Dumping Firmware for fun and learning

Thanks to

- Thanks to H2LAB!
- Check out [h2lab.org](h2lab.org)

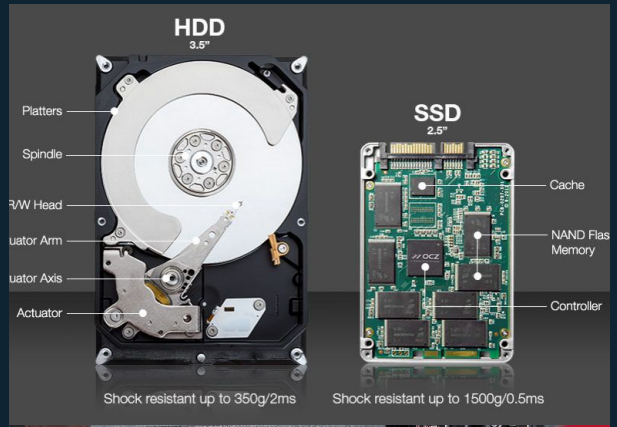
Talk aimed towards

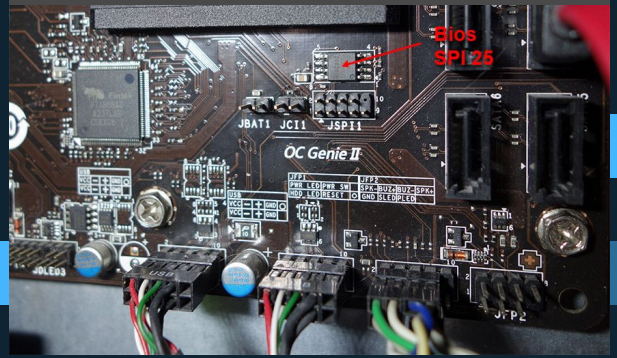
- Curious people about hardware security.
- Technical people interested in getting started.

Where are SPI flashes?

Pretty much everywhere, you need some (magic) non-trivial controls.

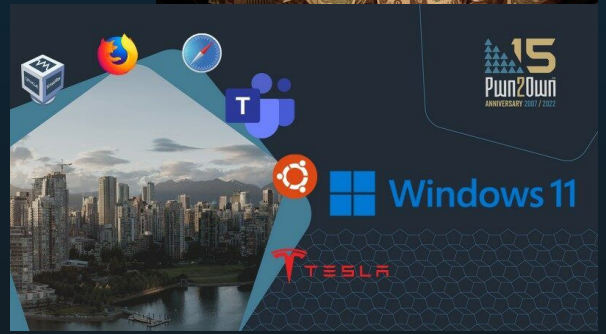
- Network cards
- HDD/SSD
- BIOS (UEFI)
- IoTs
- Network routers
- Internet boxes



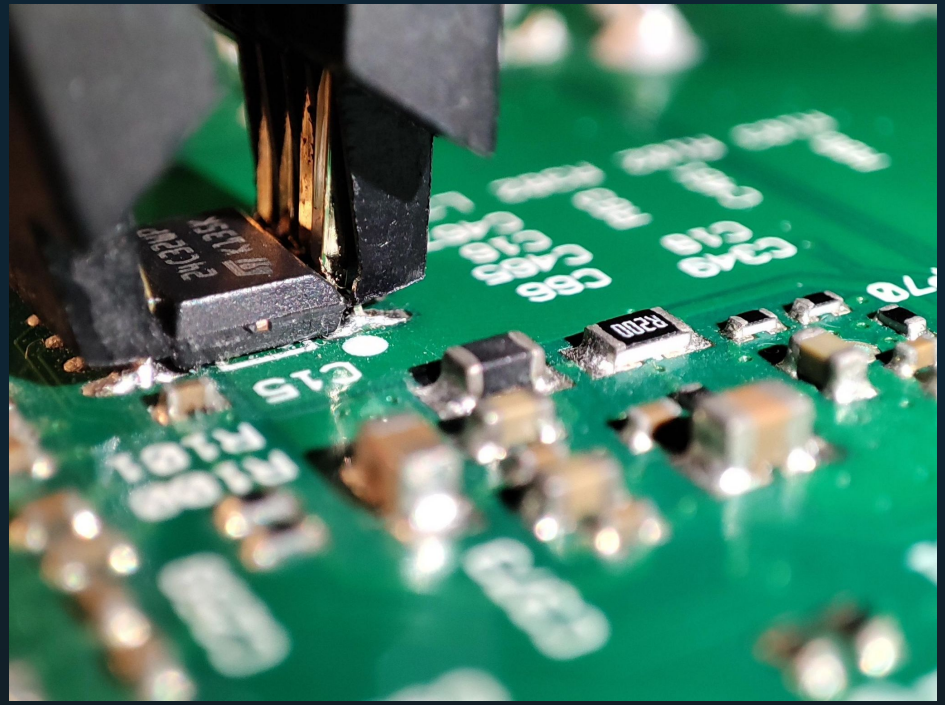


Why the hell dump firmware ?

- For fun and profit
- Patching/customizing
- For pwning purposes (train for Pwn2Own)
- Enumerating creds
- Finding 0day vulnerabilities



Main problem



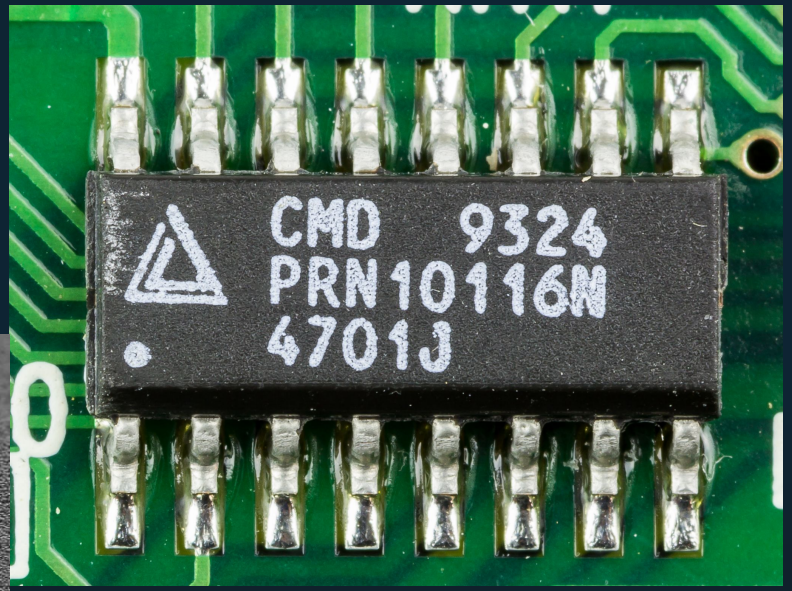
How to get binary out of SPI chips

The Plan

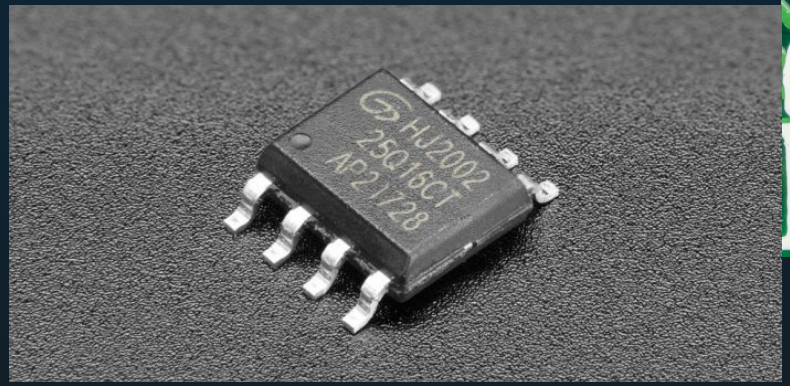
- ✅ Identify SPI chip and orientation
- ✅ Finding pinout/datasheet
- ✅ Connect on SPI and dump
- ✅ Identify architecture

Identifying SPI chips

- Usually 8 or 16 pins
- Usually SOIC IC package
- Usual voltage supported are either 1.8V/3.3V/5V

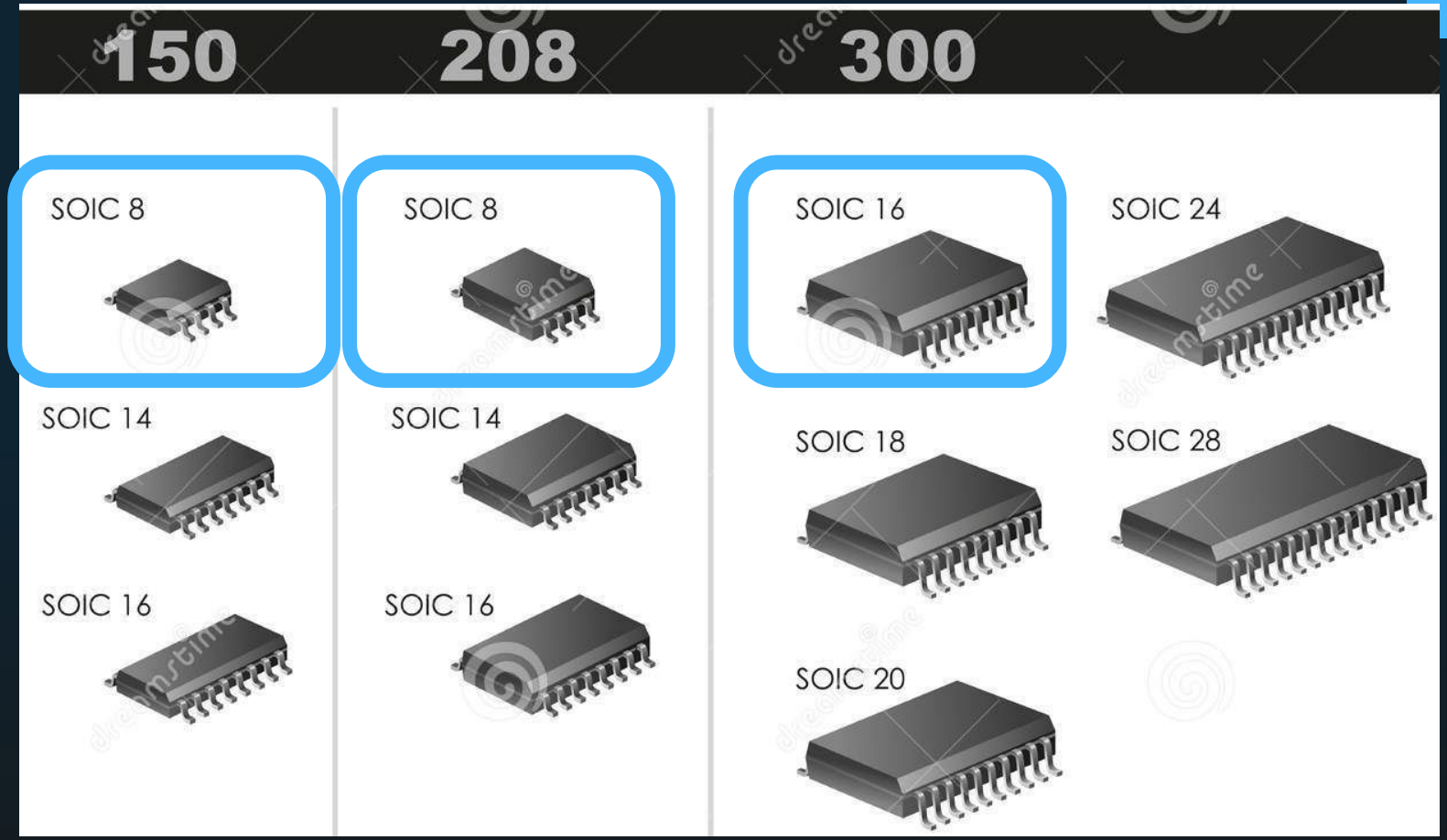


16 pins

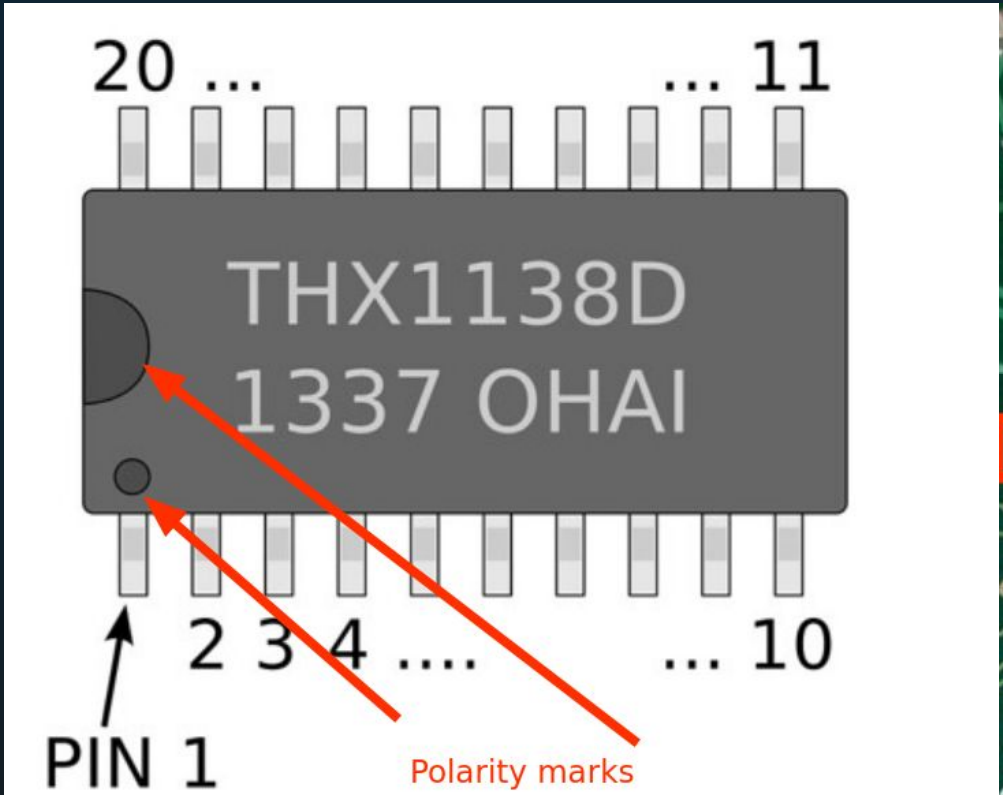


8 pins

Identifying SPI chips



Pin 1


20 ...
... 11
THX1138D
1337 OHAI
2 3 4 ....
... 10
PIN 1
Polarity marks

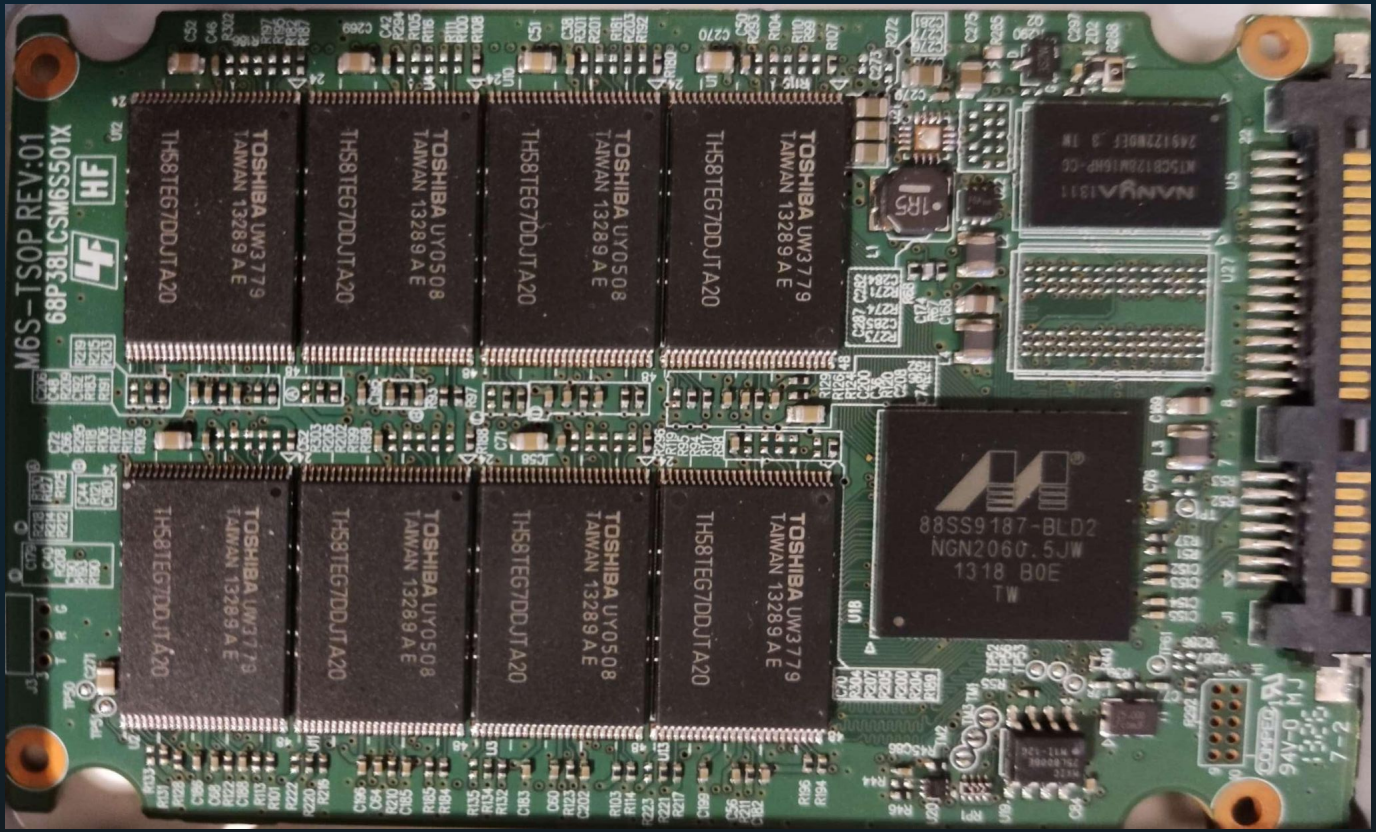

5CA7JJM
X202C
G4


U5
BROADCOM
BCM5231A4KPTG
TA0525 P14
746917A

The Plan

- ✅ Identify SPI chip and orientation
- ✅ Finding pinout/datasheet
- ✅ Connect on SPI and dump
- ✅ Identify architecture

Practice makes...



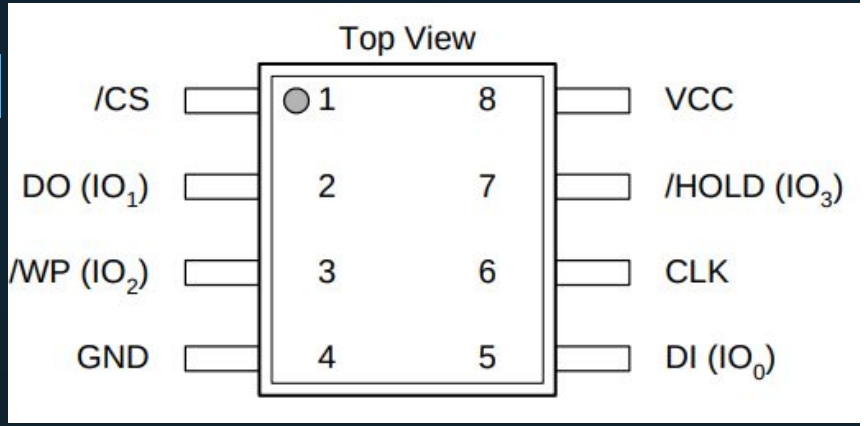
The target

- Winbond 25Q80VIG
- Datasheet online
- Flash 1024 KB
- 2.5V-3.6V
- Stores HDD firmware
- Drives the SATA SSD

Finding pinouts

- Finding datasheets, a bit of an OSINT work.
- Generally looking up the names in search engines
- IC manufacturers websites
- Some websites:
  - https://datasheetspdf.com/
  - for many HDDs: http://www.users.on.net/~fzabkar/Datasheets/DATAURLS.HTM
  - http://www.datasheetcatalog.com/, web archives for old chips

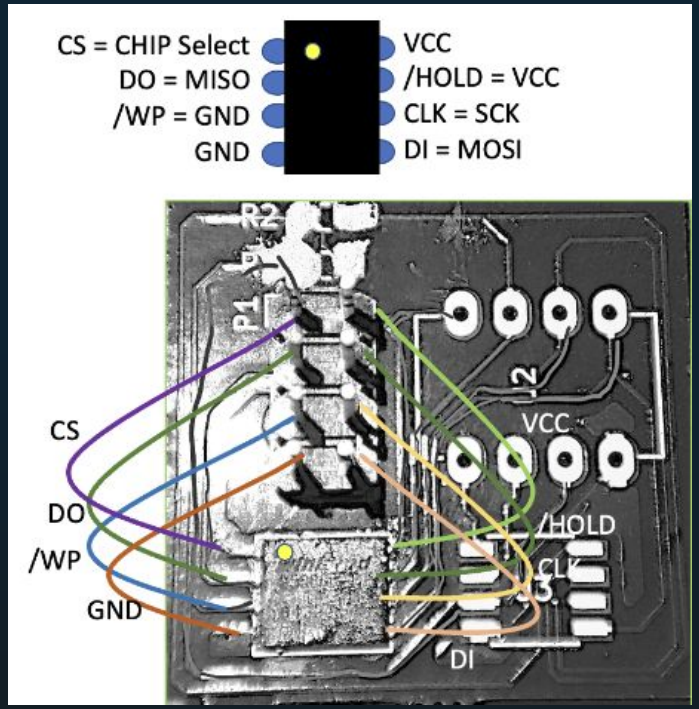
Technical documentation



2. FEATURES

- **Family of SpiFlash Memories**
  - W25Q80BV: 8M-bit/1M-byte (1,048,576)
  - 256-byte per programmable page
  - Standard SPI: CLK, /CS, DI, DO, /WP, /Hold
  - Dual SPI: CLK, /CS, IO_0, IO_1, /WP, /Hold
  - Quad SPI: CLK, /CS, IO_0, IO_1, IO_2, IO_3
- **Highest Performance Serial Flash**
  - 104MHz Dual/Quad SPI clocks
  - 208/416MHz equivalent Dual/Quad SPI
  - 50MB/S continuous data transfer rate
  - Up to 8X that of ordinary Serial Flash
  - More than 100,000 erase/program cycles[1]
  - More than 20-year data retention
- **Efficient "Continuous Read Mode"**
  - Low Instruction overhead
  - Continuous Read with 8/16/32/64-Byte Wrap
  - As few as 8 clocks to address memory
  - Allows true XIP (execute in place) operation
  - Outperforms X16 Parallel Flash
- **Low Power, Wide Temperature Range**
  - Single 2.5 to 3.6V supply

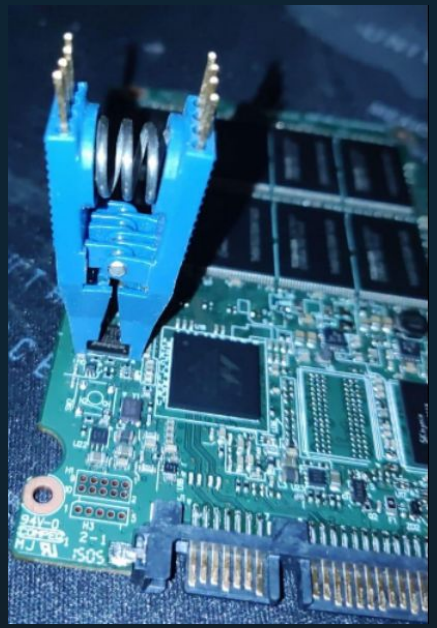
Let's talk about pinout...



The Plan

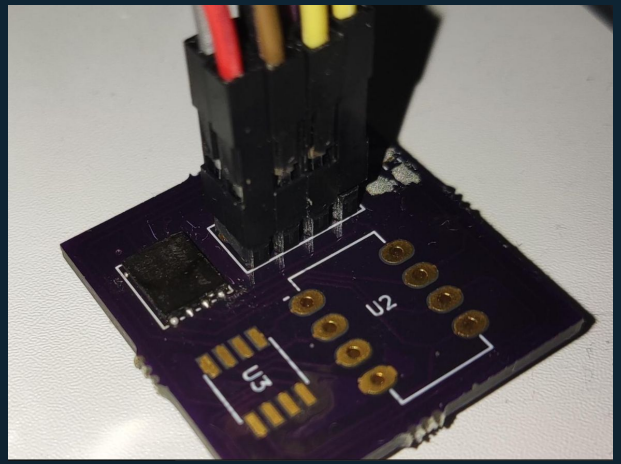
- ✅ Identify SPI chip and orientation
- ✅ Finding pinout/datasheet
- ☑️ Connect on SPI and dump
- ☑️ Identify architecture

Approach 1



- No soldering required
- Simple & easy
- Plug & Play

Approach 2

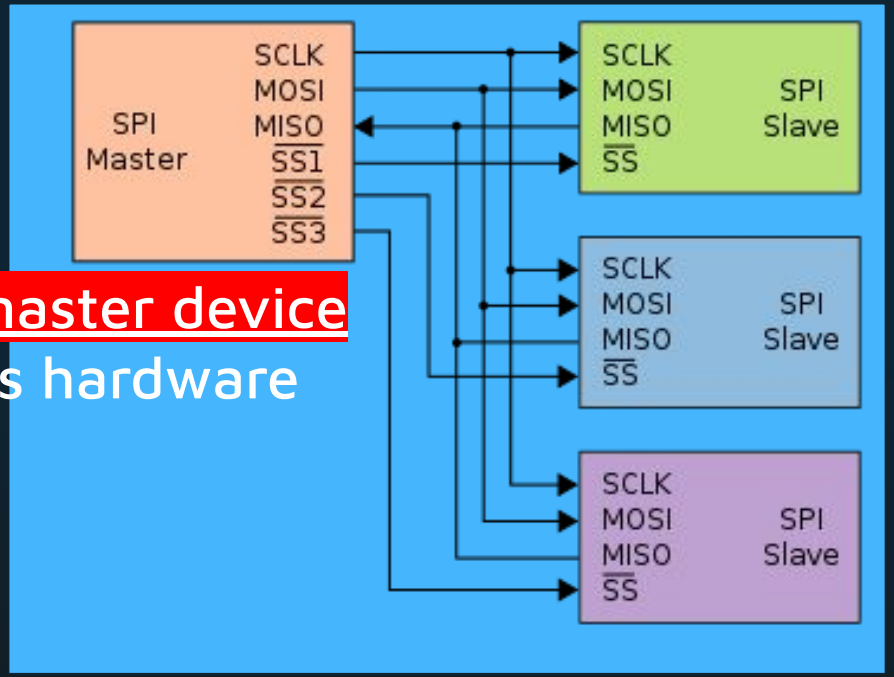


- Soldering required
- Simple custom PCB
- More work but higher chance of success

Any clue why approach 1 fails sometimes ?

Back to the good old theory SPI

1 Master /N slaves

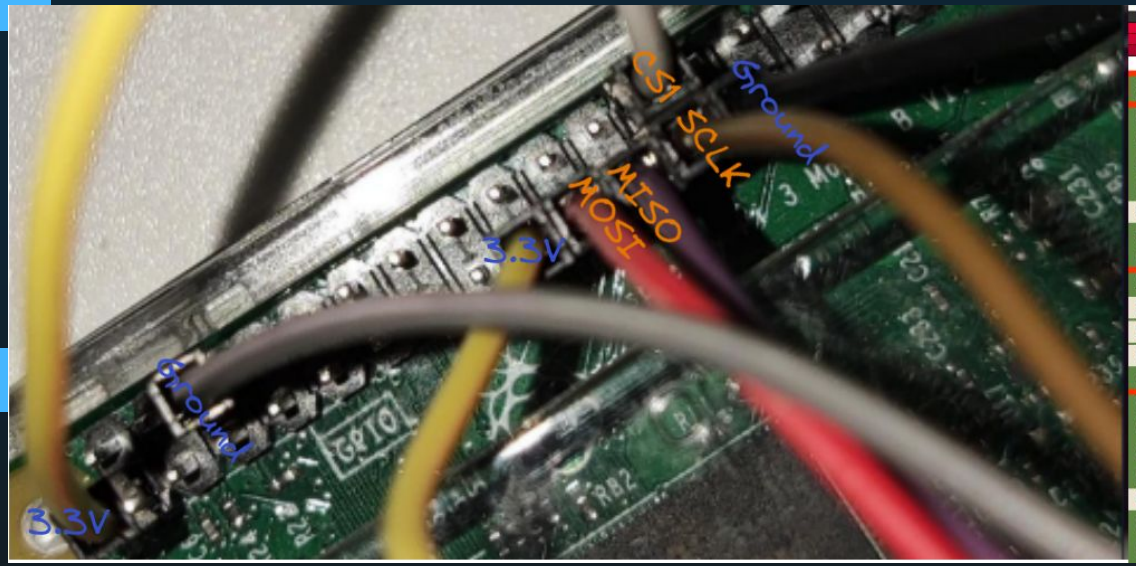


Supports only one master device (depends on device's hardware implementation)

Let's come back to our case:

The SPI chip is off so the SoC won't be the Master anymore

Plugging on the RPI



Flashrom

- Software for reading SPI chips
- Works well on Raspberry Pi, Arduino and other hardware.
- Support writing to many SPI microcontrollers.

- Can also dump from Linux BIOS SPI chip.

```
Probing for Spansion S25FL256S Small Sectors, 16384 kB: Read id bytes:  0xef 0x60 0x18 0x00 0x00 0x00.
Probing for Spansion S25FL256S.....0, 32768 kB: Probing for Spansion S25FL512S, 65536 kB: Probing for S
ef 0x60 0x18 0x00 0x00 0x00.
Probing for Spansion S25FS128S Small Sectors, 16384 kB: Read id bytes:  0xef 0x60 0x18 0x00 0x00 0x00.
Probing for Winbond W25P16, 2048 kB: compare_id: id1 0xef, id2 0x6018
Probing for Winbond W25P32, 4096 kB: compare_id: id1 0xef, id2 0x6018
Probing for Winbond W25P80, 1024 kB: compare_id: id1 0xef, id2 0x6018
Probing for Winbond W25Q128.V, 16384 kB: compare_id: id1 0xef, id2 0x6018
Probing for Winbond W25Q128.V..M, 16384 kB: compare_id: id1 0xef, id2 0x6018
Probing for Winbond W25Q128.W, 16384 kB: compare_id: id1 0xef, id2 0x6018
Added layout entry 00000000 - 00ffffff named complete flash
Found Winbond flash chip "W25Q128.W" (16384 kB, SPI) mapped at physical address 0x00000000ff000000.
Chip status register is 0x00.
Probing for Winbond W25Q128.JW.DTR, 16384 kB: compare_id: id1 0xef, id2 0x6018
Probing for Winbond W25Q16.V, 2048 kB: compare_id: id1 0xef, id2 0x6018
Probing for Winbond W25Q16.W, 2048 kB: compare_id: id1 0xef, id2 0x6018
```

https://github.com/flashrom/flashrom.git

Reading with Flashrom

```
sudo flashrom -p linux_spi:dev/spidev0.0 spispeed=8000 -r ssd-dump.bin
```

Programming device

CHIP Select number 0 or 1 on RPI

SPI speed

reading SPI into file

Advices :

- Read multiples times the ROM in different files and compare shasums.
- spispeed choices depends on programmer and target.

The Plan

- ✅ Identify SPI chip and orientation
- ✅ Finding pinout/datasheet
- ✅ Connect on SPI and dump
- ☑️ Identify architecture

To sum up

01

Dumping using a SOIC-8 clip

Failed lamentably

02

Desolder SPI chip

Easy of course

03

Solder it on a custom PCB

Using your expert micro soldering skills

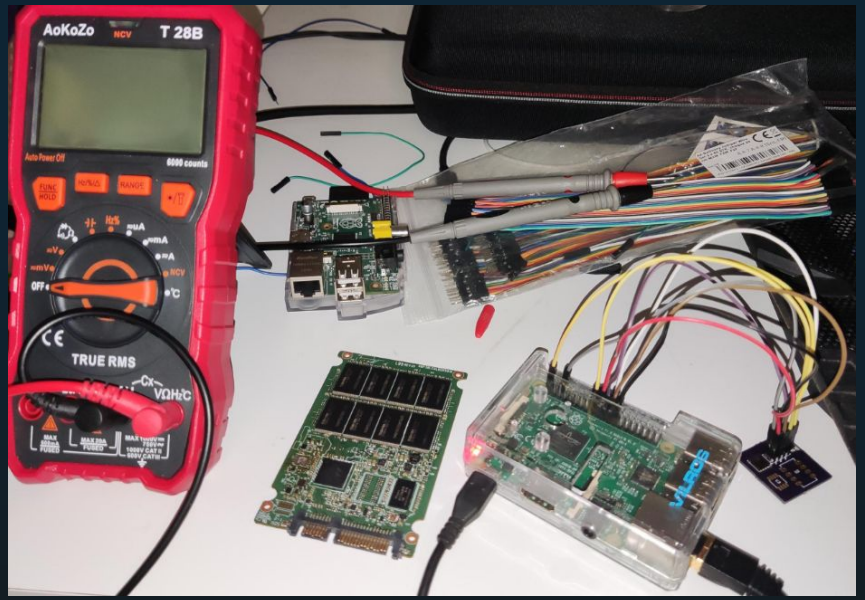
04

Dump it using a Raspberry Pi 3B+

Worked flawlessly

Final setup

- Multimeter (20€)
- Soldering Iron (30€-80€)
- Raspberry Pi 3B+ (35€)
- SOIC-8 clip (5€)
- Custom PCB (1.5€)
- Jumper wire (5€)
- Hot Air Station (25€-1500€)



Architecture of the binary

- file
- ISAdetect (uses ML to predict binary architecture)
- binwalk -A (find assembly instructions)

- binbloom (attempt to find base address of binary using heuristics)

```
{
  "prediction_probability": 0.48,
  "prediction": {
    "wordsize": 32,
    "architecture": "arm",
    "endianness": "little"
  }
}
```

```
[redacted] binwalk -A flash-ssd-1st-attempt.rom

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
6124          0x17EC          ARM instructions, function prologue
6244          0x1864          ARM instructions, function prologue
6260          0x1874          ARM instructions, function prologue
[nix-shell:[redacted]$ binbloom -a 32 ./flash-ssd-1st-attempt.rom
[i] 32-bit architecture selected.
[i] File read (1048576 bytes)
[i] Endianness is LE
[i] 249 strings indexed
[i] Found 2614 base addresses to test
[i] Base address found: 0x0000d000.
 More base addresses to consider (just in case):
```

- Interrupt vector table (usually at the beginning)
- Confirms ARM architecture.
- ARMv7 Little Endian.

flash-ssd-1st-attempt-annoted-func-1.rom

```
                    //
                    // ram
                    // ram:00000000-ram:000fffff
                    //

                    **************************************************************
                    *                          FUNCTION                          *
                    **************************************************************
                    undefined __stdcall Reset(uint param_1, uint param_2)
  undefined           r0:1            <RETURN>
  uint                r0:4            param_1
  uint                r1:4            param_2
                    Reset+1                                        XREF[1,1]:   Entr
                  ↗ Reset                                                       lots
000000 5a 00 00 00     andeq       param_1,param_1,r10, asr param_1

                    **************************************************************
                    *                          FUNCTION                          *
                    **************************************************************
                    undefined __stdcall UndefinedInstruction(uint param_1, u...
  undefined           r0:1            <RETURN>
  uint                r0:4            param_1
  uint                r1:4            param_2
                  ↗ UndefinedInstruction                           XREF[4]:     Entr
                                                                                0002
000004 11 1c 00 00     andeq       param_2,param_1,param_2, lsl r12

                    **************************************************************
                    *                          FUNCTION                          *
                    **************************************************************
                    undefined __stdcall SupervisorCall(uint param_1, uint pa...
  undefined           r0:1            <RETURN>
  uint                r0:4            param_1
  uint                r1:4            param_2
                  ↗ SupervisorCall                                 XREF[1]:     Entr
000008 10 1c 00 00     andeq       param_2,param_1,param_1, lsl r12

                    **************************************************************
                    *                          FUNCTION                          *
                    **************************************************************
                    undefined __stdcall PrefetchAbort(undefined4 param_1)
  undefined           r0:1            <RETURN>
  undefined4          r0:4            param_1
                  ↗ PrefetchAbort                                  XREF[1]:     Entr
00000c 20 00 00 00     andeq       param_1,param_1,param_1, lsr #32
```

The Plan

- ✅ Identify SPI chip and orientation
- ✅ Finding pinout/datasheet
- ✅ Connect on SPI and dump
- ✅ Identify architecture

What next ?

- Reverse engineering the SPI firmware.
- Activating the UART debug interface.
  - Finding the debug sequence.
  - Glitching the SoC during boot in order to activate the debug port ?
  - Mapping SoC's pinout, find another debug interface
- Finding JTAG pinout/interface.
  - If active, dump the SoC's BootROM
  - If not, glitching SoC until it becomes active ?
- Asking the SoC's datasheet to shady chinese suppliers.

Some proprietary documentation

Entering command line debug mode:

During power up, the user can transmit a special 64-bit debug pattern (0xDD 0x11 0x22 0x33 0x44 0x55 0x66 0x77) in a loop, to indicate a request for debug. When the bootROM begins execution, it attempts to read data from the UART0 interface (as explained above). If it successfully reads the debug pattern, it enters the command line debug mode. It configures the appropriate MPP pin to operate as a UA0_TXD signal. (If a pattern is detected on MPP[4], then it configures MPP[5] as UA0_TXD, otherwise it configures MPP[10] as UA0_TXD.) Then it prints the bootROM version, and waits on the debug prompt for user input.

BootROM Firmware Boot Sequence

The bootROM firmware boot sequence can be divided into the following phases, which are described in the following sub-sections:

1. Initialization
2. Boot mode selection
3. Sensing the UART0 interface to detect a user request for entering the bootROM command line debug mode or for booting from UART0.
4. Load, check, and update the device's main header information.
5. If an extended header exists:
   a) Load and check the device's extended header.
   b) Execute all register configurations indicated in the extended header.
6. Load and check the device image (unless it is to be executed from the SPI).
7. Execute the device image code.

In addition, bootROM firmware contains the following functionality:

- Error reporting and handling
- Exception handling

THANKS

Any questions?

AkechiShiro

Lahfa Samy

samy@lahfa.xyz

Resources

https://github.com/quarkslab/binbloom/

https://github.com/kairis/isadetect

https://pinout.xyz/pinout/spi

https://github.com/flashrom/flashrom.git

https://www.nsideattacklogic.de/en/dumping-spi-flash-memory-of-embedded-devices-2/

https://www.riverloopsecurity.com/blog/2020/02/hw-101-spi/

https://www.evilmadscientist.com/2010/basics-finding-pin-1/

https://forums.raspberrypi.com/viewtopic.php?t=277416