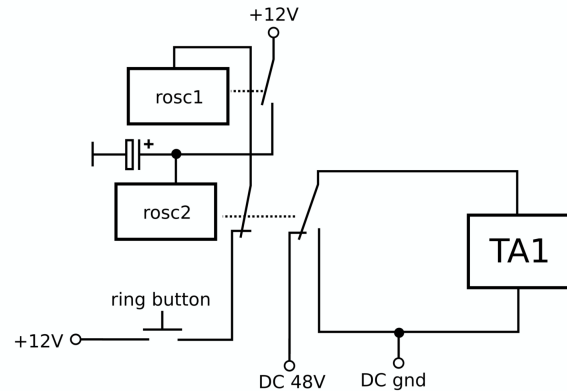


The phone does not pass direct current while on-hook, but it does pass alternating ring current. The ring capacitor value is typically $1\mu F @ 250V$. Some old phones have ring and voice capacitors made as a single unit, like the one shown here, but they are two separate capacitors anyway. So we need to apply al-

ternating current to the phone in order to make it ring, as in this design where a relay coil is used to decrease the ring voltage a little.



Another way to generate ring current is to use a pulse pair of two relays, like this:



A phone handset shouldn't be lifted during the ring cycle, as the voice circuit on newer phones can be damaged by the ring current. I blew up one of my phones this way. Also, direct voltage must be mixed with alternating voltage, it is required for some phones to operate correctly.

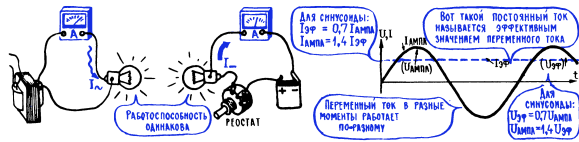
A more sophisticated schematic is shown in Figure 8, with the marked region discussed extensively here. A single relay with two coils A1 and A2 is used to power the phone. One coil can be shorted to make relay operate slower. This relay ignores ring current, but will detect off-hook direct current and disable ring. Busy relay RB is used to detect long (on-hook) pulse from the phone and perform reset.

An additional coil of power transformer is used to mix $-60V$ with 50/60 Hz 48V ring voltage. 50 Hz is a little fast for the phone, but still acceptable. Normal ring speed would be 25 Hz or so.

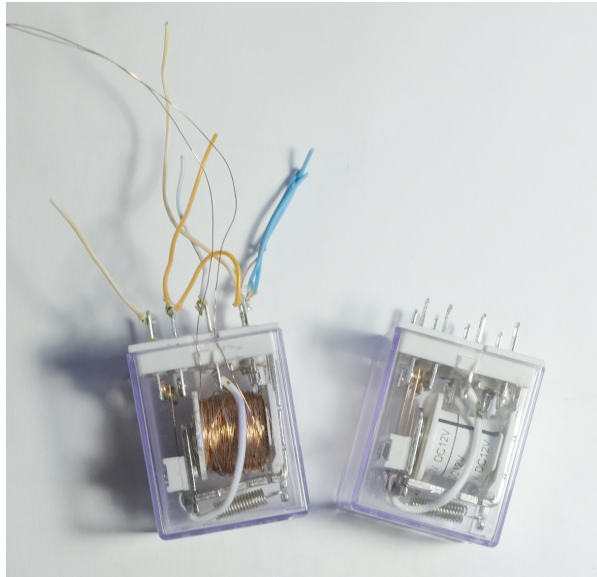
The positive power pole may be connected to ground, this will decrease corrosion on wires though an effect called cathodic protection, especially when an underground line is used. Unlike switching-mode power supplies, a power transformer galvanically separates the line from the mains network.

You might wonder why 48 VAC becomes 60 VDC

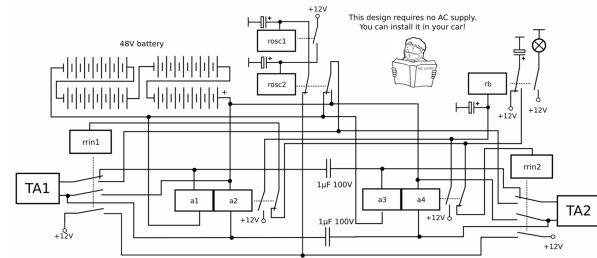
after the rectifier? That's because AC voltage is measured in RMS value. Peak voltage values can be derived from RMS when needed.



A normal single-coil relay can be modified to accommodate a second coil, simply by winding another coil up on top of factory coil, like this.



Two-way Phone Line with Ring



In this design, relays A1 to A4 are separate 24V relays without shared coils. The other five are 12V logic relays. 48V DC can be drawn from four 12V lead-acid batteries, or from some other source. $1\mu F$ capacitors divide the voice line into two independently controllable parts. The busy relay RB is shared between two phones. When one phone goes off-hook, a ring cycle will be triggered in the on-hook phone. The ring length is set by the value of the upper capacitor near the RB relay. This simple design is recommended for home use, as it requires little debugging effort.

Manual Switching

Almost all early manual telephone exchanges, both large and small, have parts in common. For each telephone there will be numbered socket on the board accompanied by line indication lamps. PBX operators would use connection kits to talk with customers and perform connections to other phones or trunks. Each connection kit has two wires with stan-

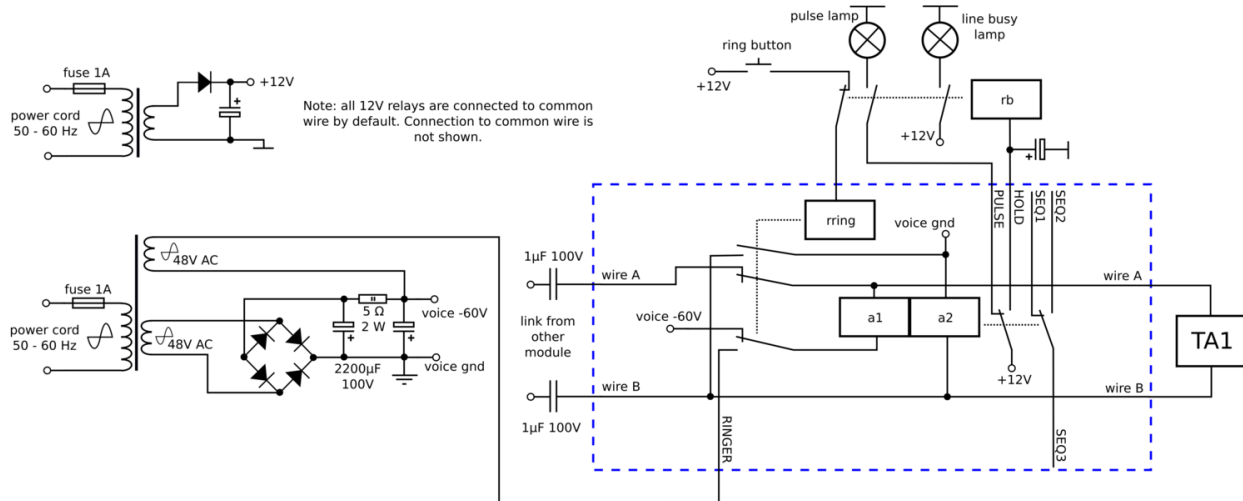
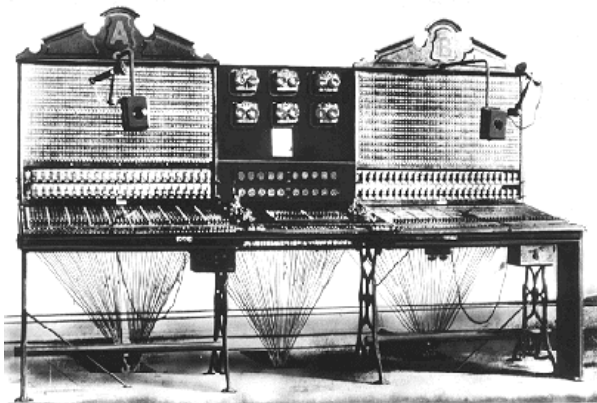


Figure 8: A more sophisticated phone schematic.

standard 3-pole TT jacks at the end. Each wire has a corresponding tri-state switch for sending ring signals and for connecting an operator's headset to the phone.

Trunk lines are used to route call between exchanges. Customers were not allowed to talk to other exchange operator via trunk lines, it's the operators who would have to talk to one another via the trunk to setup calls.

Manual exchanges have poor scalability. One improvement was to implement an exchange with separate panels, A and B. The operator at panel A would ask the customer about the destination number, while the operator at panel B would perform the connection to the destination phone.



¹³<http://library.lol/main/bbe9fc9ddf1be20fc491e8e4b997da70>

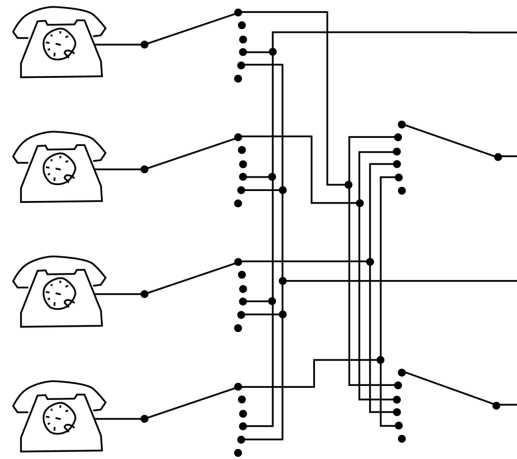
¹⁴<http://library.lol/main/321ac6b297bd75792f318fed8da5928e>

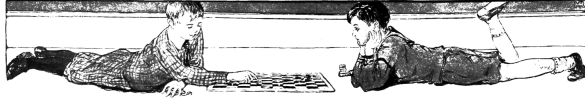
Automatic Step-by-Step Switching

I need to make a note about terminology. I have two reference books on my mind: first is a Soviet ATS-47 reference book,¹³ and the second is AT&T's training course.¹⁴ There is no strict terminology in this article, but I'm going to use the terms "forward seeking" and "backward seeking" from ATS-47 to explain one thing.

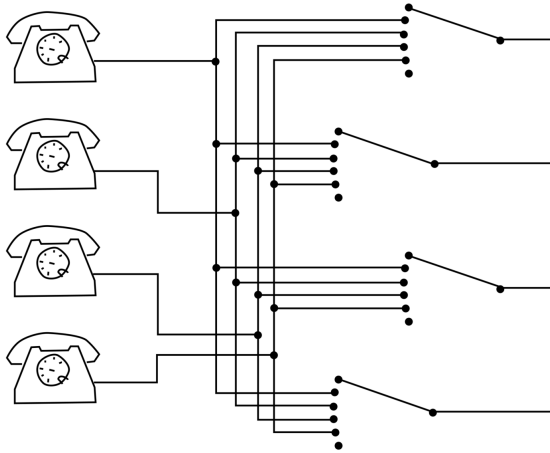
Each phone line usually has some fixed amount of equipment at the exchange office, a so called "line kit." For 10,000 phone lines there will be 10,000 line kits. Yes, sometimes two phones may be connected to one phone line and share a line kit. (Only one phone may be active, depending on polarity.) The line kit will activate other equipment when a customer begins the call. The line kit must be relatively small, especially for a home-made telephone exchange.

The Soviet ATS-47 setup uses forward seeking at the call pre-setup stage. This means that each line kit has a stepping switch which selects a free line for next dialing stage. After the call, it must return back to the home position to accept more incoming calls. Up to ten group-seekers can be connected to the outputs. This works faster than the backward seeking method, but it requires one rotating one-motion switch to be in each line kit, which can be expensive.





Backward seeking is more like manual switching. The kit consists of two switches, with the first switch (the line hunter) navigating to the phone that needs to be served. The second switch accepts one digit and connects to the destination phone.



Switch operation may be driven by pulses coming from the phone, which is slow, or it may be free switching, where the switch quickly steps itself until it finds the right position.

Strowger Switches and Trunk Lines

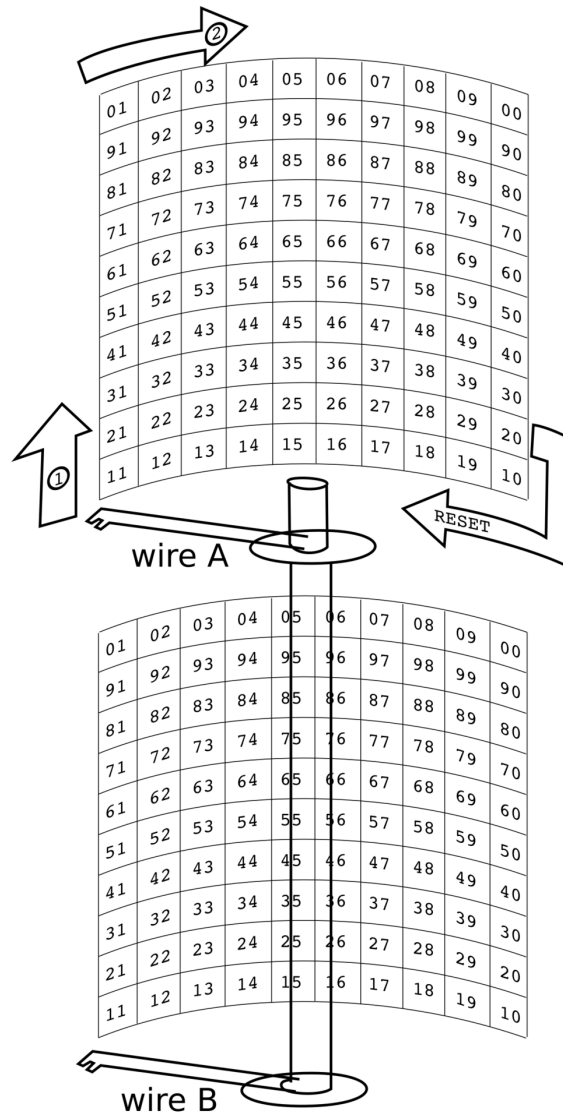
It's a good idea to use a Strowger switch with 100 outputs as an output switch. It will consume the two final digits of the phone number to drive its arm up and then around to the desired contact.

If we have more than 100 phones we'll use group seeking. A group seeker is a Strowger switch that accepts one digit to drive the arm up, and then seeks a free device from the next stage to continue dialing. Up to ten devices may be connected to the digit. If no free device is found, the arm will stop at the "trunk busy" contact at the end of the row. The group seeker must find the free line before more digit pulses begin to arrive, so all but the very oldest exchanges have a special mechanical feature to increase the pause between digits.

The line hunter is a Strowger switch, too. It connects one of the phones to the first stage group seeker. A dial tone in the handset means that first stage seeker is waiting for its first digit. A special

device is used to prevent multiple seekers from hunting the same phone. When all line hunters are busy, there will be no dial tone heard. Hot line phones, such as street phones and PBX trunks, may be connected to the first stage directly without the pre-setup stage, such phone will have one dedicated first stage device.

Wires A and B are used for voice and pulses transfer and for reset signaling between group seekers. Wire C is used to mark the line as free or busy. Most Strowger switches have three arrays of outputs and three contact arms, sometimes two fields are sufficient.



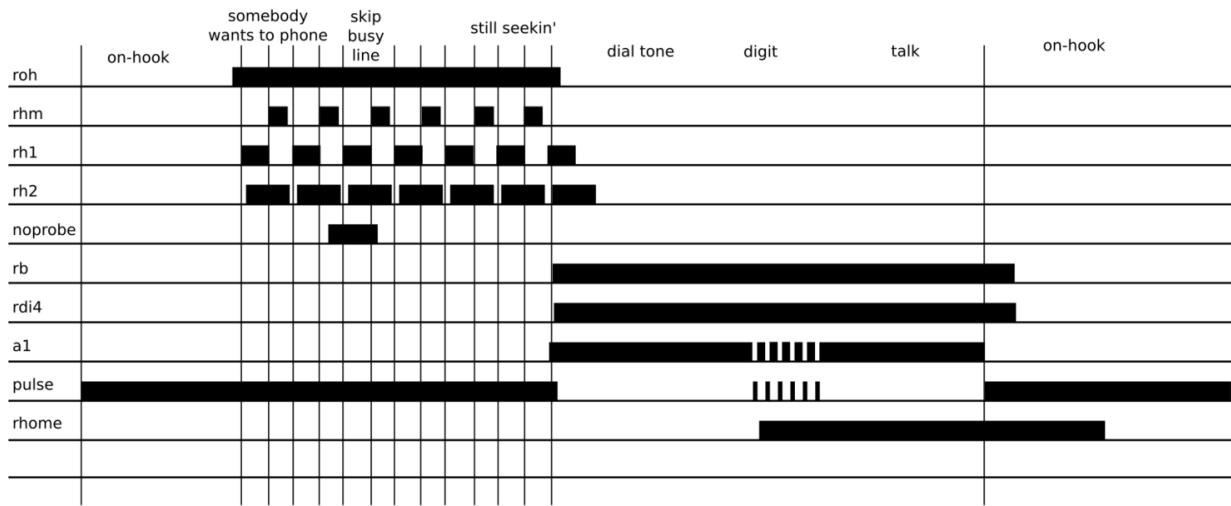


Figure 9: First Timing Diagram

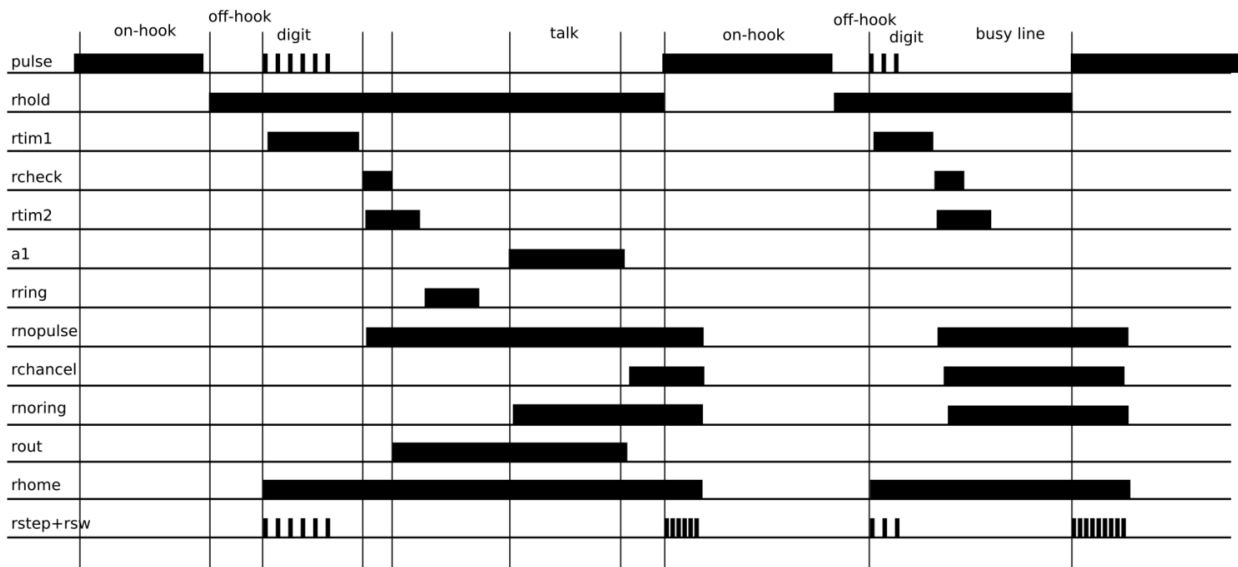
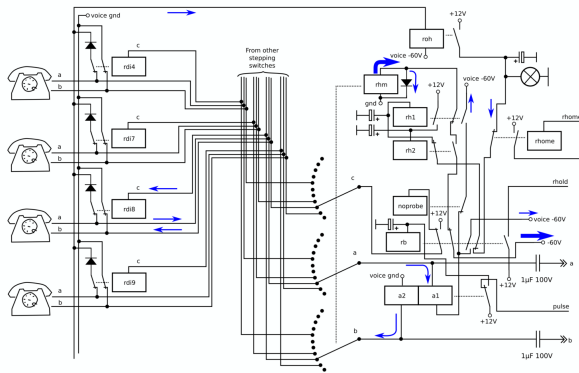


Figure 10: Second Timing Diagram

PoC Telephone Exchange Unit



I'm going to show you a more complicated design, one that I've built. It has just two rotating switches, so only one pair of telephones may use the connection kit, and other telephones must wait their turn. Up to ten phones may be used total. There is no dial tone circuit, as that would require an additional coil on the A1-A2 relay.

All About

OSI

BASIC-IN-ROM

Ohio Scientific Microsoft BASIC Ver 1.0 Rev 3.2
REFERENCE MANUAL

Complete, Concise, Accurate, Detailed. All commands, statements, and functions. Maps. USR. Tapes. Bug fixes. Variable tables. Source code storage. MONITOR.

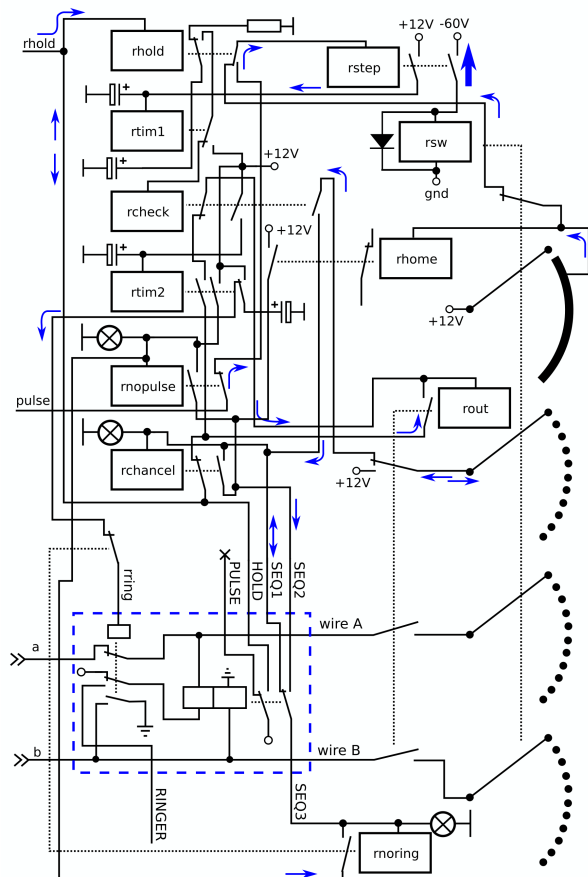
Postpaid \$8.95. Send check, or COD

EDWARD H. CARLSON 259
3872 RALEIGH DR.
OKEMOS, MI 48864

Dealer Inquiries Welcome

In real-world systems, wire C is more complicated. One simplification was to use HIGH (12V) logic level on wire C as a busy signal, to keep the line kits small. Here every line kit contains one relay and one diode. (An optional indication bulb may be hooked to wire C.) This design can be joined with a manual exchange desk, provided that each socket will pull C high to prevent the line from automatic switching.

ROH is an off-hook detection relay, it has 48V coil. The A1, A2 relays have 24V coils. All others are 12V logic relays with the other pole connected to logic ground. Some of them have capacitors to increase release time. Figure 9 is a timing diagram that will help you understand the logic.



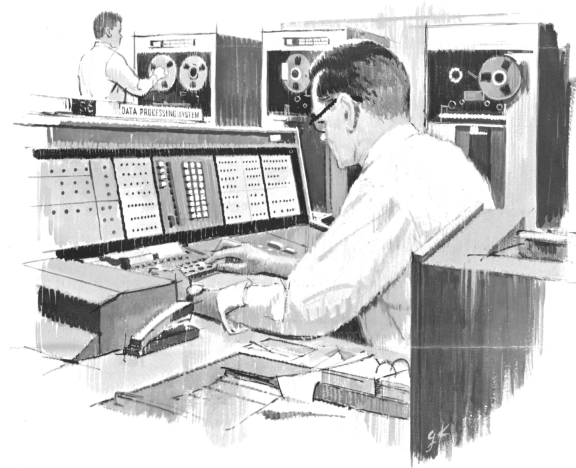
RHM and RSW are stepping relay magnets. They draw current up to 1A, so protection diodes are there to reduce sparks on relay contacts. Both stepping relay and phones draw power from the same power transformer, but there are two diode bridges, one for voice circuit and one for the magnets. Without that, the RSW relay would cause brownouts on the -60V supply and the phone would be unable to send clear pulses. The second part of the circuit performs the line checking sequence. diagram for second part: This second timing diagram is shown in Figure 10.

What are values of those capacitors?

Capacitors are used here to increase relay release time. When a relay is powered, the capacitor charges itself and accumulates a bit of energy. After power removal, the capacitor will power the relay for two seconds or so, and then discharge. The resulting delay time depends on the relay armature type, capacitor value and other things.

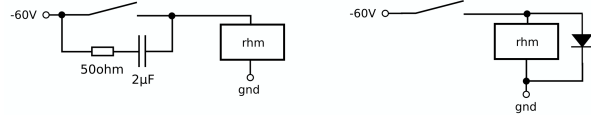
A $1000\mu\text{F}$ capacitor with 12V relay coil will give 200 milliseconds delay or so. It's suitable for the RB, RTIM1, RTIM2, RCHECK relays. $100\mu\text{F}$ is good for fast processes, like line hunters RH1 and RH2. $3300\mu\text{F}$ will power the RRING relay, giving 600 milliseconds of ringing.

An alternative way to slow down a relay is to short-circuit one of its coils or add a copper disc near the coil. This increases the inductance of the relay, and therefore also its release time.



¹⁵unzip pocorgtfo21.pdf phones3.mp3

How to protect relay contacts?



Inductive loads like relay coils, large electromagnets, and DC motors will generate inductive peak voltage when disconnected. This inductive current will generate sparks and degrade switching contacts that control inductive circuit, so a protection diode is needed.

ATS-47 uses RC filters instead. Spark current can have high frequency components, and the filter will pass it around and decrease interference to other equipment. It's okay to use both diodes and RC-filters, of course.

Audio!

I've made a small audio recording.¹⁵ Listening to it, you can hear my telephone exchange unit in action. Various phone rings and rotary dial sounds follow starting from 0:37. The ring sound will differ depending on buzzer current frequency, I've tried both fast (50Hz) and slow (using relays).

How can we trace a call?

On the ATS-47 all switches used in a call cannot be released until the recipient phone goes back on its hook. This can be used to track the originating phone, even if the caller has gone back on the hook.

Having received a call, the recipient might dial a special digit to signal an operator, who can then be asked to track the circuit back to the originating phone. During this time, the caller is unable to make another outbound call.

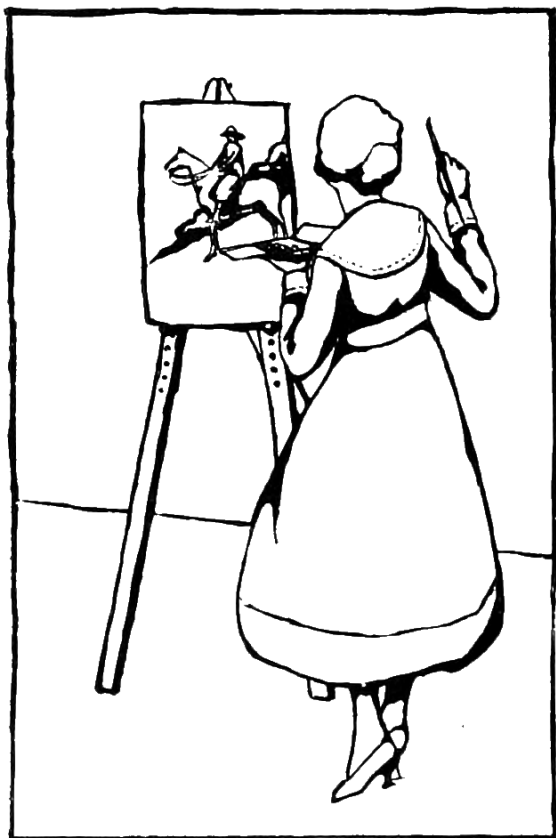
Wires A and B between switches are used to transfer signals about answered phone and reset, along the voice. These signals activate the successful calls counter, which is used for billing. A first stage seeker may swap the originating phone line polarity when the other phone is answered. Manual exchanges use this as an additional indication that recipient is ready to talk. Street phones will grab the coin when the line polarity is reversed.

A customer often forgets to place his handset on-hook. There is a special circuit to detect such phones, with a delay of one or two minutes.

Greetings for the neighbors!

I'm not alone in building relay-based devices. In fact, there is whole Hackaday thread about this.¹⁶ There's also Artyom Kashcanov aka Radiolok (Hi!) and his *BrainfuckPC*.

There is Harry Porter and his *HPRC*; Harry made a very nice introductory video about relay logic.¹⁷ And Michael whose demonstration setup using five Strowger switches is well documented at his homepage.¹⁸ Em Lazer-Walker uses small PBX switchboard as a front-end for her videogame. Frankfurt's Museum of Communication also has a large working demonstration setup.



1	Exxon
2	General Motors
3	Mobil
4	Ford Motor
5	IBM
6	Texaco
7	E.I. du Pont
8	Standard Oil (Ind.)
9	Standard Oil of Cal.
10	General Electric
11	Gulf Oil
12	Atlantic Richfield
13	Shell Oil
14	Occidental Petroleum
15	U.S. Steel
16	Phillips Petroleum
17	Sup


**27
million
Americans
can't read.
And guess
who pays
the price.**

Every year, functional illiteracy costs American business billions.

But your company can fight back...by joining your local community's fight against illiteracy. Call the Coalition for Literacy at toll-free **1-800-228-8813** and find out how.

You may find it's the greatest cost-saving measure your company has ever taken.

**A literate
America is a
good investment.**

**Ad
Council**  Coalition for Literacy



Softstrip

¹⁶<https://hackaday.io/project/11798-relay-based-projects>

¹⁷<http://web.cecs.pdx.edu/~harry/Relay/VideoTutorial/index.html>

¹⁸<https://www.seg.co.uk/telecomm/automat4.htm>



ABC 2	MNO 6	OPER 0	OPER 0
----------	----------	-----------	-----------

FLASH...

IF YOUR NUMBER HAS BEEN CHANGED

... Advise friends of the change—they'll be able to call you more easily.

The wheel that lets you roll around town without leaving home.



Just spin the dial and you're talking to the pharmacist. Spin it again and reach your insurance man. Or chat with a friend across town. Count on it any time—to save you time and traveling. Saves money, too. What else does so much yet costs so little?

please dial carefully

Please be careful not to confuse the letter "I" with the numeral "one," or the letter "O" with the numeral "zero" when dialing.

✓ WHENEVER POSSIBLE remain on the line after you've placed a call



DIAL CAREFULLY! Avoid wrong numbers by first checking the number and then DIAL EACH DIGIT CAREFULLY!

New York Telephone
1095 Avenue of the Americas
New York, New York 10036

•Hummm-mm-mm. That's the tone—your "go ahead" signal to start dialing. Please wait for it.

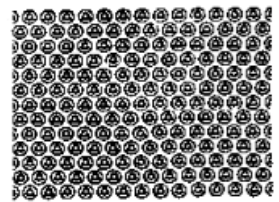
after you finish your call.....

be sure to replace the receiver

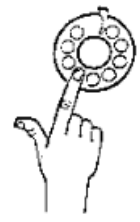
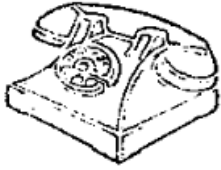
Answer Promptly

Don't make a friend wait or miss a call by not answering your telephone at once.

WHEN ANSWERING THE TELEPHONE, always identify yourself, your firm, or your department.

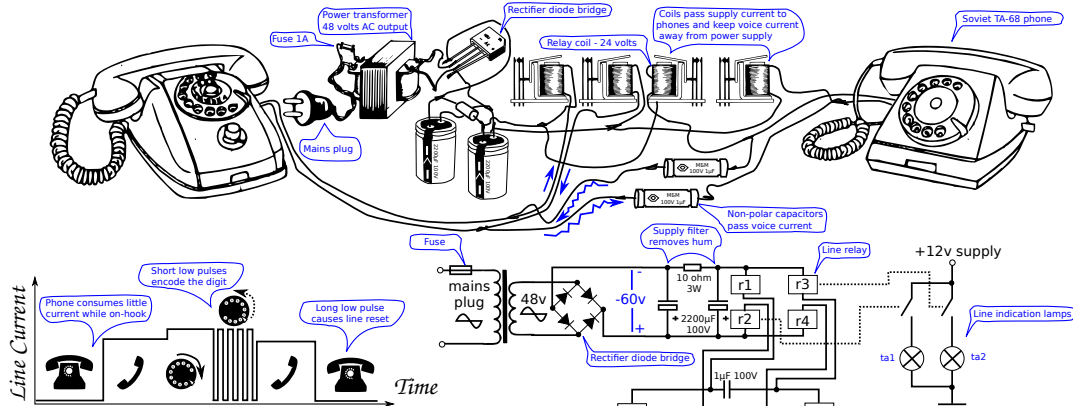


"Wow, another bank machine!"

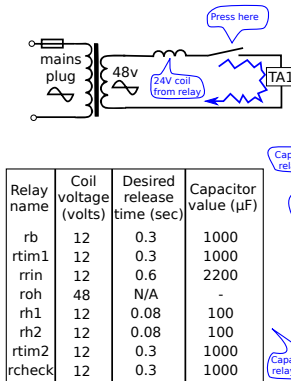


Do-It-Yourself: Stepping Telephone Exchange

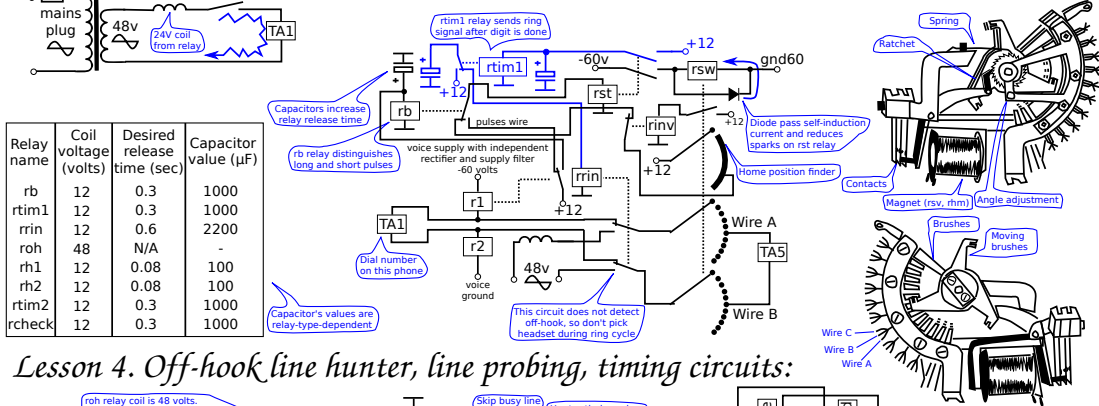
Lesson 1. Connect two phones:



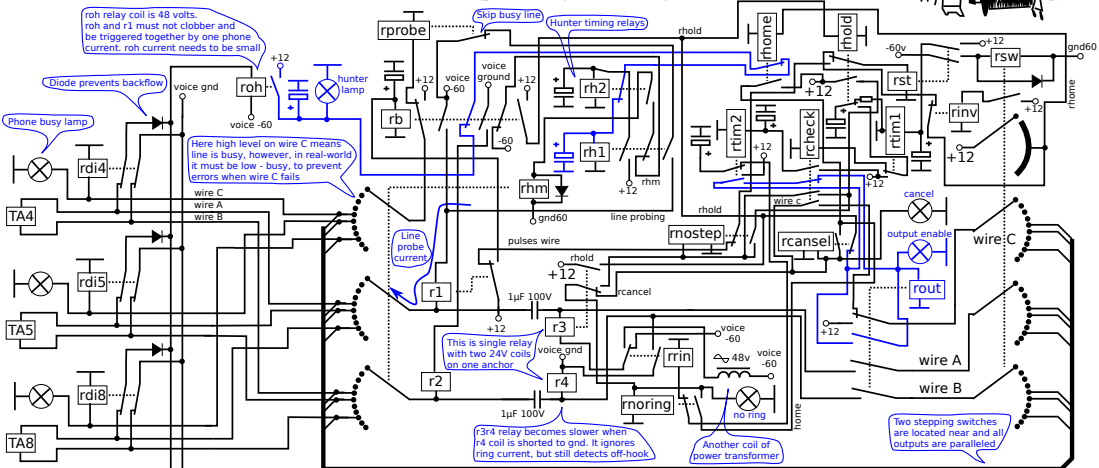
Lesson 2. Ring the phone:



Lesson 3. Connect phone to Stepping Switch (easy way):



Lesson 4. Off-hook line hunter, line probing, timing circuits:



See also: International Journal of Proof of Concept or GTFO issue 0x21, long article about phones

-> pagedout.institute <- issue #3 2020

21:09 An ELF Palindrome for AMD64

by Netspooky

The first Binary Golf Grand Prix was a challenge issued on Twitter to create a small binary that executed the same forwards as it did backwards. Included were certain rules, such as ensuring execution past the halfway point in the binary, and that scores would be based on the ratio of overall number of bytes executed to total bytes in the file.

The binary I chose to target was a 64-bit ELF binary, due to my familiarity with creating weird ELF's. I began investigating strategies for creating a palindromic binary in this format because there are quite a few sensitive areas that must remain intact for a binary to run at all.

Initial Efforts

I had already established a baseline of a barebones golf'd 64-bit ELF, and my previous attempts to produce the smallest 64-bit ELF yielded a binary that was 84 bytes in size. I chose this as my starting point.


Since I used `nasm` to create ELF files, I began by first flipping the entire source code backwards after the end of my existing source code, and meticulously placing bytes in the correct order. After I finished, I used a Perl one-liner to flip the binary backwards, then executed both binaries and compared their hashes to validate my work.

The next step was to create a payload that would be both valid, and easy to work with in both directions. My first idea was to use alphanumeric shellcode, as outlined in Phrack 57:15,¹⁹ to have a series of single byte instructions that would also display a palindrome in the hex dump output. The issue with this approach is that alphanumeric shellcode is based on 32-bit x86, which wouldn't work to run on 64-bit Linux.

I also wanted my palindrome to be readable, and since palindromes tend to rely on the ambiguity of punctuation to work, my palindrome would have to use words that could be read if presented as a single string of alphanumeric characters. I decided to go with the phrase "PULLUPIFIPULLUP," because it was readable. Testing this in a disassembler showed that certain characters would not be valid machine code.

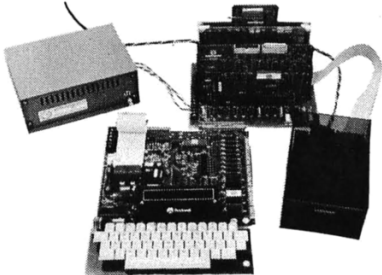
I tested all of the alphanumeric characters in a disassembler and realized that even fewer characters are usable than in 32-bit mode. This is due to prefix instructions taking the place of references to smaller registers, and certain encodings changing. These were the characters that were safest to use:

Op	Instruction	Char
50	push rax	P
51	push rcx	Q
52	push rdx	R
53	push rbx	S
54	push rsp	T
55	push rbp	U
56	push rsi	V
57	push rdi	W
58	pop rax	X
59	pop rcx	Y
5a	pop rdx	Z



compaq
microsystems

There is nothing like a
DAIM



A complete disk system for the Rockwell Aim 65. Uses the Rockwell Expansion Motherboard. Base price of \$850 (U.S.) includes controller with software in Eprom, disk power supply and one packaged Shugart SA400 Drive.

224 SE 16th St. AMES, IA 50010
P.O. BOX 687 (515) 232-8187

¹⁹[unzip pocorgtfo21.pdf phrack5715.txt](#)

Luckily, there are vowel sounds that can be used to find some words and write my own palindrome. An online Scrabble word finder came in handy for this. After searching for words to use, I ended up with the phrase “PUPPY SPY, PSY P. PUP”.

The nice thing about these particular instructions is that they are `push` and `pop` instructions, so you don’t have to worry too much about messing up data that might be in these registers, and just have to track where values might end up if you use them at all.

Mirroring

The template 64-bit ELF source only executes seven bytes to perform the exit syscall:

```

1 0: b0 3c          mov    al,0x3c
2: 48 31 ff        xor    rdi,rdi
3 5: 0f 05          syscall

```

What was particularly interesting was that when reversed, the bytes are actual usable instructions.

```

1 0: 05 0f ff 31 48  add    eax,0x4831ff0f
5: 3c b0          cmp    al,0xb0

```

This was a very lucky discovery, and I started thinking even more about interpreting instructions backwards. One of the challenges in something like this is that x86 has variable length instructions, and using bigger registers with smaller values gives a lot of null bytes to contend with. This means that carefully planning certain instructions of the basic operations I wanted to do was next on my list.

There is quite a lot of variance in both assemblers and disassemblers in generating and reading code, so ensuring that the source is assembled properly is of utmost importance. I ended up only using `nasm` and `ndisasm` to verify that instructions were what I wanted them to be.

Now that I had some ideas, I started padding out the remaining sections that might contain code with `nops`, so that at the very least, I had some wiggle room when calculating things like jumps. Since the code began at offset `0x4` in the header, padding with five `nops` filled the rest of the space to offset `0xF`.

Getting an idea of how to use jumps was the next thing to sort out. I figured that `jmp` instructions could be accounted for in one of two ways: either a pairing of `jmps` that jump over each other, or a `jmp` that is interpreted as something else backward.

I wrote a small script to generate all of the possible opcode combinations for short jumps and what they disassemble to when interpreted backwards. Even though it’s only two bytes, `EB` and the one byte `jmp` distance, there are a lot of incompatible instructions, such as references to `EBP` and other registers that aren’t easily usable in `x64`.

```

import sys
import subprocess

# python3 opiter.py opcode
# Will iter through one byte in front of the
# opcode you put in there. It's hellla
# bespoke, feel free to change heh

exp = sys.argv[1]

for i in range(0,255):
    opp = format(i, '02x')
    inf = ''+opp+' '+exp+' '
    print(opp+" "+exp+" |",end=" ")
    process = subprocess.run(
        ['/usr/bin/rasm2 -a x86 -b 64 -d '+inf],
        shell=True, check=True,
        stdout=subprocess.PIPE,
        universal_newlines=True)
    output = process.stdout
    if 'invalid' in output:
        print("—")
    else:
        print(output ,end="")

```

This is the output from the `jmp` bruteforce table with invalid opcodes ignored:

```

00 eb  add bl, ch      | 2c eb  sub    al, 0xeb
01 eb  add ebx, ebp   | 30 eb  xor    bl, ch
02 eb  add ch, bl    | 31 eb  xor    ebx, ebp
03 eb  add ebp, ebx   | 32 eb  xor    ch, bl
04 eb  add al, 0xeb  | 33 eb  xor    ebp, ebx
08 eb  or  bl, ch    | 34 eb  xor    al, 0xeb
09 eb  or  ebx, ebp   | 38 eb  cmp    bl, ch
0a eb  or  ch, bl    | 39 eb  cmp    ebx, ebp
0b eb  or  ebp, ebx   | 3a eb  cmp    ch, bl
0c eb  or  al, 0xeb  | 3b eb  cmp    ebp, ebx
10 eb  adc bl, ch    | 3c eb  cmp    al, 0xeb
11 eb  adc ebx, ebp   | 63 eb  movsxd rbp, ebx
12 eb  adc ch, bl    | 6a eb  push  0xfffffffffeb
13 eb  adc ebp, ebx   | 70 eb  jno  0xfffffffffed
14 eb  adc al, 0xeb  | 71 eb  jnb  0xfffffffffed
18 eb  sbb bl, ch    | 72 eb  jbe  0xfffffffffed
19 eb  sbb ebx, ebp   | 73 eb  jae  0xfffffffffed
1a eb  sbb ch, bl    | 74 eb  jec  0xfffffffffed
1b eb  sbb ebp, ebx   | 75 eb  jne  0xfffffffffed
1c eb  sbb al, 0xeb  | 76 eb  jbe  0xfffffffffed
20 eb  and bl, ch    | 77 eb  ja   0xfffffffffed
21 eb  and ebx, ebp   | 78 eb  js   0xfffffffffed
22 eb  and ch, bl    | 79 eb  jns  0xfffffffffed
23 eb  and ebp, ebx   | 7a eb  jpe  0xfffffffffed
24 eb  and al, 0xeb  | 7b eb  jnp  0xfffffffffed
28 eb  sub bl, ch    | 7c eb  jl   0xfffffffffed
29 eb  sub ebx, ebp   | 7d eb  jge  0xfffffffffed
2a eb  sub ch, bl    | 7e eb  jle  0xfffffffffed
2b eb  sub ebp, ebx   | 7f eb  jg   0xfffffffffed

```

After generating all of these instructions, I realized that the distance between the code at `0xF` and the corresponding code on the other half of the binary was too great for a short jump. I moved on to the next phase, working out some sort of code to jump to within the main binary. There was another example of tiny code I had used in previous work, a stream covering approach to assembly code optimization called `i2ao`.²⁰ This code was simple and portable enough to reuse for this application. The code simply printed out a string and exited.

Now, we have a palindrome that works as both code and printable text, all of the possible short jumps, and some basic code to print the string, it was time to put it all together.

Putting it all together

Throughout this, you can refer to both the finished assembly code, and the diagram featuring the full labeled binary. If you are unfamiliar with the ELF format, check out Ange Albertini's Corkami ELF file explanations on Github!

The primary concern with all of this was to make sure that the registers we need are cleared prior to making a syscall, lest we segfault. In this case, there are two calls to make: `write` and `exit`.

The registers required for a `write` syscall are `RAX`, `RDX`, `RDI`, and `RSI`. Since the first instructions executed add values to `RAX`, an explicit `mov rax, 1` is needed, rather than any clever tricks to populate `RAX`. If we wanted to use something like `xor rax, rax; inc rax`, it would add an extra byte. Some other space saving measures are also used in the `write` syscall code, which you can refer to in the `i2ao` writeups or video.

The next step is to reference the string within the code that is immediately after the `write` syscall. There are a few ways of making references to the current offset, but none of them made much sense other than simply knowing where in the binary the string is, and moving that value into the `sil` register. This can be achieved by assembling your binary, and opening in a debugger before executing, to get the exact values needed.

After the `write` syscall code was sorted, it was time to start mirroring the entire executable section. Since the bounds of the headers have already been established, you can safely do this without messing up your binary. Jumps from the main code sec-

tion back into the reverse header will be determined later.

The `write` syscall code doesn't really have too many instructions that you can safely execute backwards without entirely rewriting it. So instead of that, another approach is to simply jump over whatever wasn't executable. The alphanumeric machine code was placed before the `write` code, so that it could be used as a sled and have a known location. Since this is executable and won't interfere with the flow of the program, a `jmp` can be placed between that and the backwards code for the `write` syscall.

The size of the `write` code, along with the short `jmp`, produces `jmp 0x17`, which turns into `eb15` in machine code. This unfortunately doesn't translate to anything usable backwards. Referring to the `jmp` table, there is a usable instruction `sbb bl, ch`, that can be achieved by padding with three `nops` to bring the opcode to `18eb` when backward, `eb18` forward. This would create a nice way to both jump over junk code, and still maintain executability in the code.

All of this `jmp` encoding was done mainly to prevent generating even more junk bytes to account for. Another solution would be to just encode a `jmp` instruction backwards, `02eb`, after the `jmp` to the `write` syscall label, which would do a small hop over the `jmp 0x17` that we can't execute backwards. This approach felt cleaner in the end.

Now all we have to do is just execute our string, clear `RAX`, and jump back into the headers. This operation just adds a small, five byte block that we have to account for when we jump out of the headers the first time and into the main code section.

A space saving trick used here was to completely overwrite the `p_align` section in the ELF's program header, saving 16 bytes in total (eight on each side) within the code section.

²⁰[unzip pocorgtfo21.pdf i2ao.zip](#)

SWTP 6800 OWNERS—WE HAVE A CASSETTE I/O FOR YOU!

The CIS-30+ allows you to record and playback data using an ordinary cassette recorder at 30, 60 or 120 Bytes/Sec. No Hassle! Your terminal connects to the CIS-30+ which plugs into either the Control (MP-C) or Serial (MP-S) Interface of your SWTP 6800 Computer. The CIS-30+ uses the self clocking 'Kansas City/Biphase Standard'. The CIS-30+ is the FASTEST, MOST RELIABLE CASSETTE I/O you can buy for your SWTP 6800 Computer.



PerCom has a Cassette I/O for your computer!
Call or Write for complete specifications

Kit — \$69.95*
Assembled — \$89.95*
(manual included)
*plus 5% f/shipping



PerCom Data Co.
P.O. Box 40598 • Garland, Texas 75042 • (214) 276-1968



PerCom — 'peripherals for personal computing' TEXAS RESIDENTS ADD SALES TAX

Final Optimizations

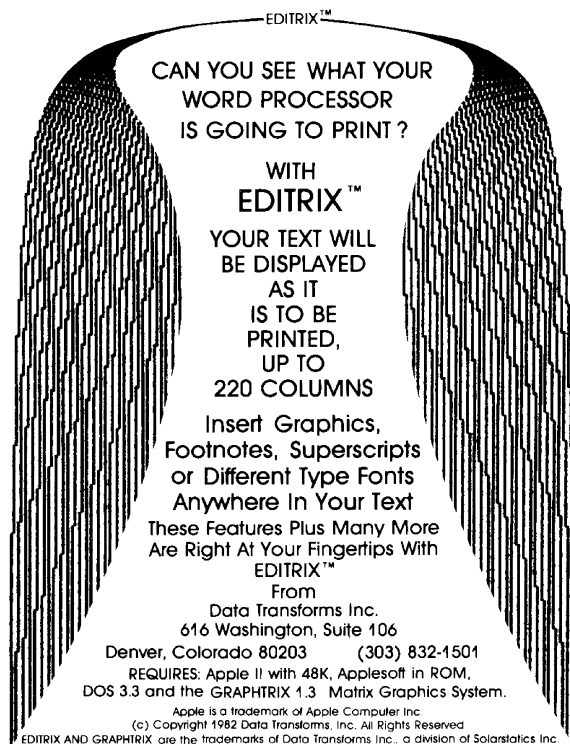
Due to the jump from the header to the main code area, there was junk code from 0x4 to 0x10, where the binary begins and ends execution. So, a final step was tried to utilize all the space here.

Previous fuzzing of the various headers determined that there is writable space at both 0x3C and 0x44 in the program header. They must be exactly the same or the binary will not execute. Each of these spots has four bytes of space to work with, which is perfect to do something simple like a short `jmp`.

A short `jmp` from the top of the ELF header at 0x0E to 0x44, produces some bytes that are usable backwards! This is `eb34`, which backwards, `34eb` decodes to `xor al, 0xeb`. Since it's only messing with AL, the lowest byte of RAX, this operation doesn't matter because the value is explicitly assigned afterwards. *Chef's kiss!*

This ensures that when we jump from the main code section, we will be able to use all of the bytes at the end of the binary before the `exit` syscall. Additionally, this increased the total number of executed bytes by four, bringing the grand total to 90 bytes executed out of 245 total.

The final code is shown on page 47.



EDITRIX™

CAN YOU SEE WHAT YOUR
WORD PROCESSOR
IS GOING TO PRINT?

WITH
EDITRIX™

YOUR TEXT WILL
BE DISPLAYED
AS IT
IS TO BE
PRINTED,
UP TO
220 COLUMNS

Insert Graphics,
Footnotes, Superscripts
or Different Type Fonts
Anywhere in Your Text

These Features Plus Many More
Are Right At Your Fingertips With
EDITRIX™

From
Data Transforms Inc.
616 Washington, Suite 106
Denver, Colorado 80203 (303) 832-1501

REQUIRES: Apple II with 48K, Applesoft in ROM,
DOS 3.3 and the GRAPHRIX 1.3 Matrix Graphics System.

Apple is a trademark of Apple Computer Inc.
(c) Copyright 1982 Data Transforms, Inc. All Rights Reserved.
EDITRIX AND GRAPHRIX are the trademarks of Data Transforms Inc., a division of Solarstics Inc.



Confirming Functionality

This was tested and built on Ubuntu 20.04 with kernel 5.4.0-42-generic. Here is a small script you can use to build and test the ASM file, and execution is shown on page 46.

```
#!/bin/bash
2
3 nasm -f bin ns.bggp.asm -o ns.bggp
4 chmod +x ns.bggp
5 echo "Executing initial binary..."
6 ./ns.bggp
7 echo ""
8 xxd ns.bggp
9 echo ""
10 echo "Reversing..."
11 perl -0777pe '$_=reverse $_' ns.bggp > ns.R
12 chmod +x ns.R
13 echo "Executing binary in reverse..."
14 ./ns.R
15 echo ""
16 xxd ns.R
17 echo ""
18 echo "Comparing hashes..."
19 sha1sum ns.bggp
20 sha1sum ns.R
```

Final Thoughts

I might've shrunk the `write` syscall code down even more to try and save 1 byte to produce a short `jmp` of `0xeb12->0x12eb` (`adc ch, bl`). Since I was coding not just for size, but for percentage of bytes executed as well, it made more sense just to leave things as they were.

It will be exciting to do another challenge like this next time around, and hopefully expand on the competition as a whole. If you'd participate, or have any questions / comments, you can email me at u@n0.101 or talk to me on Twitter [@netspooky](https://twitter.com/netspooky).

A special thank you goes to everyone who competed in the first Binary Golf Grand Prix, 0xdade, ThugCrowd and Hermit. :}

```

2  $ ./build.sh
4  Executing initial binary...
   PUPPYSPYPSYPPUP
6  00000000: 7f45 4c46 050f ff31 483c b090 9090 eb34 .ELF...1H<....4
   00000010: 0200 3e00 0100 0000 0400 0000 0100 0000 ..>.....
8  00000020: 1c00 0000 0000 0000 0000 0000 0000 0000 .....
   00000030: 0100 0000 4000 3800 0100 0200 eb0b 0000 ....@.8.....
10 00000040: 0000 0000 eb0b 0000 0000 0000 3ceb c031 .....<..1
   00000050: 4850 5550 5059 5350 5950 5359 5050 5550 HPUPPYSPYPSYPPUP
12 00000060: eb18 9090 9090 9005 0f95 b640 20e6 c148 .....@..H
   00000070: c689 0fb2 c789 0000 0001 b801 0000 0089 .....
14 00000080: c7b2 0f89 c648 cle6 2040 b695 0f05 9090 .....H..@.....
   00000090: 9090 9018 eb50 5550 5059 5350 5950 5359 .....PUPPYSPYPSY
16 000000a0: 5050 5550 4831 c0eb 3c00 0000 0000 000b PPUPH1..<.....
   000000b0: eb00 0000 0000 000b eb00 0200 0100 3800 .....8.
18 000000c0: 4000 0000 0100 0000 0000 0000 0000 0000 @.....
   000000d0: 0000 0000 1c00 0000 0100 0000 0400 0000 .....
20 000000e0: 0100 3e00 0234 eb90 9090 b03c 4831 ff0f ..>..4.....<H1..
   000000f0: 0546 4c45 7f .FLE.

22 Reversing...
   Executing binary in reverse...
   PUPPYSPYPSYPPUP
24 00000000: 7f45 4c46 050f ff31 483c b090 9090 eb34 .ELF...1H<....4
   00000010: 0200 3e00 0100 0000 0400 0000 0100 0000 ..>.....
26 00000020: 1c00 0000 0000 0000 0000 0000 0000 0000 .....
   00000030: 0100 0000 4000 3800 0100 0200 eb0b 0000 ....@.8.....
28 00000040: 0000 0000 eb0b 0000 0000 0000 3ceb c031 .....<..1
   00000050: 4850 5550 5059 5350 5950 5359 5050 5550 HPUPPYSPYPSYPPUP
30 00000060: eb18 9090 9090 9005 0f95 b640 20e6 c148 .....@..H
   00000070: c689 0fb2 c789 0000 0001 b801 0000 0089 .....
32 00000080: c7b2 0f89 c648 cle6 2040 b695 0f05 9090 .....H..@.....
   00000090: 9090 9018 eb50 5550 5059 5350 5950 5359 .....PUPPYSPYPSY
34 000000a0: 5050 5550 4831 c0eb 3c00 0000 0000 000b PPUPH1..<.....
   000000b0: eb00 0000 0000 000b eb00 0200 0100 3800 .....8.
36 000000c0: 4000 0000 0100 0000 0000 0000 0000 0000 @.....
   000000d0: 0000 0000 1c00 0000 0100 0000 0400 0000 .....
38 000000e0: 0100 3e00 0234 eb90 9090 b03c 4831 ff0f ..>..4.....<H1..
   000000f0: 0546 4c45 7f .FLE.

40 Comparing hashes...
42 c082d226c96b7251649c48526dd9766071fa5e59 ns.bggp
   c082d226c96b7251649c48526dd9766071fa5e59 ns.bggp.R

```

Figure 11: Executing the palindrome backward and forward.

```

1  BITS 64
   org 0x100000000 ; Where to load this into memory
3  ;-----|-----|-----|-----|-----|
   ; ELF Header struct | OFFS | ELFHDR | PHDR | ASSEMBLY OUTPUT
5  ;-----|-----|-----|-----|-----|
   db 0x7F, "ELF" ; 0x00 | e_ident | | | 7f 45 4c 46
7  _start:
   add eax,0x4831ff0f ; 0x4 | | | | 05 0f ff 31 48
9   cmp al,0xb0 ; 0x9 | | | | 3c b0
11  nop ; 0xB | | | | 90
11  nop ; 0xC | | | | 90
11  nop ; 0xD | | | | 90
13  jmp hjmp ; 0xE | | | | eb 34
   ;-----|-----|-----|-----|-----|
15 ; ELF Header struct ct. | OFFS | ELFHDR | PHDR | ASSEMBLY OUTPUT
   ;-----|-----|-----|-----|-----|
17 dw 2 ; 0x10 | e_type | | | 02 00
17 dw 0x3e ; 0x12 | e_machine | | | 3e 00
19 dd 1 ; 0x14 | e_version | | | 01 00 00 00
21 dd _start - $$ ; 0x18 | e_entry | | | 04 00 00 00
   ;-----|-----|-----|-----|-----|
23 ; Program Header Begin | OFFS | ELFHDR | PHDR | ASSEMBLY OUTPUT
   ;-----|-----|-----|-----|-----|
25 phdr:
   dd 1 ; 0x1C | ... | p_type | | 01 00 00 00
25 dd phdr - $$ ; 0x20 | e_phoff | p_flags | | 1c 00 00 00
27 dd 0 ; 0x24 | ... | p_offset | | 00 00 00 00
27 dd 0 ; 0x28 | e_shoff | ... | | 00 00 00 00
29 dq $$ ; 0x2C | ... | p_vaddr | | 00 00 00 00
29 ; 0x30 | e_flags | ... | | 01 00 00 00
31 dw 0x40 ; 0x34 | e_shsize | p_addr | | 40 00
31 dw 0x38 ; 0x36 | e_phentsize | ... | | 38 00
33 dw 1 ; 0x38 | e_phnum | ... | | 01 00
33 dw 2 ; 0x3A | e_shentsize | ... | | 02 00
35 ; dq 2 ; 0x3C | e_shnum | p_filesz | | 02 00 00 00 00 00 00 00
35 dw 0x0beb ; eb 0b ; Overwrites e_shnum and p_filesz
37 dw 0
37 dd 0
39 hjmp:
   ; dq 2 ; 0x44 | | p_memsz | | 02 00 00 00 00 00 00 00
41 jmp sec0 ; eb 0b ; Overwrites p_memsz
   dw 0
43 dd 0
   ; dq 2 ; 0x4C | | p_align | | 02 00 00 00 00 00 00 00
45 ;-----|-----|-----|-----|-----|
   ; Outer bounds of executable portion
47 cmp al, 0xeb ; 3c eb ; Overwrites p_align
47 db 0xc0 ; c0
47 db 0x31 ; 31
49 db 0x48 ; 48
51 sec0:
   push rax ; 50
51 push rbp ; 55
53 push rax ; 50
53 push rax ; 50
55 pop rcx ; 59
55 push rbx ; 53
57 push rax ; 50
57 pop rcx ; 59
59 push rax ; 50
59 push rbx ; 53
61 pop rcx ; 59
61 push rax ; 50
63 push rax ; 50
63 push rbp ; 55

```

```

65     push    rax                ; 50
      jmp    sec1                ; eb 18
67     nop                      ; 90
      nop                      ; 90
69     nop                      ; 90
      nop                      ; 90
71     nop                      ; 90
      add    eax,0x40b6950f      ; 05 0f 95 b6 40 ; Third byte is str offset
73     and    dh,ah              ; 20 e6
      ror    DWORD [rax-0x3a],0x89 ; c1 48 c6 89
75     dd    0x89c7b20f          ; 0f b2 c7 89
      add    BYTE [rax],al       ; 00 00
77     add    BYTE [rcx],al      ; 00 01
      ;—— split - the first byte is shared with the mov rax,1
79     sec1:
      mov    rax, 1              ; b8 01 00 00 00
81     mov    edi, eax           ; 89 c7
      mov    dl, 15              ; b2 0f
83     mov    esi, eax           ; 89 c6
      shl    rsi, 0x20           ; 48 c1 e6 20
85     mov    sil, 0x95         ; 40 b6 95
      syscall                    ; 0f 05
87     nop                      ; 90
      nop                      ; 90
89     nop                      ; 90
      nop                      ; 90
91     nop                      ; 90
      sbb   bl, ch               ; 18 eb
93     sec2:
      push   rax                ; 50
95     push   rbp                ; 55
      push   rax                ; 50
97     push   rax                ; 50
      pop    rcx                ; 59
99     push   rbx                ; 53
      push   rax                ; 50
101    pop    rcx                ; 59
      push   rax                ; 50
103    push   rbx                ; 53
      pop    rcx                ; 59
105    push   rax                ; 50
      push   rax                ; 50
107    push   rbp                ; 55
      push   rax                ; 50
109    xor    rax, rax           ; 48 31 c0
      jmp    rstart             ; eb 3c
111    ;—— Header Mirror      ; old offset |
      dd    0
113    dw    0
      dw    0xeb0b              ; 0x44 |          | p_memsz | 02 00 00 00 00 00 00 00
115    dd    0                  ;
      dw    0                  ;
117    dw    0xeb0b              ; 0x3C | e_shnum   | p_filesz | 02 00 00 00 00 00 00 00
      db    0                  ;
119    db    2                  ; 0x3A | e_shentsize | ...     | 02 00
      db    0                  ;
121    db    1                  ; 0x38 | e_phnum     | ...     | 01 00
      db    0                  ;
123    db    0x38               ; 0x36 | e_phentsize | ...     | 38 00
      db    0                  ;
125    db    0x40               ; 0x34 | e_shsize    | p_addr   | 40 00
      dw    0                  ;
127    db    0                  ;
      db    1                  ; 0x30 | e_flags     | ...     | 01 00 00 00
129    dd    0                  ; 0x2C | ...        | p_vaddr  | 00 00 00 00

```



```

131 dd 0 ; 0x28 | e_shoff | ... | 00 00 00 00
    dd 0 ; 0x24 | ... | p_offset | 00 00 00 00
    dw 0 ;
133 db 0 ;
    db 0x1c ; 0x20 | e_phoff | p_flags | 1c 00 00 00
135 dw 0 ;
    db 0 ;
137 db 1 ; 0x1C | ... | p_type | 01 00 00 00
    dw 0 ;
139 db 0 ;
    db 4 ; 0x18 | e_entry | | 04 00 00 00
141 dw 0 ;
    db 0 ;
143 db 1 ; 0x14 | e_version | | 01 00 00 00
    db 0 ;
145 db 0x3e ; 0x12 | e_machine | | 3e 00
    db 0 ;
147 db 2 ; 0x10 | e_type | | 02 00
rstart:
149 xor al, 0xeb ; 34 EB ; Jmp in reverse
    nop ; 90
151 nop ; 90
    nop ; 90
153 mov al, 0x3c ; b0 3c
    xor rdi, rdi ; 48 31 ff
155 syscall ; 0f 05
    db "F"
157 db "L"
    db "E"
159 db 0x7F ; 0x00 | e_ident | | 7f 45 4c 46

```

DJAMALUDIN MALIK
Mempersembahkan **E. ZAINAH**
ABDUL HADI ★ DJ. EFFENDI
dalam
SUPIR JSTIMEWA
memperkenalkan **M. S. PRIADY**

dengan **AMINAH HARDY**
BUS BUSTAMI ★ ROOS ITJANG
Sutradara
ISKANDAR RAIS ★ REMPO URIP

Didirikan oleh **PERSARI FILMS**

21:10 BootNoodle: A Palindromic Bootloader for BGGP

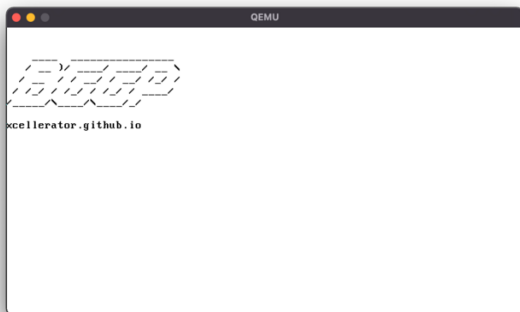
by Harvey Phillips

Recently @netspooky announced the first annual Binary Golf Grand Prix on Twitter. The objective was to create a binary of any sort that is the same forwards as it is byte-reversed, but with an emphasis on creating as small a binary as possible, hence *golfing*.

This was one of those challenges where I thought that I had no chance of creating a qualifying submission, where it might be better to just wait for the results and admire the work of others. However, it wasn't long until I found myself thinking about how it would even be possible to create such a binary. Clearly, executables that are just pure x86 instructions (like COM files) wouldn't count; otherwise, I could've just submitted 0x90 and been done!

I decided that if I was going to attempt something like this, I'd have to first settle on a file format. In the end, I think I took the easy option and chose to create an x86 bootloader palindrome. The main reasons for this were that bootloaders are essentially formatless: the only requirement for a valid bootloader is that bytes at offset 511 and 512 are 0x55 and 0xAA respectively. The rest can be just raw x86 instructions.

That brings us to the second reason: technically (as far as I am aware) the absolute minimum size of an x86 bootloader is 512 bytes. This felt like a bit of a double-edged sword, just enough space to do something interesting, but still fairly limited. Especially since it has to read the same backward!



²¹<http://xcellerator.github.io>

²²<http://www.ctyme.com/intr/int.htm>

Workflow

I knew that the first thing I had to get right was generating a palindromic file, whether or not anything really executed. The bootloader itself was going to be written in NASM, so I could then just use `dd` to snip off the first 256 bytes, reverse it with a bit of Perl from StackOverflow and `cat` the two halves together. I stuck all this into a shell script and started to get to work.

Once this is complete, we can run the bootloader with `qemu-system-x86_64 bootnoodle.bin`.

```
1 nasm -f bin -o bootnoodle.bin bootnoodle.asm
2 dd if=bootnoodle.bin of=tmp.bin \
3   bs=1 count=256
4 rm bootnoodle.bin
5 perl -0777pe '$_=reverse $_' \
6   tmp.bin >tmp2.bin
7 cat tmp.bin tmp2.bin >bootnoodle.bin
8 rm tmp*
```

x86 Bootloaders

Creating an x86 bootloader is a surprising easy task. After fighting with a few different ideas for what to do, I settled on printing a nice bit of "BGGP" ASCII art and a link to my blog, where this write-up first appeared.²¹

You might wonder how on Earth you might fit a printing function into just 256 bytes. It turns out that a huge amount of functionality, from printing characters to graphics primitives, are built into the BIOS. In our specific case, we'll be targeting the SeaBIOS that ships with QEMU. We call these built-in functions by selecting the byte we place into the AH register before invoking a particular interrupt.

For example, we can call the Teletype Output routine to print a character by placing 0x0E into AH, the ASCII character we want to print into AL and calling interrupt 0x10. There are couple of extra options to this function, like a page number and foreground colour, that we can put in to the BX register as well. In general, this is the flow of a bootloader; load registers, interrupt, load registers, interrupt, etc. We can find an exhaustive list of all these dif-

ferent routines and which registers are used at the infinitely useful x86 Interrupt Table.²²

The rough flow of execution of my bootloader is as follows:

- Clear the screen with the “scroll up window” routine (Int 10/AH=06h).
- Set the cursor to the position we want to start printing from with the “set cursor position” routine (Int 10/AH=02h).
- Load the memory address of our null-terminated string into the SI register.
- Call the string printing routine.
- Halt

The reason we have to clear the screen is that otherwise we’d have fragments of information about the BIOS cluttering the screen up. In QEMU’s case, you get the SeaBIOS copyright string stuck at the top of the screen, so for the sake of a few extra bytes, it’s nicer to clear that out. Also, one of the extra options we get with “scroll up window” is that we can change the background/foreground colour by setting BH. I opted for 0x03, which keeps the background black but makes the foreground cyan.

The only thing left as far as the actual programming goes is the `printString` function. The BIOS provides us with only a character printing function, so we have to handle the looping logic and checking for a null byte ourselves. This is all pretty standard stuff if you’ve done x86 assembly programming before.

```

printString:
2  pusha
   .loop:
4   lodsb
   test al, al
6   jz .end
   call printChar
8   call delay
   jmp .loop
10  .end:
   popa
12  ret

```

²³AH=86h, CX:DX = interval in microseconds.

²⁴<https://n0.lol/bgpp>

First, we push all the registers onto the stack with `pusha`. Entering the loop, we use `lodsb` which loads a single byte from the SI register into AL, and increments SI. We check for a NULL byte with `test al, al`, and if found, jump to `.end` where we restore the saved registers with `popa` and return. If we don’t have a NULL byte in AL, then we call the `printChar` and `delay` routines. These are less interesting and very similar to the `clearScreen` and `setCursor` routines: set some registers, interrupt.

The last thing worth mentioning is why we have the call to `delay`, which uses the “wait” routine.²³ The reason is simple: introducing a 20ms delay between printing each character results in a poor man’s animation effect!

There is still one thing that we’re forgetting. Earlier, I mentioned that a bootloader, while being exactly 512 bytes long, *must* finish with bytes 0x55 0xAA. Because we’re creating a palindrome, this means that our binary must *start* with 0xAA 0x55. Execution starts at offset 0, so we cannot avoid executing these two bytes as the first instructions.

```

aa ; stosb es:di
2 55 ; push bp

```

The `stosb` instruction is similar to `lodsb`; it stores whatever is in AL in DI, ignoring segment registers as they aren’t really relevant to this discussion. We don’t care about this because, (1) we aren’t using DI and (2) we’re about to load AL with the address of our string, so whatever happens to be loaded in AL beforehand is irrelevant to us. (It’s probably just a null byte, but that might vary from BIOS to BIOS.)

Clearly, `push bp` also doesn’t matter to us. In theory, we should clean up the stack later by `poping bp`, but seeing as we’re halting into infinite loop with the `jmp $` instruction after printing our string, it really doesn’t matter either.

So, thankfully we don’t need to worry about these two extra instructions. Merely starting the source file with `db 0xaa, 0x55` before going straight into the `_start` entrypoint is enough to get us out of trouble.

Palindrome Time

So far, we've only used up 0xf5 bytes of the 0x1ff available to us. This means that when we run our build script, we end up with a 512-byte binary that's reflected about the 256-byte boundary, with a patch of 20 NULL bytes positioned neatly in the middle. While we technically have a palindrome, Netspooky is way ahead of us, as can be seen by the stipulation in the rules on the contest page.²⁴

An easy solution would be to just have the binary end, and append the binary backwards at the end of the original file. Because of this, in order to qualify for entry, your binary must at a minimum execute > 50% of the bytes in your binary, and must execute past the halfway mark in your binary as well.

So far, we just about meet the 50%+ byte execution mark thanks to the 0xAA 0x55 bytes at the very beginning. Unfortunately, we don't yet execute past the halfway mark, so we've got to do some thinking.

We've still got to do something a little more interesting than just producing a bootloader in under 256 bytes and flipping it back on itself. There's not a huge amount we can do about the data part of the binary (which makes up about 63% of all the bytes) unless the text itself is symmetric, which it isn't. Besides, the rule above specifically mentions that *execution* has to pass the 50% mark. That leaves us to look at what can be done with the code.

My idea was to purposefully reverse portions of the code in the upper half, so that they are reversed in the lower half. This means that I also need to fix the `call` offsets manually because NASM won't be able to calculate them for me.

I used Ghidra as a disassembler, but you might want to use `objdump` as a slimmer alternative.²⁵ Ghidra makes it easy to compare the NASM source with the disassembled bytes. Because my routines are all quite short, I just wrote in the bytes next to the functions that I wanted to reverse. These were chosen alternately so the execution jumps around as much as possible, and lives up to its *noodly* name. For example, `clearScreen` looks like this.

²⁵`objdump -D -b binary -mi386 -M intel,addr16 bootnoodle.bin`

²⁶<https://www.felixcloutier.com/x86/call>

```
clearScreen:
2  pusha                ; 60
   mov ah, 0x06         ; b4 06
4  xor al, al           ; 30 c0
   mov bh, 0x03         ; b7 03
6  xor cx, cx           ; 31 c9
   mov dx, 0x184f       ; ba 4f 18
8  int 0x10             ; cd 10
   popa                 ; 61
10 ret                 ; c3
```

We could have worked this out without Ghidra and just used a hex editor, but hey, Ghidra is faster and takes out the guesswork. We can replace this with the raw bytes, but in reverse order:

```
clearScreen db 0xc3, 0x61, 0x10, 0xcd, 0x18,
2             0x4f, 0xba, 0xc9, 0x31, 0x03,
             0xb7, 0xc0, 0x30, 0x06, 0xb4,
4             0x60
```

But we also have to take a look at the `_start` entrypoint where we call `clearScreen` by name. This clearly will no longer work once we comment out `clearScreen` in favour of the reversed bytes above. You can try running the build script but NASM will exit out with a load of errors.

The solution here is that we need to replace `call clearScreen` with a raw short call. As explained on Felix Cloutier's x86 instruction reference, a short (or "near") call is a call to a memory address *relative to the next instruction*, where a `ret` is expected to be encountered eventually.²⁶ This means that we can replace the line `call clearScreen` with a simple `db 0xe8, 0x00, 0x00`. This won't work yet because we haven't specified an offset, but it will let us assemble the binary and look at some bytes.

After building, we get a binary that we can disassemble again. Even after picking "x86 Real Mode" from the list of languages in Ghidra, we're left without very much. Clicking on the first byte 0xaa and pressing D kicks off the disassembly.

After the two bogus instructions caused by the reversed 0x55 0xaa signature, we immediately get two `calls`. These are to `clearScreen` and `setCursor` which appear at the top of the source file! In particular, note that the first `call` is to 0x00 0x00; this is what we need to change.



In order to know which offset to set this to, we need the address of the instruction *after* this call, which is the call to `setCursor` at `0x5`, and the address of the re-reversed `clearScreen` routine. Scrolling through the disassembly, once we cross the half-way point, Ghidra doesn't know what it's looking at. Keep going, and eventually, towards the end, you'll find another bit of disassembled code. Checking it carefully, you'll see that it matches perfectly with the disassembly of `clearScreen` above! The address of this routine is `0x1e0`. Subtracting `0x5` from this gives `0x1db`, the relative offset that we need to set our hand-made call instruction to!

Going back to the source file, we change `db 0xe8, 0x00, 0x00` to `db 0xe8, 0xdb, 0x01`. (Remember that x86 is little-endian!) Rebuilding gives us a working bootloader.

I repeated this trick for the `printChar` routine using the exact same steps as for `clearScreen`: reverse the bytes by hand, replace any calls to `printChar` with `e8 00 00`, fire up Ghidra to calculate the correct relative memory address, and fix the call by hand again.

For fun I reversed the data in the binary, too. This meant that I had to fix the line in `_start` that loads the address of the data into `SI`. This was done by reversing the data and rebuilding. (It builds fine because the code is unaffected; running it will just print the string backwards.) Then, using a hex ed-

itor, I looked for the start of the re-reversed string, and found it at `0x10a`.

The instruction for moving into `SI` is `0xbe` followed by a memory location or register, as described back in the x86 Instruction Reference.²⁷ However, there is one caveat and it involves the very first line of the source file. Here, I've put `org 0x7c00`, which tells NASM that we are expecting our bootloader to be loaded to memory address `0x7c00` before being executed.

The reason for this seemingly arcane choice of load address is that 1024 bytes after `0x7c00` is `0x8000`, which is where the kernel is normally loaded. The usual purpose of a bootloader is to simply load the kernel from a hard disk (or other storage device) into memory address `0x8000` and then `jmp` to it. Seeing as a bootloader has to be 512 bytes in size, it makes sense to always load it in the memory region immediately prior to where the kernel will be copied to.

For a nicely commented example of loading the kernel into memory and passing execution to it, check out my ThugBoot project.²⁸ If you've been following this article, then you shouldn't have any issue reading the NASM source there. For a more in-depth read, @0xax's incredible book *Linux Insides*, which goes into infinitely more detail.²⁹

Anyway, all this means for us is that we have to add `0x7c00` to the address within the file of the re-reversed string we want to print, so we end up with a final address of `0x7d0a`, and our manual `mov` instruction becomes `db 0xbe, 0x0a, 0x7d`.

And with that, the palindromic bootloader is done! Source code is available by github, attached to this PDF and on page 54.³⁰

I'd like to thank the good folks at ThugCrowd for being so encouraging and inspirational. It was there that I first discovered my interest in exploring x86 bootloaders that lead to the ThugBoot project. I'd also like to thank Netspooky in particular for starting this competition and I highly recommend taking part next year!

²⁷<https://www.felixcloutier.com/x86/movsx:movsxd>

²⁸<https://github.com/xcellerator/thugboot>

²⁹<https://0xax.gitbooks.io/linux-insides/>

³⁰`git clone https://github.com/xcellerator/bootnoodle || unzip pocorgtfo21.pdf`

21:11 Windrose Fingerprinting of Code Architecture

by EVM

Often we come across a firmware from a device that we don't have in hand, and don't know anything about beyond pictures or sales glossies on a vendor website. We'd like to be able to load this firmware into a disassembler and analyze it anyway.

ELF firmware files will happily tell you and your disassembler the CPU architecture, but what do we do when analyzing a flat binary firmware file? We need a method to determine the architecture by comparing the file to previous samples from known architectures.

Each processor architecture has a unique byte histogram fingerprint, which others have described previously. This is because in machine code some types of opcodes are used more frequently than others (Register/memory move, comparison, jump/branch/call are usually the most common.) This gives each architecture a unique balance of bytes reflecting the designer's choice of representation of common and uncommon opcodes.

What I'm adding to the toolbox here is the concept of visualizing byte histograms as a windrose diagram. Byte histograms can be compared using a chi-squared test, but windrose diagrams may allow for a more-nuanced, visual comparison.

The following diagrams were generated from samples, mostly Linux kernels and Busybox binaries, and the occasional random large firmware file. Linux kernels and Busybox binaries work well because they are very large and contain a mix of lots of different kinds of code.

Here is a Python script that outputs a windrose diagram for a sample that you can compare against the fingerprints shown. This code bins the bytes in groups of four for more readable diagrams, and ignores bytes 0x00, 0x40, 0x80 and 0xC0 (to avoid over-representing top address bytes). Note that for best results you need to only map the text section of a binary, and remove any padding. Normally in a flat firmware binary the text section appears before the data section, and depending upon where you make the cut, your mileage may significantly vary on very small binaries.

As an example, Figure 12 presents windrose diagrams from the `.text` section of two 32-bit MIPS binaries. These are the first two MIPS binaries in the ALLSTAR dataset³¹ whose `.text` section is greater than 64KB, `7kaa` from Debian's `7kaa` package, and `jmdlx` from Debian's `aajm` package. Notice their largest three spikes (the 0x00, 0x20, and 0x8C bins) match the 32-bit MIPS fingerprint well. The double spike (0x20 and 0x24 bins) appears in all three prints. `jmdlx` has a shorter spike at 0x00, and a longer spike at 0x08, but we can still easily see that its best match is 32-bit MIPS.

THE HQ-140-X...



"A Ham's Dream"
—says W4VPU

After trying out his new Hammarlund HQ-140-X receiver, Harry H. Harris, Jr., of Charlottesville, Va., W4VPU commented, "This is truly a Ham's dream."

Creating 'dream' equipment for hams is the Hammarlund goal. How well this goal has been achieved is proven by the enthusiastic comments received from satisfied Hams. They appreciate the little extras in design, circuitry and construction built into every Hammarlund product.

For example, the HQ-140-X—the amateur receiver built to professional standards—is rated XFB by Hams everywhere because of its—

FREQUENCY STABILITY — less than .01% frequency drift after warmup anywhere from 540 Kc. to 31 Mc.

EXTREME SELECTIVITY — sharp signal separation even in the most crowded bands.

LOW NOISE LEVEL — a noise limiter that really works.

RUGGED CONSTRUCTION — built for easy use for many years.

The HQ-140-X is available either as a cabinet model or for rack mounting. For complete details, write to The Hammarlund Manufacturing Co., Inc., 460 West 34th Street, New York 1, New York. Ask for Bulletin R-3.



³¹<https://allstar.jhuapl.edu>

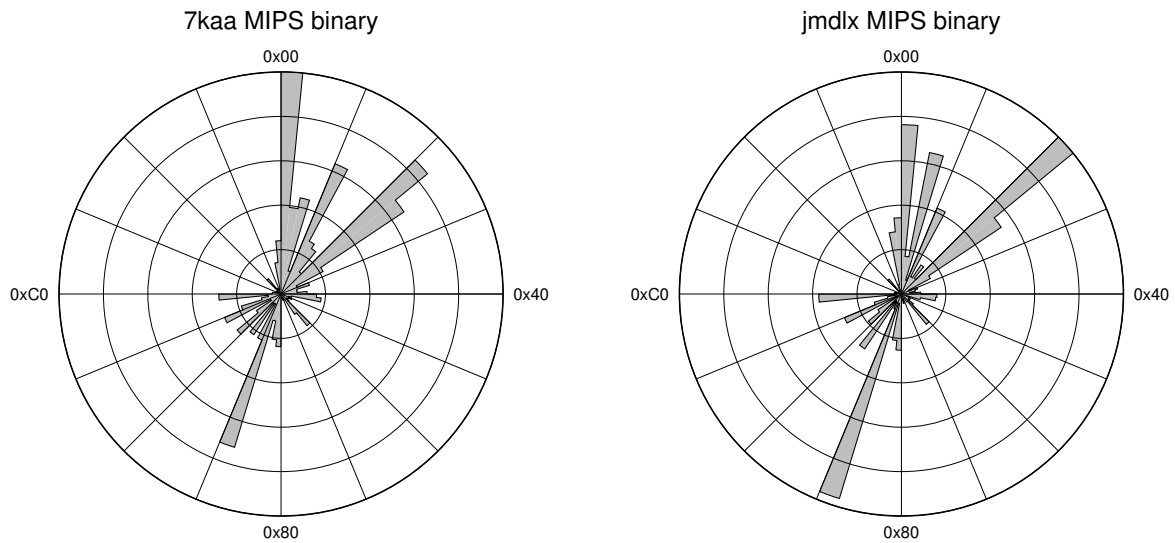
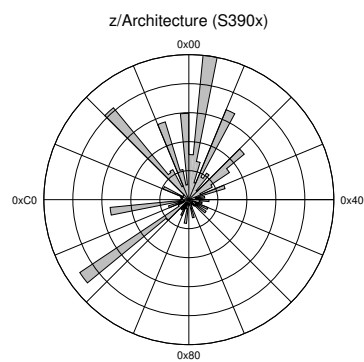
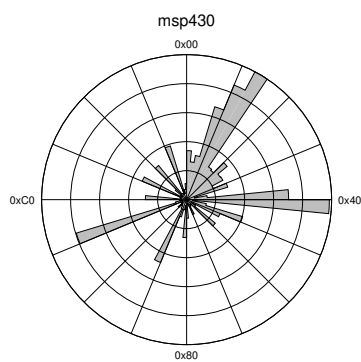
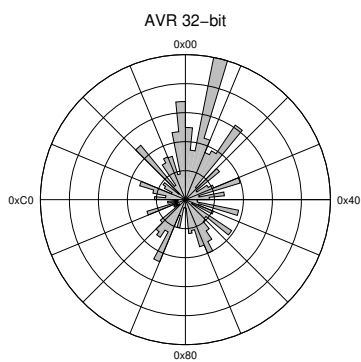
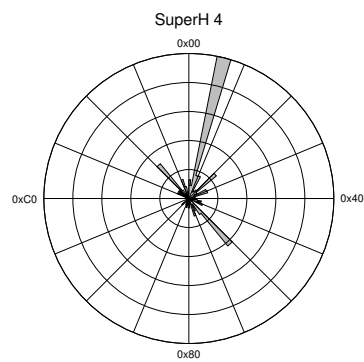
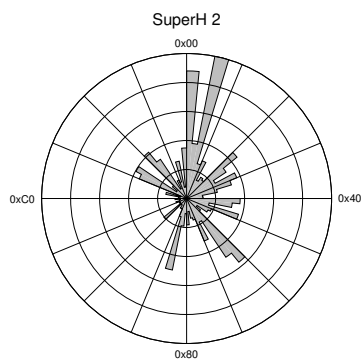
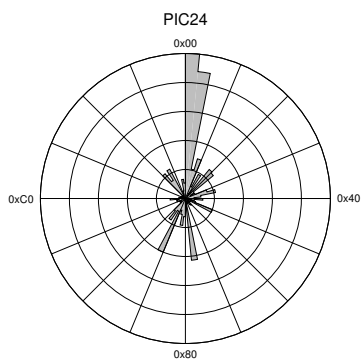
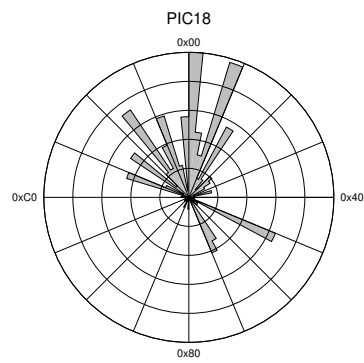
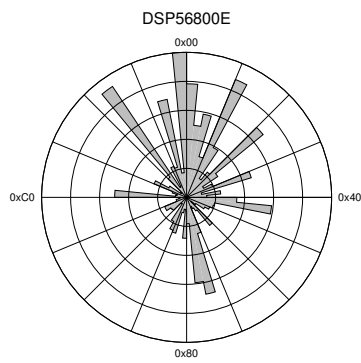
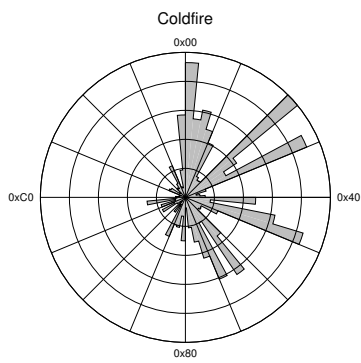
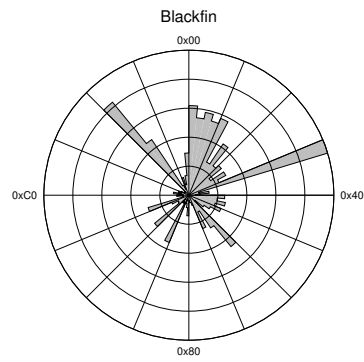
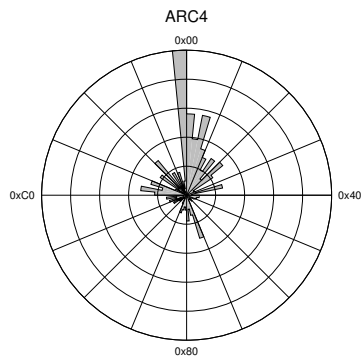
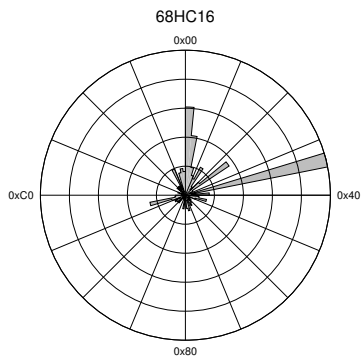


Figure 12: Fingerprints of two sample MIPS executables.

```

#!/usr/bin/python
2 import sys
  import struct
4
  fname = sys.argv[1]
6
  bytes=[]
8  entries=[]
  total=0
10
  for i in xrange(0,256):
12     bytes.append(0)
14
  with open(fname,'rb') as f:
    while True:
16         b=f.read(1)
        if b=="":
18             break
        bint = struct.unpack('<B',b)[0]
20         bytes[bint]+=1
22
  for i in xrange(0,256,4):
    entry=0
24     for j in xrange(0,4):
        if (((i+j) % 0x40) != 0):
26             entry+=int(bytes[i+j])
        entries.append(entry)
28     total+=entry
30
  for i in xrange(0,0x40):
    print "%f,%f" % (100*entries[i]/1.0/max(entries), i*360/1.0/0x40)

```

21:12 NSA's Backdoor of the PX1000-Cr

by Stefan Marsiske

I was supposed to be doing paid work, porting some silly crypto protocol to browsers. This implied getting dirty with JavaScript, the insanity of fast changing and incompatible browser interfaces, and other nasty beasts. Instead, I remembered an exciting device from a Crypto Museum exhibition. Behold the incredible PX 1000 Cr!

This diabolical pocket telex (an antique peer-to-peer messaging thingy) from 1983 had a unique feature: it came with DES encryption and was marketed toward small companies and journalists. According to some rumors, even the Dutch government used some.

This freaked out the NSA, who sent an emissary to buy up all the stock from the market and pressured Philips to suspend sales of any such infernal devices. In '84, the NSA provided Philips with an alternative encryption algorithm, which they were happy to sell to the public. The astute reader, being knowledgeable about the NSA's backdooring efforts, should immediately suspect that the new firmware might be weird in some ways. I certainly suspected a little mischief.

³²[unzip pocorgtfo21.pdf brucker-thesis.pdf](#)

Exposition

Luckily, the fine people of the Crypto Museum have not only dedicated a couple of pages to this device, but they also published ROM dumps of both the original DES-enabled device as well as the agency-tainted device. They also published Ben Brücker's bachelor thesis,³² in which he reverse engineered much of the DES variant of the device.

Although his thesis did not contain much source code, it was enough of a head start that I could dive directly into the encryption code.

My first steps were a mistake. I could not resist the irony of using a tool from Fort Meade to break their own backdoor, so I loaded the ROM into Ghidra and started annotating memory addresses. Unfortunately, Ghidra does not support this particular CPU. Luckily, IDA Pro has excellent support for this chip.

The CPU in question is a Hitachi HD6303. It's a derivative of the Motorola 6800, which was a simple but, for its time, powerful 8-bit processor. It only has four 16 bit registers: a stack pointer, an instruction pointer, an index register to address memory,



Text Lite PX1000, Photo from the Crypto Museum

and an accumulator. The latter can be accessed as a 16 bit register or as two 8-bit registers. The instruction set is simple, but certainly capable of doing great things. Turns out the same CPU was also used in the venerable Psion II Personal Digital Assistant. This means there are fans of this device who document, for example, the instruction set.³³

The fine people of the Crypto Museum also published an undated photocopied and scanned version of the CPU datasheet.³⁴ It took me a few days to progress from realizing that some pages are missing, to realizing that only the even-numbered pages are missing, to realizing that the even-numbered pages start half-way into the document in decreasing order. All hints that the pages have been scanned while holding discordian principles high. Hail Eris, indeed!

Having all this supporting information available made it an easy effort to work my way from the binary dump to an equivalent algorithm written in C. However, there were some weird things. For example, the encryption function starts by decrementing the pointer to the plaintext by one. Why? And where does that preceding byte come from, and will people be offended if I index a C char array with `-1`? Answering these questions meant I had to either reverse engineer other parts of the ROM that were unrelated to the cryptographic algorithm—which I am too lazy to do—or I had to find another way. Turns out the Psion II fans also created an emulator: SIM68xx³⁵ by Felix Erckenbrecht and Arne Riiber.

One of the most active contributors to this fine piece of software is Mayer Gabor, a Hungarian name. As I've lived and founded a hackerspace in that fine country, it wasn't hard to confirm that this contributor is a regular in such circles. After a friendly chat on IRC, he also became interested in the ROM dumps, but—just like Ben Brücker—he focused on the DES version. I complained about the plaintext array that is indexed by `-1`, and that it would be easy to figure out with a proper emulator.

³³<https://www.jaapsch.net/psion/mcmnemal.htm>

³⁴`unzip pocorgtfo21.pdf hd6303rp.pdf`

³⁵`git clone https://github.com/dg1yfe/sim68xx || unzip pocorgtfo21.pdf sim68xx.zip`

³⁶`unzip pocorgtfo21.pdf sim68xx-px1000.zip`

`git clone https://github.com/iddq/sim68xx.git; cd sim68xx; git checkout origin/px1000`

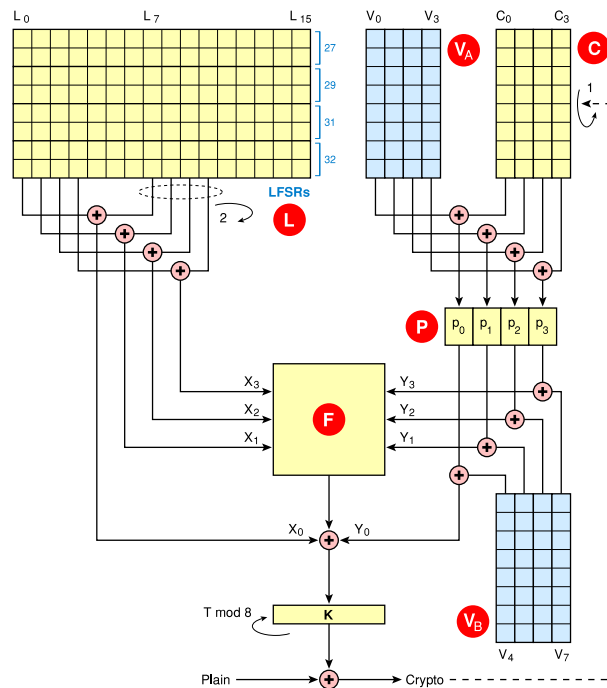
One day later, Gabor shared a branch of SIM68xx that adds support for running the PX1000 firmware, given a few minor patches for compatibility.³⁶

There it was, a working emulator with a display and keyboard. It turned out it is possible to set the text-width. The `-1` character is indeed encoding the text-width, which is limited by the size of the display to 40. It also turned out that the plaintext is also post-fixed with another character, `0x8d`, before encryption. Here, the most significant bit, which is never set in ASCII, marks the end of the string. Thus `0x8d` encodes both a newline character and the EOS.

With the working emulator I was able to verify my C interpretation of the encryption algorithm. It was finally time to start breaking the crypto!

Dramatis Personae

The algorithm itself can be shown in a simplified block diagram, helpfully provided by the Crypto Museum.



The Mysterious Key

Remember, this device is a 7-bit ASCII input device. How can someone enter an encryption key without much hassle? The engineers came up with a nice idea: Take an arbitrary 16 byte string, zero out the top nibble of each byte, and only use the lower (and slightly higher entropy) nibble, providing with a 64-bit key, which is stronger than the measly 56 bit key of DES.

Let's introduce our other main characters. In the schema, on the top left, the 16 byte block denoted **L** is supposed to be a set of four linear feedback shift registers. This is the bad guy, the end level boss. He is elusive and changes like a chameleon.

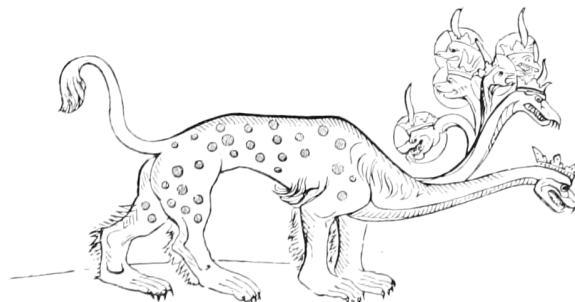
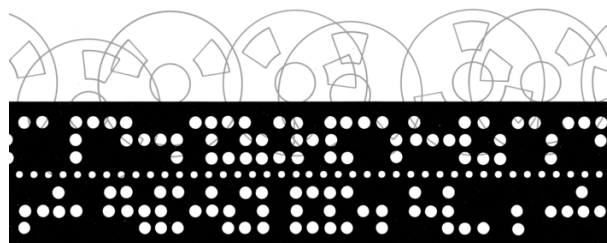
To the right we have two blue blocks, **V_A** and **V_B**, of four bytes each which contain some transformation of the encryption key. This is a supporting character, mostly stays in the background, and has little character development.

Right of **V_A** we have the four byte **C** block, which is a FIFO initially containing a transformation of parts of the encryption key, but it later becomes a cipher-feedback buffer containing the last four bytes of ciphertext. Another supporting character, this guy looks strange in the beginning, but later on becomes a familiar face we know and recognize.

The block denoted by **P** is really just a transformation which replaces each 4-bit nybble with another 4-bit nybble based on a lookup-table. This young lady is the sister of **F**, but is mostly staying predictable.

The big yellow block **F** in the middle is eight non-linear transforms that converts 6 input bits into one output bit, more on this later. This lady is another trouble-maker; she's the femme fatale of this play, working with the evil guy, making things difficult.

And last the small block **K** is a transformation of the keystream byte, that rotates the keystream byte left by the number of byte being currently encrypted modulo 8. Just another supporting character without much depth.



Act I

It is very important to see how these blocks are initialized—this is the part where the alarm bells start getting louder. During initialization, one operation comes up everywhere: the low nibble gets complemented and set as the high nibble.

```
1 //Invert low nybble into the high nybble.
  uint8_t invert2hi(uint8_t x) {
3   return ((~x) << 4) | x;
  }
```

In fact this is how 15 of the 16 bytes of the LFSR are initialized: Each low nibble of the key is taken and inflated into a byte. The last byte is set to **0xff**. As code:

```
for (i=0; i<15; i++) {
2   lfsr[i] = invert2hi(key[i]);
  }
4 lfsr[15]=0xff;
```

Now, if you happen to somehow know the internal state of the LFSR and also know how to reverse it, then it becomes trivial to check if any state has the special structure of the initial state, from which the key can be trivially recovered. I'm not sure if that actually helps, but it's ugly anyway.

Blocks **V_A**, **V_B**, and **C** are similarly initialized:

```
for (i=0; i<4; i++) {
2   V[i]   = invert2hi(key[i] ^ key[i+4]);
   V[i+4] = invert2hi(key[i+8] ^ key[i+12]);
4   C[i]   = V[i] ^ V[i+4] ^ 0xf0;
  }
```

It's not obvious at first, but if you expand **V[i]** and **V[i+4]** when setting **C[i]** and do the math, you will come to the conclusion that the values of **C** can only be one of these sixteen legal values: **0x0f**, **0x1e**, **0x2d**, **0x3c**, **0x4b**, **0x5a**, **0x69**, **0x78**, **0x87**, **0x96**, **0xa5**, **0xb4**, **0xc3**, **0xd2**, **0xe1**, or **0xf0**.

My alarm bells are kinda deafening by now, how are yours?

After the initialization, the stream cipher is ready to be used. For each key-stream byte the LFSR is mutated, then combined with the **V** and **C** blocks, fed into the **F** function, and then XOR'd into the plaintext. Let's have a look first at the mutation of the LFSR block:

```

1 for (round=0x1f; round>=0; round--) {
2   acc = 0;
3   // FAC7 in the code this loop is unrolled
4   for (i=0; i<16; i++) {
5     acc ^= lfsr[i]lookupTab[(round+i)%16];
6   } // FB43
7
8   // FB45..FB4A
9   acc = ((acc >> 1) ^ acc) & 0x55;
10
11  // tmp is twice the sequence 15..0
12  tmp=(round ^ 0xff) & 0xf;
13  lfsr[tmp] = ((lfsr[tmp]<<1)&0xAA) | acc;
14 } // FB63

```

Doesn't really look like a traditional LFSR to me, or even a set of them. But if the Crypto Museum people say so, I'm going with their insights. *Nota bene:* Those 16-bit hex numbers in the comments mark the addresses for where in the ROM this code can be found.

Normally an LFSR emits a bit after each advancement. In this code it is not obvious how this is done. The following snippet shows how four bytes are extracted from the LFSR after it has been mutated:

```

1 for (i=0; i<4; i++) {
2   tmp = lfsr[i+7]; // FB68..FB6C
3
4   // 2x rotate left FB6E..FB72
5   tmp = (tmp << 2) | (tmp >> 6);
6
7   // FB74 .. FB7A
8   lfsr_out[i] = tmp ^ lfsr[i];
9 }

```

If you squint you might imagine that there are four LFSRs but, as you will see, this doesn't matter much for our final attack. This concludes the left side of the schema before being fed into the non-linear function **F**.



On the right side of the schema you can see how **V_A** and the ciphertext FIFO are being XOR'd and mapped through **P**. It looks like this in code.

```

1 for (i=0; i<4; i++) {
2   tmp = V[i] ^ CiphertextFifo[i];
3   acc = map4to4bit[i][tmp >> 4] << 4;
4   acc |= map4to4bit[i][tmp & 0xf];
5   pbuf[i] = acc ^ V[i+4];
6 }

```

Looks straightforward, but if we unpack this in the context of encrypting the very first character (which is probably “?”, but this is irrelevant here), then we can unpack:

```
tmp = V[0] ^ CiphertextFifo[0]
```

where

```
CiphertextFifo[0] = V[0] ^ V[4] ^ 0xf0
```

which drops out V[0], and thus:

```
tmp = V[4] ^ 0xf0
```

and we know that all values of V are values where the high nibble is just the inversion of the low nibble, and if we XOR that with 0xf0, we conclude that tmp can only be one of these 16 values: 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, or 0xff.

Strange, huh? This loop runs four times, with similar results for each output byte. Later, when the Ciphertext FIFO is filled with real ciphertext, this doesn't apply anymore, but then the contents of this buffer are known, since it's the ciphertext. The mapping itself of four bits to four bits was relatively uninteresting ... or at least I couldn't immediately see anything wrong with it. And also XORing that with **V_B** was also much less exciting.

Now that we have the inputs to the **F** function, we can analyze what happens there. The code is a bit dense, we'll unpack it later:

```

1 for (i=8, acc=0; i>0; i--) {
2   // FBB9
3   for (j=1, tmp=0; j<4; j++) {
4     tmp = (tmp << 1) | (lfsr_out[j] >> 7);
5     lfsr_out[j]<<=1;
6     tmp = (tmp << 1) | (pbuf[j] >> 7);
7     pbuf[j]<<=1;
8   }
9   tmp=lookupTab6To1bit[tmp];
10
11  acc=(acc<<1) + ((tmp>>(i-1)) & 1);
12 } // 0xfbd9

```

The outer loop takes care that all eight bits of each input of the six input bytes get used in **F** and that the output of **F** is being assembled back into one byte. The inner loop interleaves the 6 input bits from `lfsr[1]`, `pbuf[1]`, `lfsr[2]`, `pbuf[2]`, `lfsr[3]` and finally `pbuf[3]`. The lookup table produces one bit, which in the last line is put into the correct bit-position of the accumulator. It's a pretty straightforward bit-sliced 6-byte-to-1-byte mapping. The lookup table is neat, it's 64 bytes, which is indexed by the six-bit interleaved value, and from the resulting byte the i^{th} bit is extracted. Very compact, neat.

The next steps are unspectacular, keeping in mind that `curChar` starts with `-1`:

```

2 acc ^= pbuf[0] ^ lfsr_out[0];
  // FBDF
4 tmp = (curChar + 1) & 7;
  // rotate left by tmp
6 acc = (acc << tmp) | (acc >> (8-tmp));
8 ciphertext[curChar]=plaintext[curChar]^acc;

```

For decryption, note that only this last line needs to be swapped around.

One last step is needed before we can loop back to mutating the LFSR, and that is advancing the ciphertext FIFO, now that there is a ciphertext byte. Again, this is pretty straightforward, and after four ciphertext bytes, the peculiar structure noted above of the initial four bytes in this FIFO is lost:

```

  // FC05
2 CiphertextFifo[4] = ciphertext[curChar];
  for(i=0; i<4; i++) {
4   // rotate left
   CiphertextFifo[i] =
6   (CiphertextFifo[i+1] << 1) |
   (CiphertextFifo[i+1] >> 7);
8 } // FC15

```

A small optimization is that the array holding the FIFO is actually five bytes, and the newest ciphertext byte is always added to the fifth position, which enables this compact loop updating the four effective items in this FIFO.

If there are more plaintext bytes to encrypt, then the algorithm loops back to mutating the LFSR. Otherwise, everything is done

ACT II: Climax

This all looks a bit fishy, but how does one actually *break* this scheme? Well for a long time I focused on somehow figuring out the LFSR and how it can be decomposed in four LFSRs of 32, 31, 29, and 27 bit lengths as indicated on the Crypto Museum schema. Many hours were wasted into slicing and dicing the LFSR, mutating it, slicing and dicing it again, writing bit level differs, staring at colored bits, throwing Berlekamp-Massey at it, trying to write my own 32/31/29/27 bit LFSRs and seeing if I could somehow slice-'n-dice a state from the big one into the ones I implemented. It was a nightmare of dead ends, failure, and despair. Boredom started to set in, and I started to ask friends if maybe they could figure out how this works. They said it's easy, but they do not have time for this now. Anyway, maybe this is an LFSR or even four, but I was unable to figure out how.

I also started to consult the bible of cryptanalysis, Antoine Joux's masterpiece: *Algorithmic Cryptanalysis*. It has a chapter, *Attacks on Stream Ciphers*, about LFSRs hidden behind a non-linear function **F**. Antoine calls these *filtered generators*:

The *filtered generator* tries to hide the linearity of a core LFSR by using a complicated non-linear output function on a few bits. At each step, the output function takes as input t bits from the inner state of the LFSR. These bits are usually neither consecutive, nor evenly spaced within the register.

Bingo! Exactly what I've been staring at for days now: the *big guy* and the *femme fatale*. The chapter mostly covers correlation attacks, but at the end there is also mention of algebraic attacks, the latter giving me a warm fuzzy feeling. Algebra is elementary school stuff, I can do that!

Antoine goes on:

The function f is usually described either as a table of values or as a polynomial. Note that, using the techniques of Section 9.2, f can always be expressed as a multivariate polynomial over \mathbb{F}_2 .

The technique in section 9.2 is called the *Möbius transform* which is used to calculate the *Algebraic Normal Form* (ANF) of a boolean function. I tried to implement the *Möbius transform* as given

in algorithm 9.6 in Joux' masterpiece, but the results were not providing the expected outputs as the lookup table. After reading a bunch of papers on algebraic normal forms, I learned that different disciplines call this different names, such as

- ANF Transform (ANFT),
- Fast Möbius Transform,
- Zhegalkin Transform, and
- Positive Polarity Reed–Muller Transform.

Valentin Bakoev's excellent paper *Fast Bit-wise Implementation of the Algebraic Normal Form Transform*³⁷ went into much more detail than Joux on this topic, and an implementation of Bakoev's Algorithm 1 gave the expected results.

```

2 void moebius(uint8_t *f, int n) {
3     int blocksize=1;
4     for(int step=1; step<=n; step++) {
5         int source=0;
6         while(source < (1<<n)) {
7             int target = source + blocksize;
8             for(int i=0; i<blocksize; i++) {
9                 f[target+i]^=f[source+i];
10            }
11            source+=2*blocksize;
12        }
13        blocksize*=2;
14    }
}

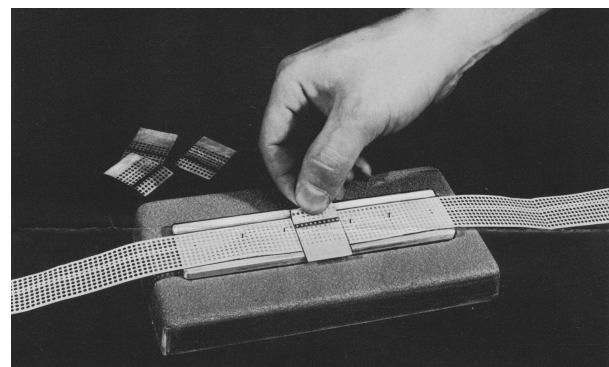
```

We can split up the original **F** lookup-table bit-by-bit.

```

2 static uint8_t lookupTab6To1bit[64]={ // at 0xFE9B
3     0x96, 0x4b, 0x65, 0x3a, 0xac, 0x6c, 0x53, 0x74,
4     0x78, 0xa5, 0x47, 0xb2, 0x4d, 0xa6, 0x59, 0x5a,
5     0x8d, 0x56, 0x2b, 0xc3, 0x71, 0xd2, 0x66, 0x3c,
6     0x1d, 0xc9, 0x93, 0x2e, 0xa9, 0x72, 0x17, 0xb1,
7     0xb4, 0xe4, 0xa3, 0x4e, 0x27, 0x5c, 0x8b, 0xc5,
8     0xe8, 0x95, 0xe1, 0xd1, 0x87, 0xb8, 0x1e, 0xca,
9     0x1b, 0x63, 0xd8, 0x2d, 0xd4, 0x9a, 0x99, 0x36,
10    0x8e, 0xc6, 0x69, 0xe2, 0x39, 0x35, 0x6a, 0x9c
};

```



³⁷unzip pocorgtfo21.pdf bakoev-afn.pdf

Feeding it into the moebius function, we get this.

```

2 f0= 01100010011010101011100011101011
3     001010110111110001101001000101100
4 g0= 01100101000011111011011101001001
5     01011011010010010011000110001110
6 f1= 11010010001101010111011000110110
7     00111010000010111100010111010010
8 g1= 101110001001111011001001110001011
9     10010110101111010100100111111110
10 f2= 10101101011011001100001110010010
11     11011101010010100001100111000101
12 g2= 11000111101010110110111111101110
13     01110111010000101100000010110110
14 f3= 010111001000101111010000111011000
15     00010110100001111011011010101011
16 g3= 0100111010111100100000111000101
17     01011001010100110100111100110000
18 f4= 10010011100100110100110110100111
19     10000100010101101010111100001101
20 g4= 1110110000000001011001010010101
21     00010110101110001000110011100110
22 f5= 00111101110101000010101100011101
23     11101000101001000101000100111110
24 g5= 00101001100101110001011110110011
25     1011111110010001100010010000010
26 f6= 01100111101010110101111001000100
27     0101010101100010111000001110010
28 g6= 0110000101000000010110010111101
29     00100001001111000000010100111100
30 f7= 10001000010101001001010001101001
31     1110001111111010010111011010001
32 g7= 1111000010110001000101100110010
33     01101011101010011011101010101010

```

The output of the Möbius transform is just another lookup table, a boolean function with exactly the same amount of input parameters as the original non-linear function. Using this it is possible to create the ANF of the non-linear function:

$$f(x_0, \dots, x_{n-1}) = \bigoplus_{(a_0, \dots, a_{n-1}) \in \mathbb{F}_2^n} g(a_0, \dots, a_{n-1}) \prod_i x_i^{a_i}$$

In this equation the $g(\dots)$ coefficient is the output of the Möbius transform, and since these bits are either 0 or 1, we can eliminate around half of all terms.

By inputting the fx/gx pairs we obtained from the moebius function into the following Python beauty, we can construct the ANF.

```

1 ^ ^ .join(
2     ' (+c+)'
3     for c in [
4         '&'.join(
5             f"x[{i}]"
6             for i, x in enumerate(reversed(f'{a:06b}'))
7             if x == "1")
8         for a in range(64) if moebius[a]=='1'
9     ] if c)

```

This can then be evaluated for all values between 0 and 63 and should produce the same result as the corresponding fx. If the result is the exact inverse of fx, then the ANF has an odd number of constant 1 terms, and the ANF must be fixed by prefixing it with 1 ^.

For illustration, behold the ANF of `f4`:

```

1 1 ^ (x0) ^ (x1) ^ (x2) ^ (x0&x2) ^ (x4)
   ^ (x1&x4) ^ (x0&x1&x4) ^ (x1&x2&x4)
3  ^ (x3&x4) ^ (x0&x1&x3&x4) ^ (x0&x2&x3&x4)
   ^ (x0&x1&x2&x3&x4) ^ (x0&x1&x5)
5  ^ (x0&x2&x5) ^ (x1&x2&x5) ^ (x3&x5)
   ^ (x1&x3&x5) ^ (x0&x1&x3&x5) ^ (x2&x3&x5)
7  ^ (x4&x5) ^ (x2&x4&x5) ^ (x0&x2&x4&x5)
   ^ (x3&x4&x5) ^ (x0&x3&x4&x5)
9  ^ (x1&x3&x4&x5) ^ (x1&x2&x3&x4&x5)

```

Woohooo, look ma, I converted a lookup-table into algebra! I mean, I defeated the evil temptress, the femme fatale! After a few days of pondering, I also converted the lookup table marked **P** in the schema to its ANFs. Erm, I mean, I defeated the younger sister. The path to this victory was not immediately obvious, since **P** is a 4-bit to 4-bit table, and the Möbius transform only applies to boolean functions with one output bit.

The trick was to deconstruct the 4-to-4 mapping into four times 4-to-1 mappings, one for each output bit, while of course the input bits will be always the same for the same nibble. Hah! Take that, NSA! Most of your backdoor is now reduced to a bunch of polynomials!

ACT III: The Fall

But what do we do with that big guy, the end level boss, that pesky LFSR block? I kinda gave up on finding the polynomial for the LFSR, but maybe there is a different way to convert this into algebra? I've always been a big fan of Angr and symbolic execution. Maybe if I let Angr consume the loop that mutates the LFSR, I can get some symbolic constraints. Symbolic constraints being nothing other than equations. The trick was to modify the the loop to not run in place, but to output another 16 byte LFSR. Angr can then tell me, symbolically, how the output LFSR depends on the input LFSR. The (much truncated) output is promising.

```

1 <BV128 state_19_128[87:87] ^ state_19_128[63:63] ^
   state_19_128[55:55] ^ state_19_128[31:31] ^
3  state_19_128[23:23] ^ state_19_128[7:7] ^
   state_19_128[102:102] ^ state_19_128[86:86] ^
5  state_19_128[70:70] ^ state_19_128[62:62] ^
   state_19_128[54:54] ^ state_19_128[46:46] ^
7  state_19_128[30:30] ^ state_19_128[14:14] ..

```

Notice the trailing `..` in the last line. This signals concatenation of bit vectors. In total, 128 bits are being concatenated! The big guy finally reveals some weakness! Angr gave me the bits I needed to

XOR together for each bit in the next state. After running some `sed` magic on this output, I had 128 lists, with only the bit positions contributing to the next state of this bit.³⁸ Wow, this really looks like algebra, but first lets analyze this list of lists a bit more.

I was very interested how these bits are related. I wrote a recursive function, taking one bit and visiting recursively all bits that this bit depends on. My goal was to figure out if there is loops or islands in this graph. This was my recursive function:

```

1 def walk(bit, c):
   c.append(bit)
3  for b in bits[bit]:
   if b in c: continue
5  c=walk(b, c)
   return c

```

I ran it for all values from 0 to 127, discarding any duplicate results. Here are the bit indices for which I first saw a result, its length, and the result values themselves.

```

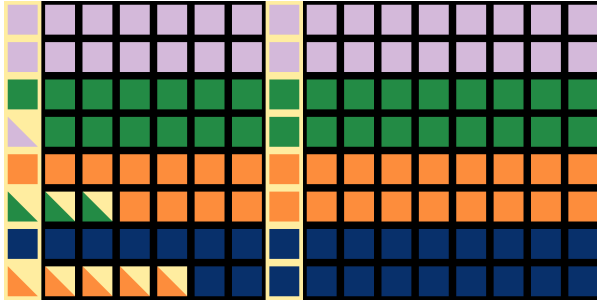
0 32 (0, 1, 8, 9, 16, 17, 24, 25, 32, 33, 40, 41, 48,
   49, 56, 57, 64, 65, 72, 73, 80, 81, 88, 89, 96,
   97, 104, 105, 112, 113, 120, 121)
4 2 31 (2, 3, 10, 11, 18, 19, 26, 27, 34, 35, 42, 43,
   50, 51, 58, 59, 66, 67, 74, 75, 82, 83, 90, 91,
   98, 99, 106, 107, 114, 115, 122)
6 4 29 (4, 5, 12, 13, 20, 21, 28, 29, 36, 37, 44, 45,
   52, 53, 60, 61, 68, 69, 76, 77, 84, 85, 92, 93,
   100, 101, 108, 116, 124)
10 6 27 (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
   54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
   102, 110, 118, 126)
12 95 28 (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
   54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
   95, 102, 110, 118, 126)
14 103 28 (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
   54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
   102, 103, 110, 118, 126)
16 109 30 (4, 5, 12, 13, 20, 21, 28, 29, 36, 37, 44, 45,
   52, 53, 60, 61, 68, 69, 76, 77, 84, 85, 92, 93,
   100, 101, 108, 109, 116, 124)
18 111 28 (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
   54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
   102, 110, 111, 118, 126)
20 117 30 (4, 5, 12, 13, 20, 21, 28, 29, 36, 37, 44, 45,
   52, 53, 60, 61, 68, 69, 76, 77, 84, 85, 92, 93,
   100, 101, 108, 116, 117, 124)
22 119 28 (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
   54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
   102, 110, 118, 119, 126)
24 123 32 (2, 3, 10, 11, 18, 19, 26, 27, 34, 35, 42, 43,
   50, 51, 58, 59, 66, 67, 74, 75, 82, 83, 90, 91,
   98, 99, 106, 107, 114, 115, 122, 123)
26 125 30 (4, 5, 12, 13, 20, 21, 28, 29, 36, 37, 44, 45,
   52, 53, 60, 61, 68, 69, 76, 77, 84, 85, 92, 93,
   100, 101, 108, 116, 124, 125)
28 127 28 (6, 7, 14, 15, 22, 23, 30, 31, 38, 39, 46, 47,
   54, 55, 62, 63, 70, 71, 78, 79, 86, 87, 94,
   102, 110, 118, 126, 127)
30
32
34
36
38

```

Whoa! The first four results are with lengths 32, 31, 29, and 27. That seems to be the source of the Crypto Museum people claiming that there are four small LFSRs hidden in there. There are also nine positions that are not contributing to the first four loops, but which themselves do depend on bits in those.

³⁸`unzip pocorgtfo21.pdf px1k.zip; unzip px1k.zip lfsr-next-bits.txt`

To make this all much clearer, I made my script draw a handy illustration of the LFSRs.



Bytes of the char array are horizontal increasing indexes left to right, bits vertical with increasing indexed top-down. The homogeneous squares constitute the four LFSRs, and the squares half-yellow depend on the LFSR of their other color. Just for reference I also framed with yellow border byte 0 and byte 7 of each LFSR block, as these are used when extracting bits as described above in the discussion of the encryption algorithm.

Interestingly, some of the orphan bits are included in extraction of entropy from the LFSR. Just to add a bit of confusion, `claripy`'s bitvector considers such char arrays as one big-endian value, which means that bit 127 is the bottom bit of byte 0 (the bottom left-most bit) and the least significant bit of the bitvector is bit 0 of byte 15, thus the top right corner of the diagram.



ACT IV: Revelation

I had everything converted to polynomials and constraints, so I started to try to feed it all directly into Z3, but Z3 seems to be geared more toward non-boolean equations. Working with vectors of booleans was quite tedious. After some long nights, I gave up and started anew in `claripy`, a wrapper around Z3 from the fine Angr people.

With `claripy`, everything went well, I had the first solution! It took nearly two minutes but, alas, it was incorrect! After a few days of debugging my constraints, I finally had the correct solution, and it only took 50 seconds! I defeated the beast! What a symbolic execution!

All you need to do is feed the solver 17 bytes of ciphertext, and the solver will either declare that the ciphertext cannot be the output of the PX1000cr algorithm, or it outputs the encryption key and the decrypted 17 bytes of plaintext. The rest of the plaintext can be recovered by decrypting the ciphertext with the recovered key. With a little change it is also possible to solve keys for shorter ciphertexts, but then there will be multiple key candidates which must be tested by the user. The number of key candidates in that case is $2^{17-\min(\text{len}(\text{ciphertext}),17)}$.

Looking at my script I realized I could keep everything symbolic, pre-computing all constraints, and with this change a speed-run is possible. With this, calculating the solution takes now less than four seconds!

I invite everyone to download the emulator and run the ROM themselves and plug the ciphertext into the solution. With a few changes you can even calculate things backwards, like what plaintext and key combination generates the following ciphertext: “(NSA backdoor fun”.

I do not know if the NSA had a SAT solver like Z3 back in 1983, but 40 years later the fact is I can recover a key within seconds in a single thread on a laptop CPU. I am far from being able to do so if DES were used. This lets me confirm that the PX1000cr algorithm is indeed a backdoor.

Finally I would like to thank Ben, Phr3ak, the Crypto Museum people, Jonathan, Antoine, the Angr devs, Ascimoo and Dnet for their support!

21:13 Solving the Load Address; or, Fixing Useless Firmware Disassembly

by EVM

Our Objective

In this article, I present a little trick for determining the load address in a processor's memory space for a piece of embedded code when the load address is not on a nice clean boundary. Oftentimes, the load address is easy to figure out, but in certain cases it might not be, such as when Flash code is copied into RAM or when a firmware update file contains code for multiple processors.

The concept is to load the code at 0, locate all the absolute function calls, and sort them starting from the lowest address. Then choose the two lowest function addresses, f_1 and f_2 . Calculate $d = f_2 - f_1$. Scan through the functions starting from 0, look for the first pair of functions that are offset by d , and call these s_1 and s_2 . The load address is then determined by calculating $f_1 - s_1$.

The reader convinced of the need for such a technique will desire to know *why* this works, we'll cover that in a moment. First, we'll remind other readers how much of a pain finding a firmware load address can be.

Motivation; or, Why Bother?

The problem we're dealing with is determining the load address for a piece of firmware that doesn't load on a round number boundary. This problem seems to primarily arise in two situations.

The first situation is when a piece of firmware is written to run mostly from RAM. This firmware will generally have a small stub at the beginning that copies the code from Flash into RAM and then jumps to it.

The second situation is when you have a firmware update file that is used to update multiple processors on a system. (For example, a drone that's got a handful of processors but only one update process.) So we might be able to clearly identify a blob within the file as belonging to a particular CPU, but we don't know how that blob ends up loaded in the processor's address space.

Many times it's pretty easy to solve the load address by just doing a disassembly and guessing a round number as the base address for the code. For instance, you may try to load the image at address 0, which IDA does by default, and then realize that all of the absolute addresses of functions begin with 0xC0. This works most of the time because sections of memory often start on a nice round number in their address space. If we see `sub_10`, `sub_24`, `sub_3A` and then absolute references to 0xC00010, 0xC00024, and 0xC0003A, it's probably safe to assume that 0xC00000 is the load address.

However, it's not always that straightforward. (Figure 13.) What sometimes happens in Situation 1 is that the image might get loaded at some offset that's not a round number. In Situation 2, what can happen is you may not know exactly where the code begins, so it may load on a round number but you may not know exactly which part of the code gets loaded on the round number.

The main symptom of loading at the wrong load address is broken absolute calls, which results in some form of bad disassembly. Depending on the architecture or the size of the code, there may not be that many absolute calls, so the failure may be subtle. A secondary symptom of loading at the wrong address is that a disassembler will fail to recognize large sections of code as subroutines because it does not look like any code is calling functions at those addresses.

RIEGER'S Monogram Whiskey

Purity and age guarantee Good Whiskey. Rieger's Monogram is absolutely pure and wholesome. Guaranteed under the Pure Food Laws. Its exquisite, smooth, mellow flavor has made it a lasting favorite with over 100,000 satisfied customers. We are U. S. Registered Distillers (Distillery No. 350, 5th Dist. of Ky.) Why pay exorbitant prices, when you can buy Rieger's Monogram Whiskey at the regular wholesale dealer's price and save money by ordering your goods shipped direct?

WE PREPAY ALL EXPRESS CHARGES

8 Qts. RIEGER'S MONOGRAM PRIVATE STOCK	\$5.00	FREE WITH EACH ORDER
4 Qts. RIEGER'S MONOGRAM EXTRA FINE	\$3.00	Two sample bottles of Rieger's Fine Monogram Whiskey, Gold-Tipped Whiskey Glass and Patent Corkscrew.

No Marks on Packages to Indicate Contents

Send us an order, and when you get the Whiskey, test it for flavor, smoothness, and all the essentials of GOOD Whiskey. Compare it with other Whiskies (no matter what the price); test it for medicinal purposes; let your friends try it; use half of it if necessary to satisfy yourself on these points—then if you are not thoroughly convinced that "Rieger's Monogram" is as good as any Whiskey you ever drank, return the balance to us, and we will pay return charges and at once send you every cent of your money.

J. RIEGER & CO. 1512 Genesee Street KANSAS CITY, MO.

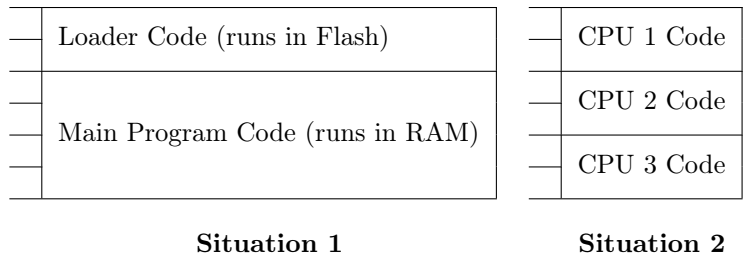


Figure 13: (Left) Situation 1: a program with an initial stub that copies the rest of the code into RAM and executes it. (Right) Situation 2: multiple pieces of code within a single firmware update file.

Our Clinical Trial

In an era where science is sometimes forgotten, it's helpful to look at a specific example as a study. For our clinical trial to determine technique efficacy, we'll look at a SH-2 microprocessor image that is an example of Situation 1. The code is initially run from Flash, but then most of the code was copied to RAM and run from there. The RAM code was loaded at a round number, but the RAM code began several functions into the binary image, which caused the disparity. I use IDA Pro for this example, but any other disassembler could just as easily be used.

Relative vs. Absolute Calls

Most processor architectures contain both absolute and relative jump and call instructions. In most programs, most of the jump and call instructions are relative. Relative means that the address of the target function is encoded within the instruction as an offset (usually from the start of the next instruction). Relative jumps and calls are position independent, meaning they can be moved around and will work correctly. They will disassemble to correctly target the right offset wherever they might be loaded.

Figure 14 shows an example of a relative call instruction (branch to subroutine) in SH-2. The processor makes the calculation: $0x401C32 + 4 + 2*(0x209) = 0x402048$, since $0x209$ is the offset in 2-byte words, and the program counter value is actually four bytes ahead of the beginning of the instruction. (This is because it is just past the branch delay slot, which I'll just skip over explaining for now!)

However, you might notice that for SH-2, the maximum offset for the branch-to-subroutine instruction is $\pm 0x7FF$. This means you cannot jump

more than 4K away from your current position, which is a problem. As such, we need to be able to specify the absolute address for a call. An absolute call (also known as a far call or far jump) specifies the full (in this case, 32-bit) address of the target function. Figure 15 shows an example of an absolute call instruction (`jsr`) in SH-2.

The processor makes the calculation $0x40083A + 4 + 2*(0xD) = 0x400870$, and this address is then dereferenced to get the full 32-bit address of the function, in this case $0x409FD8$.

You'll notice that we need to look at absolute calls for clues to where the program is loaded. From this one absolute call example above, we can see that the code is likely loaded somewhere in the area of $0x400000$.

Epson QX-10™ ?

Graphics?

Q PLOTTER

GRAPHIC POWER FOR THE EPSON QX-10™

with one key . . .

- Draw Circles ■ Squares
- Lines ■ Axis, etc.

also zooming and panning, multiple screens, animation and much more . . .

Q Plotter also turns your Mbasic into a high powered graphics basic.

introductory price **\$175.00**

To order call toll-free 1-800-824-7888 Operator 409
For technical information or dealer nearest you
Call (602) 747-0005 or Telex 709234

Metro Software Inc.
5648 E. Broadway Blvd., Dept. B, Tucson, Arizona 85711

QX-10 is a TM of Epson America • Mbasic is a TM of Microsoft

```

1 ROM:0401C32 B2 09          bsr sub_402048

```

Figure 14: An example of a relative call instruction (branch to subroutine) in SH-2.

```

1 ROM:0040083A D3 0D          mov.l #sub_409FD8, r3
ROM:0040083C 43 0B          jsr @r3 ;sub_409FD8
3 ...
ROM:00400870 00 40 9F D8     dword_400870: .data.l h'409FD8

```

Figure 15: An example of an absolute call instruction in SH-2. Note that the full 32-bit target address is embedded as data just after the instruction.

Walkthrough: Measuring the (Social) Distance

In this example, IDA did an initial disassembly with the code loaded at 0. We see the following in the Functions window:

```

sub_0
2 sub_2A
sub_34
4 sub_66
sub_8A
6 sub_9C

```

Most of the absolute calls are in the 0x400000 range and are simply broken. Rebasing the image to 0x400000 (the obvious guess) makes those calls point to valid addresses; however, the calls land at seemingly random code addresses, in the middle of functions.

So here's the trick: we ask IDA to give us all of the absolute calls by doing a text search for the instruction, such as `jsr` for SH-2. We then sort these by the instruction. IDA helpfully labels the `jsr` instructions with their target functions, so this sorts these instructions by target function, with the smallest target function at the top:

```

jsr @r2; sub_400000
2 jsr @r2; sub_400000
jsr @r2; sub_400036
4 jsr @r2; sub_400036
...

```

I'm oversimplifying here because the compiler will use different registers, so you may need to sort and then look for the low addresses used for each register. Regardless, the objective remains as locating the lowest two functions referenced by absolute calls.

Next, we choose the two smallest target functions. We calculate the difference (d) between the two functions. We then scan starting from the smallest function which IDA lists and look to see whether there is a function defined at distance d . For this example, $d = 0x36$. Scanning through the beginning of the functions list, we see `sub_66` and `sub_9C` are $0x36$ apart. This means that `sub_66` gets loaded at $0x400000$, and consequently `sub_9C` gets loaded at $0x400036$.

See Figure 16 for some code to help with this if you use IDA. In this example, `sub_0` was an entry vector in the Flash code that eventually called `sub_34`, an initialization function that copied the Flash program beginning at offset $0x66$ to $0x400000$ in RAM.

Using our newfound knowledge, we can do two things. Either we can rebase the entire image to $0x3FFF9A$, or instead we can reload the input file, creating a new segment for RAM, telling IDA to start at offset $0x66$ in the file. (Rebasing to an address beneath the section isn't technically correct, but it's quick and easy.)



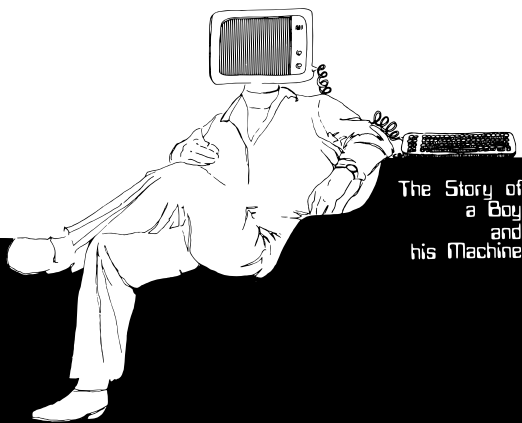
```

1 define find_offset_functions(delta):
    f = idc.get_next_func(0)
3 while (f != idc.BADADDR):
    f2 = idc.get_next_func(f)
5 while ((f2 != idc.BADADDR) and (f2 - f <= delta)):
    if (f2 - f == delta):
7         print("Functions %s and %s are 0x%x offset" %
                (idc.get_func_name(f), idc.get_func_name(f2), delta))
9         f2 = idc.get_next_func(f2)
    f = idc.get_next_func(f)

```

Figure 16: IDA Python code to print a list of functions offset by a given amount.

Manhattan Punch Line Theatre
 Steve Haglan Mich McGuire Richard Erickson Jerry Heymann
 Producing Directors
 presents
 a new play
 by
 Mike Eisenberg
HACKERS
 Directed by Jerry Heymann



The Story of
 a Boy
 and
 his Machine

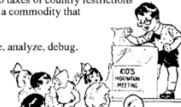
Manhattan Punch Line
 at the Lion Theatre
 422 West 42nd Street
 on Theatre Row

The Age Of Personal Reverse Engineering has arrived!

Solved: That when tongues turn white, breath feverish, stomach sour and bowels constipated, that our mothers give us tiny portions of love and sugar, we claim pills and shells in exotic architectures in order to port the thing everywhere.

No need to wait more for this to happen! The era of personal reverse engineering has finally arrived. No taxes or country restrictions involved! Free radar2 licenses is a commodity that everybody can enjoy

With radar2 you can disassemble, analyze, debug, patch any binary for a wide range of CPUs and OSs even for your shiny 4004 running PC/M!



With whom might we share this?

In addition to matching deltas between absolute function calls, this process can be similarly applied in other situations where differences between objects in the binary can be matched to differences between absolute addresses in the code.

This approach has been successfully employed on a processor with separate code and data segments to determine the load address of the data segment. In this case, a function had been identified which was presumed to be a debug print function, and it took the address of a string as the first parameter. The load address was determined by dumping all of the first parameters in calls to the debug function, and then dumping all string offsets in the file, and matching the differences in those lists.

It seems possible to automate this approach by treating the two lists of offsets as signals and running a correlation function on both of them to determine the best match. The difficulty to automation would most likely be ensuring that disassembly is clean and accurate, and that a majority of subroutines are properly identified, even when the code is loaded at the wrong address.

AT LAST... SOFTWARE THAT TEACHES READING

PAL is the only diagnostic/remediation program ever written for reading education. PAL actually diagnoses the cause of reading problems, and provides remediation directly targeted at those problems.

PAL covers the entire scope and sequence of reading education for each grade 2 through 6, and evaluates up to 40 major skills and 160 subskills per grade level.

The PAL MASTER DISK PACKAGE (required for use with the Curriculum Packages) operates the PAL system. It includes an upper/lower case chip for the Apple II, so that lessons are presented in a 'real world' format. \$99.95.

The PAL READING CURRICULUM PACKAGES provide the diagnosis and remediation. \$99.95 per grade level. A two-disk demonstration package is available for only \$9.95.

If you are uncertain about which grade level to purchase for your child, order the PAL PLACEMENT TEST (includes a \$10.00 coupon good on your next PAL purchase). \$29.95.

THE WAY TEACHERS WANT READING TAUGHT.



System Requirements: Apple II with AppleSoft, 48K RAM, one or two disk drives.
 VISA, MasterCard, check, C.O.D. accepted. Colorado residents add 3% sales tax.
 Universal Systems for Education, Inc.
 230 Academic Circle, Suite 1
 Colorado Springs, Colorado 80901
 1983 07-0475

21:14 Counting words with a state machine.

by Robert Graham

In this paper we implement `wc`, the classic Unix word count program, using an asynchronous state machine parser. We implement this twice: first a simplified version supporting ASCII, then a more complete program supporting Unicode UTF-8 encoding. We implement this algorithm in both C and JavaScript. Even the latter is significantly faster than the standard versions of `wc`, such as the GNU Coreutils `wc` that comes with Linux.

Introduction

A parser is software that translates external data into some internal data structures.

At university, they teach you abstract and formal parsers, often in a class that builds a compiler. However, little of that theory is used elsewhere in your coursework. In your networking class, the code they teach uses concrete and ad-hoc parsers, discarding everything you learned in your parser class. While parser theory they teach you is useful, even academics struggle to use it in practice.

In this paper, we do a mix of theory and practice. On one hand, we look at abstract theory of state-machines and deterministic/non-deterministic finite automata. On the other hand, we build the state-machine by hand, banging the bytes together.

The reason this concept is important is demonstrated by Nginx replacing Apache as the dominant web server on the Internet. Apache parses input the legacy way they taught you in networking class. The newer Nginx parses input using a state-machine. This parsing is more scalable, allowing much higher loads on the web server.³⁹

In this paper, we demonstrate state-machines by re-implementing the classic Unix command-line program `wc`. Over the last year, it has been popular for proponents of various languages to re-implement `wc` in order to show that their favorite language can compete with C in performance. In this case, we do this to demonstrate our favorite algorithm is better than existing algorithms, implementing it in two different languages.

³⁹This overstates the importance of just the parsing. Nginx scales better than Apache for a lot of reasons. However, these reasons are all interconnected: if you write an asynchronous server, then state-machine parsers are a much better way of parsing the requests.

These re-implementations are usually incomplete, only parsing ASCII. In this paper, we do a more complete version, correctly parsing UTF-8.

The intent of this article isn't that you should go and parse everything with state-machines. It puts a burden on future programmers trying to read the code, most of whom are unfamiliar with the technique. On the other hand, when performance and scalability are needed, state-machines are a good choice. You probably wouldn't want to use them for `wc` in the real world, as the program doesn't need to be especially fast. We choose `wc` in this paper only because it's a popular benchmark target, the simple thing that more complex endeavors are compared against.

What is WC?

This command-line utility has been part of Unix since time began on the first of January, 1970. As defined in the POSIX standard, it counts the number of lines, words and characters, when the corresponding flag of `-l`, `-w`, and `-c` is set. If no parameters are set, then the default is all three, `-lwc`.

```
2 $ echo "basic input/output" | wc
   1      2     19
```

We see here that the program has reported one line, two words, and 19 characters. Words are counted by the number of strings of non-spaces separated by spaces. Thus, this example is only two words, not three.

Modern character encodings can use multiple bytes per character, such as UTF-8 or various character sets for Chinese, Japanese, and Korean. In such cases, the `-m` parameter replaces the `-c` parameter, counting the number of multi-byte characters instead of the number of bytes. As we see in these two examples, changing from `-c` to `-m` changes the character count:

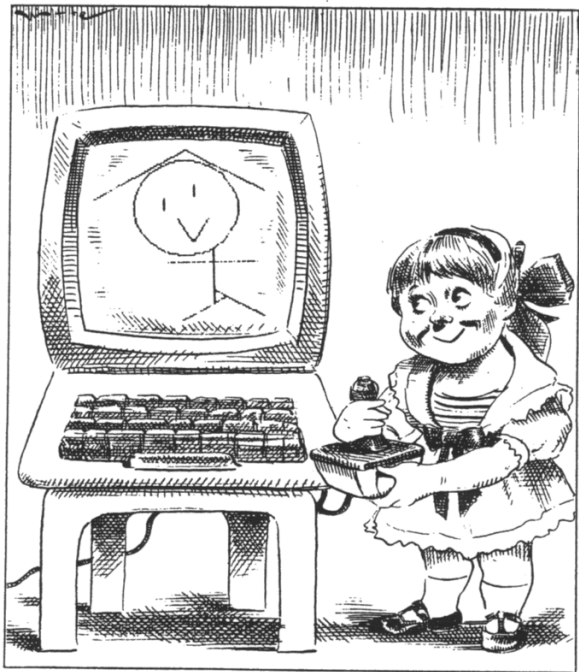
```
$ echo わたしは にほんごがすこししか はなせません | wc -lwc
   1      3     67
$ echo わたしは にほんごがすこししか はなせません | wc -lwm
   1      3     23
```

How do they implement WC?

There are many versions of the program, such as GNU's Coreutils for Linux, BusyBox, macOS, FreeBSD, OpenBSD, QNX and SunOS. Most implementations count words by counting the number of times a space is followed by a non-space. Think of it as an edge-triggered condition, going from space to non-space. As we'll soon see, this can also be treated as a state-machine with two states.

Parsing words is easy, the hard part is character-sets. We could hard-code ASCII values into our program, such as `0x20` for space and `0x0A` for newline, but this wouldn't work for non-ASCII systems. IBM mainframes that use the EBCDIC character-set will represent a space using `0x40`.

Thus, instead of using hard-coded values these programs use the standard `isspace()` function to test if a character is a space. Recently, many people have re-implemented `wc` in their favorite language to show that they can be just as fast as C. In fact, most of the processing time is spent in the `isspace()` function, so all they really proved is that hard-coded constants like `0x20` in other languages are faster than `isspace()` function calls in C.



⁴⁰The 'r' in `mbrtowc()` means "re-entrant." If parsing at the end of a fragment, it saves state before resuming at the start of the next fragment.

The problem is worse for multi-byte character-sets like UTF-8. The program must first parse multiple bytes into wide characters using functions like `mbtowc()` (or `mbrtowc()`),⁴⁰ then test if they are a space with `iswspace()`. Re-implementations often do only ASCII. This paper includes two re-implementations, the first for ASCII, the second for UTF-8.

How do we implement it?

Our first version supporting ASCII is shown in Figure 17. In the GitHub project accompanying this article, the program is `wc2o.c`, where the 'o' stands for "obfuscated C version." This program is pretty darn opaque when trying to figure out how it counts words. On the other hand, it exposes the idea of state-machine parsing.

Line 5 declares the state-machine table consisting of four states. Each state is a row of three transitions. (Table 1)

Line 7 declares a table that will translate bytes. All 256 ASCII values translate into one of three possible values: `word(0)`, `space(1)`, and `newline(2)`. Specifically, the character `0x0A` or '`\n`' translates to `newline(2)`, and the characters '`\b\t\m\v\f`' translate to `space(1)`. All other values translate to `word(0)`. The reason we include this translation step is that so that the state-machine on line 5 is 4×3 states rather than 4×256 states. In our final version, we don't do this translation, and just have large state-machines instead.

Line 15 loops getting the next byte of input, one byte at a time. Calling `getchar()` here for every character is potentially expensive, but we aren't benchmarking this program, just showing the algorithm. In our final version, we read input a buffer at a time instead of a byte at a time.

Line 16 does the state transition, in other words, it parses the input. We translate the byte into one of the three column values, `word(0)`, `space(1)`, or `newline(2)`. We look up that in the current row, then set the next row according to the transition. Thus how in the `was-space(0)` state, if we receive a `non-space(0)` character, we transition to `new-word(2)` state.

Line 17 processes what we parsed. In our case, the processing is trivialized to just counting the number of times we visit each state.


```

#include <stdio.h>
2 int main(void)
{
4     static const unsigned char table[4][3] = {
        {2,0,1}, {2,0,1}, {3,0,1}, {3,0,1}
6     };
    static const unsigned char column[256] = {
8         0,0,0,0,0,0,0,0,0,1,2,1,1,1,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,
10    };

12    int state = 0;
    int c;

14    while ((c = getchar()) != EOF) {
16        state = table[state][column[c]];
        counts[state]++;
18    }
    printf("%lu %lu %lu\n", counts[1], counts[2],
20        counts[0] + counts[1] + counts[2] + counts[3]);
    return 0;
22 }

```


Figure 17: wc2o.c, an obfuscated word counter for ASCII.

	word(0)	space(1)	newline(2)
was-space(0)	new-word(2)	was-space(0)	new-line(1)
new-line(1)	new-word(2)	was-space(0)	new-line(1)
new-word(2)	was-word(3)	was-space(0)	new-line(1)
was-word(3)	was-word(3)	was-space(0)	new-line(1)

Table 1: Simple word-count state machine.

NEW FROM XITEX

\$95 MORSE TRANSCIVER



SEND:

- 1 to 150 WPM (set from terminal)
- 32 character FIFO buffer with editing
- Auto Space on word boundaries
- Grid/Cathode key output
- LED Readout for WPM and Buffer space remaining

SERIAL INTERFACE:

- ASCII (110, 300, 600, 1200) or Baudot (45, 50, 57, 74) compatible
- Simplex Hi V Loop or T'L electrical interface
- Interfaces directly with the XITEX® SCT-100 Video Terminal Board; Teletypes® Models 15, 28, 33, etc.; or the equivalent

MRS-100 CONFIGURATIONS:

- \$95 Partial Kit (includes Microcomputer components and circuit boards; less box and analog components)
- \$225 Complete Kit (includes box, power supply, and all other components)
- \$295 Assembled and tested unit (as shown)

Overseas Orders and dealer inquiries welcome

COPY:

- 1 to 150 WPM with Auto-Sync.
- Continuously computes and displays Copy WPM
- 80 HZ Bandpass filter
- Re-keyed Sidetone Osc. with on-board speaker
- Fully compensating to copy any 'fist style'

See your local dealer or contact XITEX® direct.

MC/Visa accepted

XITEX CORP
13628 Neutron • P. O. Box 402110
Dallas, Texas 75240 • (214) 386-3859

Line 19 prints the results. The number of lines is the number of times we visited the `new-line(1)` state. The number of words is the number of times we visited the `new-word(2)` state. The number of characters is the number of times we visited all the states combined.

Consider reading input whose only letter is 'x'. We start with `state==was-space(0)`. The 'x' is then translated into a column value, `word(0)==column['x']`. The new state becomes `new-word(2) == table[was-space(0)][word(0)]`.

The resulting program produces the same output as the built-in program.

```
$ wc < wc2o.c
2 26 74 649
4 $ ./wc2o < wc2o.c
26 74 649
```

What about Unicode and UTF8?

This program does hard-coded ASCII, a single-byte character set. We need something that can handle multi-byte character-sets, like Unicode and UTF-8.

Unicode is a character-set; UTF-8 is an encoding. Unicode characters, or wide characters or code points, are integers between 0 and 0x100FFFF. Each code point can be represented by from one to four bytes, as shown in Table 2.

Consider the character U+1680, Ogham Space Mark. According to the table, this is encoded as the three-byte sequence 0xE1 0x9A 0x80.

Ogham is an alphabet from the sixth century for writing Old Irish that survives today as roughly 400 inscriptions on monuments and gravestones. A compliant version of `wc` must count spaces on those monuments. The space mark counts as a space character according to `iswspace(0x1680)`. Thus, we should be able to count words in such inscriptions.⁴¹

```
$ echo "1 4 30" | wc -lwm
```

There are about thirty such Unicode space characters for which `iswspace()` will return true. The libraries for Windows, Linux, and macOS have slight disagreements about this, so what some will recognize as a space will not be recognized as a space on others. However, they all agree that U+1680 is a space.

⁴¹If the editors have done their job right, you should be able to copy/paste this from the online PDF document and reproduce these results. It works on macOS, Linux, and on Windows using the PowerShell `Measure-Object` commandlet, when the locale is set to Unicode. —Rob

There are many invalid UTF-8 sequences. Among those are unnecessarily long, redundant sequences. The table suggests that 0x0A may also be represented as 0xC0 0x8A and 0xE0 0x80 0x8A. Since this can be encoded as simply 0x0A, the longer sequences are declared to be officially invalid and must be rejected.

Thus, a parser for Unicode must not only consider the basic math as shown in the table, but also recognize spaces and reject invalid sequences.

How do state-machine parsers work?

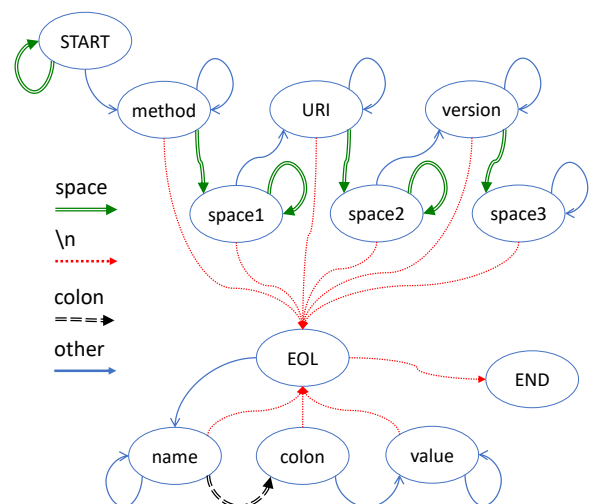
You are familiar with state-machines in other parts of computer-science, such as the famous TCP/IP state-machines. In those state-machines, some sort of event happens that causes a transition from one state to another. For parsers, the event is the next byte of input. Each byte of input is read sequentially, and depending up that byte's value, a transition happens from one state to another.

There are two ways to represent these transitions: either through a big lookup table as in `wc2o.c` on page 73, or with a `switch/case` block.

Consider HTTP. A request header looks like the following:

```
1 GET /index.html HTTP/1.0
Host: www.google.com
3 User-Agent: Mozilla (actually Chrome)
```

A state-machine that parses it might assign a state to each field, like this.



Scalar Unicode Value	First Byte	Second	Third	Fourth
00000000 00000000 0xxxxxxx	0xxxxxxx			
00000000 00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
00000000 zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Table 2: UTF-8 Bit Distribution, Unicode 6.0

As we receive the bytes of an HTTP request, we enter the `method` state the first time we receive a non-space character. We remain in the `method` state until we receive a space character, at which point we transition to the `space1` state.

In C, we might process each byte of input with a function like the following `switch/case` logic:

```

1 int http_parse(int state, unsigned char c,
2               ...) {
3     switch (state) {
4         ...
5     case METHOD: /*GET, POST, HEAD, ...*/
6         if (c == '\n') {
7             ...
8             return EOL;
9         } else if (isspace(c)) {
10            ...
11            return SPACE1;
12        } else {
13            ...
14            return state; /* no change in state */
15        }
16        ...
17    }
18 }


```


Most major web servers that aren't Apache use this method. Nginx calls this state `sw_method`, which you can see in the open-source online.⁴²

You can test on a live network whether a web server is parsing requests using a state-machine. Send a request to the server consisting of GET, followed by five billion spaces and only then the rest of the request. If the server acts like Apache buffering a complete header, then it'll run out of buffer space. If instead the server acts like Nginx and parses input with a state machine, it'll happily keep reading spaces as long as it's in that state. If the connection terminates prematurely, it'll be because of a timeout instead of running out of buffers. (It takes a while to send five gigabytes.)

⁴²See near line 159 of `ngx_http_parse.c`.

This example uses a `switch/case` block of code to handle the transitions. In our state-machines for counting words, we use a lookup table instead. A third choice is to use a mixture. The program `masscan`, for example, does a lot of parsing of such protocols like FTP, SMTP, X.509, and so. It uses a mixture of `switch` statements and lookup tables.







(ALMOST) FREE CLUES:

If you've ever been stuck in an adventure game, you need **The Book of Adventure Games** by Kim Schuette. This fantastic book contains complete legible (typeset) maps, magnificent illustrations, and all the hints you need to complete 77 of the all-time most popular adventure games including **Zork I, II, III, Deadline, Starcross, Witness, Planetfall, Enchanter, Sorcerer, Infidel, Suspended (with map), Wizardry, Knight of Diamonds, Legacy of Lylgamyn, All Scott Adams, All Sierra On-Line** including **Time Zone, Ultima I, II, III** and many more! Best of all, the book doesn't spoil your fun! At about 25¢ an adventure, it's the biggest bargain around. So stop getting ripped off by \$10 cluebooks and call:

1-(800)-821-5226 Ext. 500
24 hrs. a day, 7 days a week
or write:
Witt's End
42 Morehouse Rd., Dept. 6
Easton, CT 06612

Free UPS shipping. Add \$3.00 for C.O.D. APO's FPO's o.k. Add \$5.00 for foreign shipping. No charge for credit cards. We accept Visa: Mastercard, Personal Check (allow 2 weeks to clear), Certified Check or money order.


All Trademarks are acknowledged.

ELIZABETH GRANT
HIGH CLASS MILLINERY

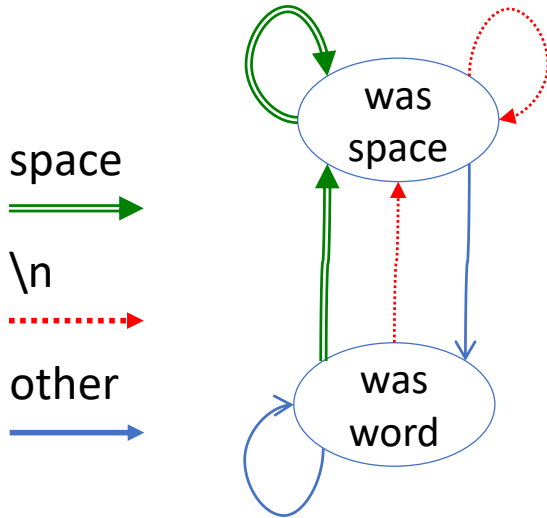
12 WEST STREET (Over Bigelow-Kennard's)

Smart Tailored and Dress Hats. Made of fine materials and of the best workmanship. Exclusive styles. No two hats alike. Courteous attention whether you buy or not.
PRICES, SIX DOLLARS AND UP.

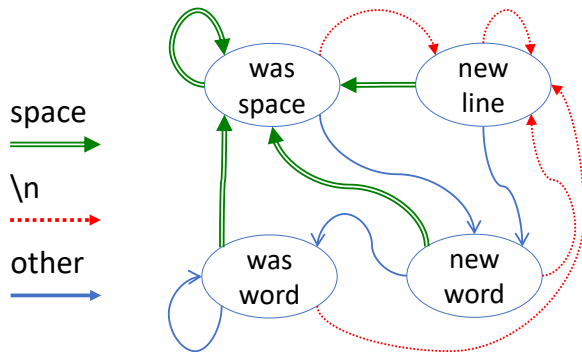


How can we construct a state-machine for word-counting?

Most implementations of `wc` effectively use a machine with two states which can be represented with the following diagram. Note that they aren't designed explicitly as a state-machine, but that's effectively how the code works.



In the `wc2o.c` program, we changed this to a machine with four states. This is the table with three types of transitions and four states:



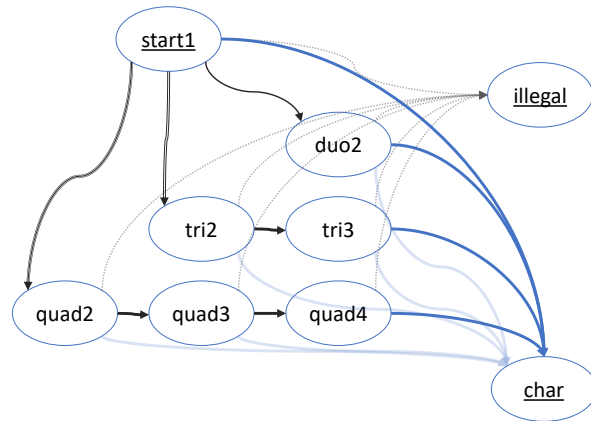
We did this in order to be overly clever in how we were going to process the data.

Remember, there's two things going on here. One step is parsing the input. The next step is processing the results. Thus, we first need to parse out things like words, characters, and newlines. Then we need to process this information, which for word-counting, is done by counting each time we enter a state. We've cleverly collapsed the processing into a simple operation.

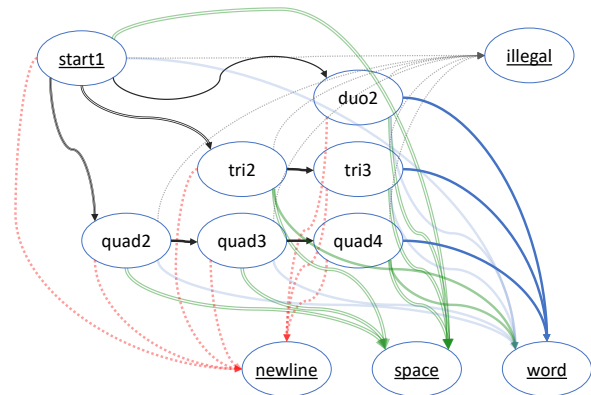
SAM COUPE AND SPECTRUM MAGAZINE!
 PROGRAMS, UTILITIES, INFO, IDEAS! NEWS, REVIEWS AND HOMEGROWN SOFTWARE MONTHLY SINCE 1987!
"OUTLET"
 GRAPHICS, AND HELP PAGES, SERIOUS SOFTWARE
 SPECIAL OFFER! Latest issue £2.50 to newcomers on:-
 +3, DISCIPLE/+D, MICRODRIVE, OPUS, TAPE, SAM DISC
 CHEZRON SOFTWARE, 605 LOUGHBOROUGH RD., BIRSTALL, LEICESTER LE4 4NJ

The lesson here isn't that parsers can completely trivialize processing as we've done here, but instead that we often add artificial states to benefit later processing.

Now let's talk about UTF-8. Using the original table as a guide, we might construct a state-machine for parsing 1-byte sequences, 2-byte sequences called a "duo," 3-byte sequences "tri," and 4-byte sequences "quad."



However, our needs are simpler. We don't need to parse out the code point and test with `iswspace()` but can instead include that functionality within the state-machine itself, where the output is one of four values: word, space, newline, or illegal.



There are, in fact, more states than just this. Instead of a simple path for 3-byte characters, we must add additional states that recognize 3-byte characters that result in spaces. This creates a table of roughly thirty states that's too complex to draw here.

Instead, here are snippets of the code that take an existing table and adds states for characters like U+1680 Ogham Space Mark. It clones existing states that follow the same path, but at the end marks the character as a `space` instead of a `word`:

```

/* clone existing states */
2 memcpy(table[TRI2_E1], table[TRI2],
        sizeof(table[0]));
4 memcpy(table[TRI3_E1_9a], table[TRI3],
        sizeof(table[0]));
6 /* link in new states */
table[0][0xE1] = TRI2_E1;
8 table[TRI2_E1][0x9a] = TRI3_E1_9a;
table[TRI3_E1_9a][0x80] = SPACE;

```

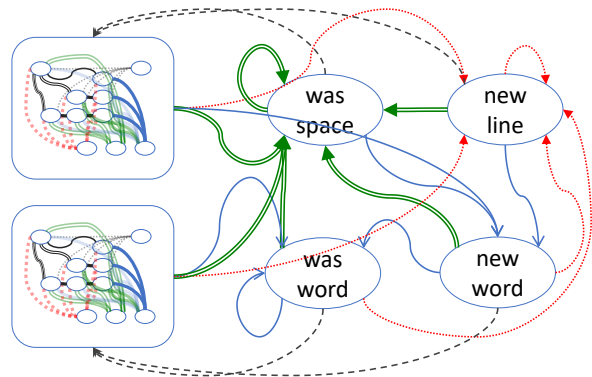
What this code is doing is exactly what generic regex code would do. All we are doing here is creating manually what regex libraries would do based upon expressions. What we are doing here is manual optimization for concepts that exist abstractly.

Now let's combine our UTF-8 state-machine parser with our word-count state-machine parser. There's two ways of doing this. The obvious way is to feed the output of one as input to the other. The other way is to combine the two into a single state-machine.

This is a multiplicative process. That means replicating one state-machine for every state in the other state-machine. Again, let's talk regex theory. There are two ways of representing such a thing. One way increases computation, what we call an NFA or non-deterministic finite automata, which is what would happen if we fed the output of one as the input to the next. The other way keeps computation the same but increases the size of the table. This is a DFA or deterministic finite automata. As you build complex regexes, you cause either computation to explode or memory to explode. In this case, we've chosen DFA, so memory explodes.

Thus, where one state-machine needs 35 states and the other just four, that means the combination may needs as many as $4 \times 35 = 140$ states. However, we are going to do a small trick. The `was-space` and `new-line` states are clones of each other, as are `was-word` and `new-word`. Thus, we

only need to double rather than quadruple the UTF-8 state-machine. This produces something that may be represented like:



The Final Code

The final code is in `wc2.c`. It's a few hundred lines so is not included in this article but is instead available on GitHub.⁴³

The complicated part that takes hundreds of lines is where it builds that state-machine table. This results in a table roughly with 70 states (rows), and 256 columns, where each column represents the transition that will happen when a byte of input is received.

Once we've built the table, we simply process chunks of input analogous to the following. The actual code looks slightly different, with the inner loop separated into a `parse_chunk()` function.

```

1 unsigned counts[MAX_STATE];
  //Get the next chunk of input.
3 length = fread(buf, 1, sizeof(buf), fp);
  //For all bytes in that chunk,
5 for (i=0; i<length; i++) {
  //Get the next byte.
7     c = buf[i];
  //Do the state transition.
9     state = table[state][c];
  //Do the counting.
11    counts[state]++;
  }
13 //Report the results.
word_count = counts[NEW_WORD];
15 line_count = counts[LINE_COUNT];
char_count = counts[0] + counts[1]
17             + counts[2] + counts[3];

```

⁴³[git clone https://github.com/robertdavidgraham/wc2](https://github.com/robertdavidgraham/wc2) || `unzip pocorgtfo21.pdf wc2.zip`

sized blocks, like 64k; you can't choose the size of blocks depending upon the parsed contents. When data is received, it is dispatched to the appropriate copy of the state for parsing each file. We just need an 8-bit integer for every file to hold all the per file parser state.

This would be a silly thing to do with files but is an important thing for networking services. Apache is broken and can't scale beyond 10,000 concurrent TCP connections because it struggles with 10,000 threads in the system. All its major competitors use a single thread (or single thread per CPU core) and handle things asynchronously.

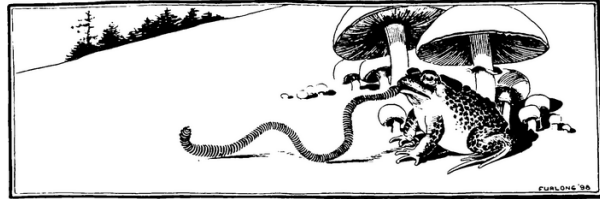
Again, the parser can't influence data reception. The network stack simply receives packets from the other side, whatever size of packets those might be. The parser must handle the case where it receives exactly as much data as it was expecting, or too much data, or not enough data.

Conclusion

There have been many posts over the last year of people implementing `wc` in their favorite language, such as Haskell or Python. In this paper, instead of a different programming language, we've chosen a fundamentally different algorithm, that of an asynchronous state-machine parser. We've implemented the same algorithm in both C and JavaScript, to show that the speed is property of the algorithm instead of the language.

Instead of a simplified problem of just handling ASCII, we've demonstrated the algorithm using the difficult problem of UTF-8 encodings. Given something bizarre like Ogham text, we still produce the same answer as compliant `wc` programs. While only the UTF-8 encoding is implemented, the concept extends to any character-set, including the CJK (Chinese, Japanese, Korean) multi-byte character-sets.

Such state-machine parsers are costly in terms of code maintainability: most programmers are unfamiliar with them. However, they have clear advantages for writing scalable, secure code for modern Internet applications.



PRO-LOG IS PROVIDING LOCAL STOCKING FOR THEIR STD MICROPROCESSOR BOARD PRODUCTS

Effective
September 21, 1981

Pro-Log will begin a local distribution plan for their microprocessor board product line. Off the shelf delivery will be available through their stocking representative network in selected locations across the United States. The initial stocking locations will include:

Northern California	Manco 3350 Scott Blvd., Bldg. 55 Santa Clara, CA 95051
Southern California	Advanced Digital Group 15904 Strathern Sr., Suite 21 Van Nuys, CA 91406
New York Area	Tecnimat 500 Grand Avenue Englewood, NJ 07631
New England States	Martindale Associates 212 Main Street North Reading, MA 01864

Watch for additional stocking locations in the very near future!

108560A 11K 9/81

21:15 Never say ‘no’ to adventures.

*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.

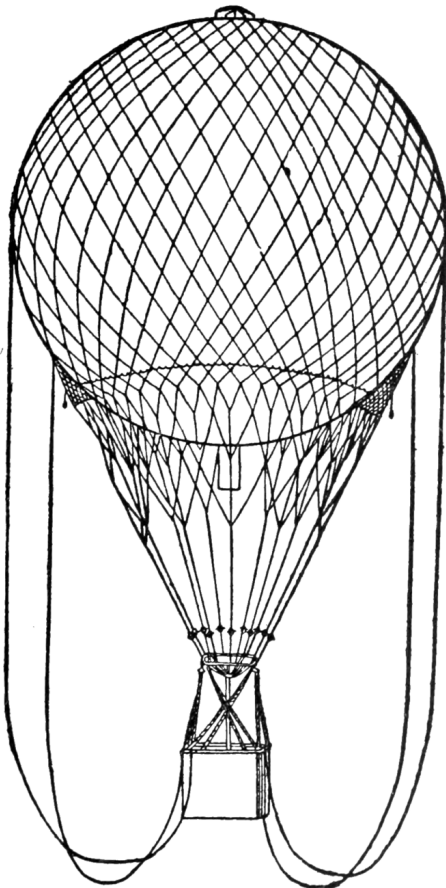


So today, in that spirit of exploration and wonder, I pass around the collection plate and ask you, neither for paper money nor pocket change, but for nifty projects and the clever tricks that make them possible.

Maybe share a technical story from the good old days, such as when the Super Nintendo and Apple IIGS both used a 16-bit 65C816 CPU, with two instruction sets for backward compatibility with their 6502 predecessors. Maybe share a clever trick from the modern day, such as how to scale a disassembler to terabytes of input, or how to explore all the BARs of a PCIe card to quickly rig up a new driver.

Give me source code for the software, and give me schematics for the hardware, but most of all teach me how to build these things for myself. Teach me to know the difference between those things that are really hard, and those things that only look intimidating before a bit of practice and the right advice collapse the problem into something a clever child might solve.

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. My gang and I will clean it up, typeset it in T_EX, index it and print it for the world. We’ll happily translate from French, Spanish, Portuguese, German, Russian, Hungarian, Hebrew, Serbo-Croatian, and Southern Appalachian.



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