Analysis of an encrypted HDD

J. Czarny, R. Rigo

May 11, 2015

Abstract

The idea of this presentation is to debunk the myth that analyzing the security of hardware is difficult. From the perspective of a software reverse engineer we present the process of studying a hardware-encrypted, PIN secured, external hard drive: the Zalman VE-400.

While the end result of the analysis and attack is not a full success as it was not possible to recover an encrypted drive, it shows that the drive is nevertheless vulnerable by design.

1 Introduction

1.1 The target

Analysing the security of hardware products seems to scare software reversers and hackers. It used to scare us.

But, fortunately, today there is no need to know much about electronics, as most products are just *System on Chip* (SoC), a single integrated circuit with IO and a CPU or microcontroller. All that is needed is a bit of soldering ability, a multimeter, some useful pieces of hardware and a brain.

The subject here is not *embedded* systems in the now common sense of "*small hardware running Linux*" but smaller systems using one or two chips.

Most interesting for software hackers are *encrypted HDD/USB keys*: their functionnality is conceptually simple and they provide an interesting target.

We will focus on the Zalman VE-400 encrypted drive, a consumer enclosure for 2.5" SATA hard drives which supports hardware encryption. It also supports advanced features like mounting ISO files as virtual optical drives. To do so, the user must choose between the ExFAT and NTFS filesystem support, by reflashing the correct firmware.

Encryption is unlocked by a physical PIN pad, which makes it particularly interesting because the absence of unlocking software on the PC means that everything is done on the drive itself.

1.2 Methodology

This article presents the approach we applied on this drive, from basic aspects to in-depth details. It starts with basic analysis of drive functionnality, followed by a PCB analysis. We then explain how we try to access the code running on the board and proceed to do a black box analysis. With the results, we mount a potential attack, which fails. We finally review our research and conclude.

2 Getting started

2.1 Before opening the drive

The idea is to understand the functionnalities and the limitations as best as possible before going into the technical details.

This means:

- activating encryption: PIN is between 4 and 8 digits
- trying to change the PIN: not possible

- disabling encryption: *wipes* the drive (probably deletes the key)
- entering bad PINs: incremental wait between trials

2.1.1 Basic cryptographic checks

Of course, as we are dealing with encryption, we must verify that data seems encrypted and that the (key,IV) pair is not constant. This can be done with simple steps:

1. configure encryption
2. write zeros on the drive
3. remove drive from enclosure
4. read encrypted data (A) directly from the disk (use a normal USB/SATA bridge)
5. verify that the entropy is very high and that the block cipher mode ECB is not used[1]

In our case, the encrypted blocks have high entropy and ECB is apparently not used ; the documentation states AES-XTS is used, which is the current standard mode for disk encryption [8]. We must also check that *resetting* encryption actually changes the encryption key (or IV) and that it is not directly derived from the PIN:

1. using the same VE400, *disable* and *reconfigure* encryption with the same PIN
2. write zeros again
3. ensure that the (raw) encrypted data is different from read A

After verification, our enclosure encrypted the same zeros to a different output, which means that the encryption key (and hopefully not only the IV) is changed everytime the PIN is configured, which is good.

2.1.2 Is encryption tied to the enclosure ?

Before digging deeper, a simple step can give us information about *where* the "secrets"[2] needed to decrypt the drive are stored:

1. put HDD in enclosure A
2. configure encryption with PIN P
3. write zeros
4. put HDD in a new, out of the box, enclosure B
5. analyze behavior

With the VE400, the result is surprising: everything works ! *Putting an encrypted HDD in another enclosure and entering the correct PIN lets you access the data.*

This means that everything needed to decrypt the data is stored on the encrypted disk itself and not in components of the enclosure.

Summary 1

Everything needed to decrypt the drive data is stored on the drive itself. No secret is present in the enclosure.
$\Longrightarrow$ The attacker should be able to brute-force the PIN or extract the encryption key only in software.

[1] *Electronic codebook* preserves patterns, making it the worst mode for encryption [7]. For example, zeroed sectors are readily identifiable.

[2] From now on, "secrets" will always refer to cryptographic data, not user data stored on the encrypted drive.

2.2 Analysing "secrets" stored on disk

As we know that the secrets are stored on the HDD itself, we must find where and how.

The procedure is simple:

1. fill raw drive with zeros
2. enable encryption in enclosure
3. dump raw disk content (outside enclosure)
4. analyse differences

The only part that changes is the end of the drive, ten sectors before the end. We can see two identical blocks with high entropy, 0x300 bytes long:

Data 1 - Block at the end of drive

```
# dd if=/dev/sdf bs=1M skip=476920 | hd
1404c00  fb f4 be fa 14 1a c7 b2  2f a2 a0 df 52 b6 ae bf
1404c10  07 36 f6 50 04 b0 fc fa  c7 cc f7 2d 54 2e 14 5c
1404c20  be 88 1c 55 90 3a 0d 49  61 09 e0 8d a1 83 2d ad
1404c30  e3 61 61 bc fb b5 07 38  fb da e1 0a 4a cb e8 c4
1404c40  00 d7 f7 07 d4 3d 2c fd  1b 61 37 8b 6f a2 d3 69
1404c50  f3 bf b9 64 9a 1f cb d3  28 0d 69 89 5b d6 05 fb
1404c60  80 2b 98 ac ff 54 80 2b  40 57 ce a0 5b be 7b c7
1404c70  a4 5c 90 3c f6 47 2f 68  70 85 3c 52 e2 80 56 bf
[...]
1404ec0  a8 62 20 d8 8f 01 df 6f  1a c1 87 41 5d 75 c9 05
1404ed0  1c a4 d5 22 63 de cb 60  c4 83 ef f2 ca fc 0f 7b
1404ee0  6e f6 e5 76 0d 85 01 65  12 62 59 25 da 0b be c1
1404ef0  72 bf 73 2c 04 62 52 9c  bf 29 28 ee 82 4e 96 0e
1404f00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
1405000  fb f4 be fa 14 1a c7 b2  2f a2 a0 df 52 b6 ae bf
[...]
```

After reconfiguring encryption, with the same PIN, the whole block changes.

What could be stored in this blob ? It is hard to say, but, depending on the cryptographic scheme used to derive encryption keys, the secret may include:

- the AES-XTS encryption keys and IVs
- the PIN hashes
- random IVs for deriving the PIN into encryption keys

So, we know that the sensitive data is stored encrypted at the end of the disk in an opaque encrypted blob. We also know that this blob is decryptable by all Zalman enclosures.

Summary 2 - Goal definition

The main goal is to understand the encryption of "secrets" stored at the end of the disk.
Accessing the decryption code should enable PIN bruteforce or data decryption.
$\Longrightarrow$ We must now analyze the PCB to try to access to the firmwares.

3 PCB

3.1 Identifying components

The board identifies itself as an iodd 2541, from Korea. A little Google search clearly shows that the Zalman (a Korean company) disk is just a rebranding of the iodd one.

Reading the markings on the main components gives us:

- one Fujitsu MB86C311A USB3-SATA controller (1 on figure 1)
- one PIC32MX150F128D MIPS microcontroller (3 on figure 1)

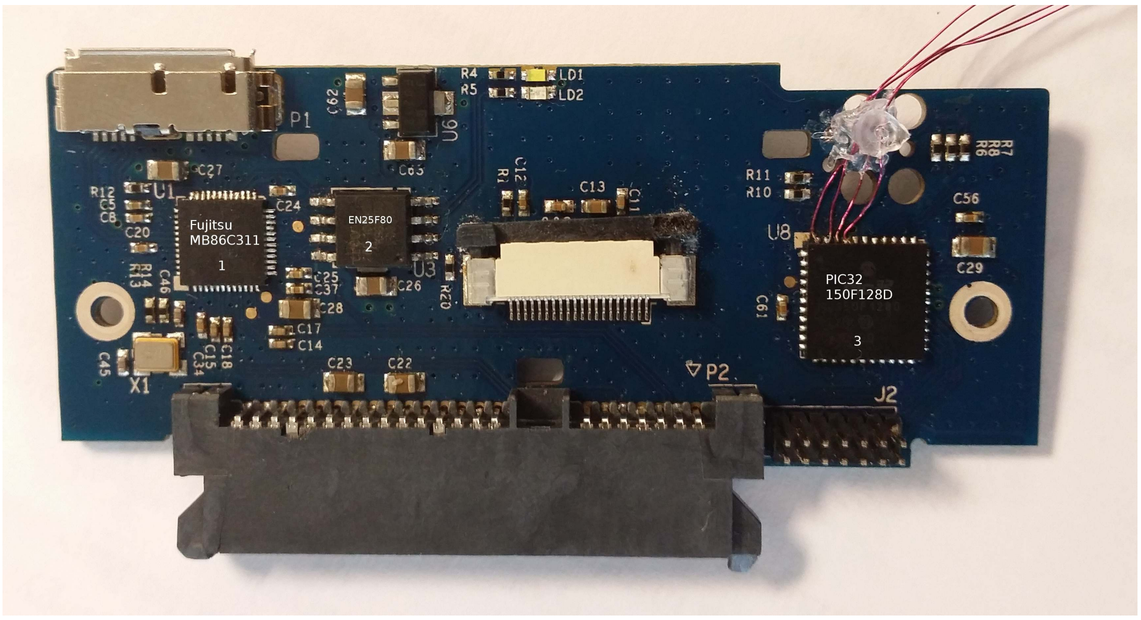


Figure 1: PCB front side

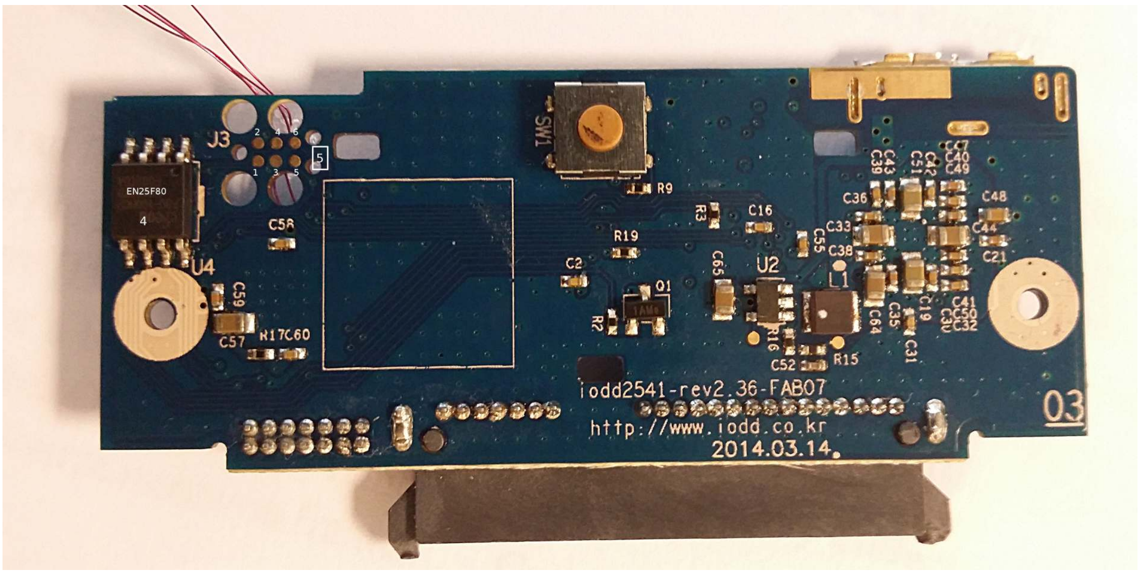


Figure 2: PCB back side

- two EEPROM SPI EN25F80-100HCP flash (2 on figure 1, 4 on figure 2)

Before going further, looking for datasheets is essential, as they will give us information on the features of each chip and, most importantly, the role of each pin.

While finding the datasheets for the PIC and EEPROM is straightforward, the Fujitsu controller is only briefly described on the official site. Luckily, some leaked datasheets are available on Chinese sites[3], mostly for the previous version of the controller, the MB86C30, but we also have the pinout of the MB86C311A.

The Fujitsu datasheet states that the firmware can be customized, in which case it is stored on an external SPI flash chip. Some interestings excerpts from the datasheet:

- "Control Information of Encryption is stored in serial flash."

[3]For an interesting discussion about Chinese sharing, see `http://www.bunniestudios.com/blog/?p=4297`

- “MB86C30 has also additional Encryption function for Firmware itself which is stored in serial flash.”

It may also be interesting to look for other products using the same SoC: the PlayStation 4 or the Buffalo LBU3. They may provide more information *via* firmware updates or manuals.

3.2 PCB traces analysis

To map chips with functionnalities, following the traces on the PCB is essential. Fortunately, this PCB only uses 2 layers, one on each side. Traces are connected from one side to the other by *vias*, caracteristic little holes.

Our primary tools for analysis are:

- a multimeter, in *beep* mode
- GIMP, with aligned pictures of both sides, one per layer, to follow traces which use *vias*

Ideally, we would use an X-Ray machine to get a view of both sides simultaneously. However, with only our basic equipment, we find that[4]:

- the PIC is connected to the keypad using pins 19, 20 and 23 to 27
- the flash on the back is connected to the PIC
- the flash near the Fujitsu is connected to the SPI pins of the SoC
- the PIC is connected to the Fujitsu’s GPIOs by PINs 9,10,42,43,44
- the PIC is connected to the LCD screen
- the PIC is connected to the J3 6-pads connector on the back side

It should come as no surprise that the PIC is responsible for handling display and input. Of course, it must exchange information with the SoC, hence the connections. Both flash chips are probably used for firmware and/or settings storage. While the PIC has an internal flash, its size may not be sufficient.

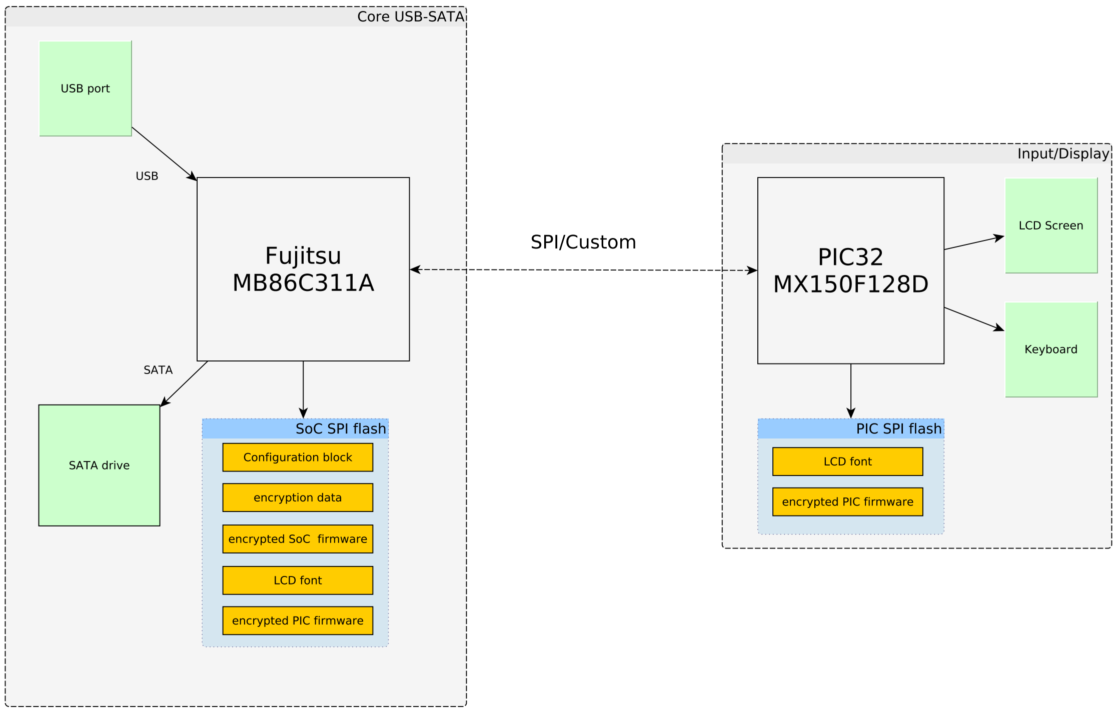


Figure 3: Schematic view of the various components

[4]pin numbers are those from the datasheets.

Summary 3

Now that we know the different components, as summarized on figure 3, we would like to dump the code from the PIC and Fujitsu controllers to complete our understanding.

4 Trying to get the code

As software hackers, our prefered way is to do everything using IDA. So before attacking hardware, we look for easier ways. Luckily, Zalman provides a tool to update their product's firmware, which we will study first.

4.1 Firmware updater

4.1.1 Basics

Hopefully the code will be directly available in the firmware update. On the manufacturer website [2] , two firmware versions are available: one for FAT filesystems and one for NTFS filesystems.

After extraction, and the merge of both versions, the file listing is:

```
config-zalman-ve400-02.ini
font5_worldwide_01.fon
fw-integrity.dll
VE400_firmware_1-48(FAT).bin
VE400_firmware_1-48(NTFS).bin
VE400_micom_1-48(FAT).mic
VE400_micom_1-48(NTFS).mic
zalman-fw-updater-03.exe
zalman-fw-updater-03.ini
```

Unfortunately, the "micom" (probably short for *microcontroller*, so PIC) and SoC firmwares are encrypted.

The PIC one is probably protected with a real block cipher as the two (presumed) very similar firmwares `VE400_micom_1-48(FAT).mic` and `VE400_micom_1-48(NTFS).mic` are completely different

However, the two SoC firmwares are probably encrypted with a stream cipher, as we can see on figure 4.

The `.ini` files also contain interesting parameters:

Extract 1 - INI params

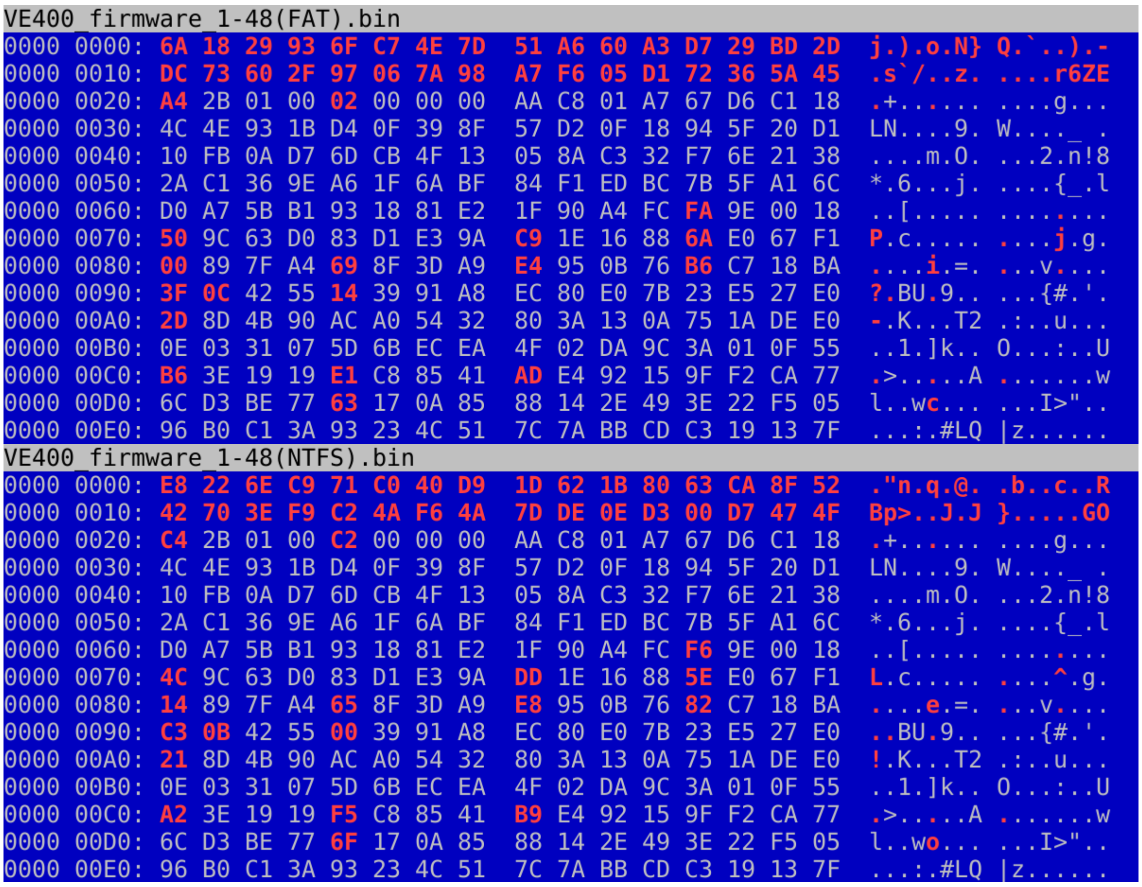
```
[Authentication]
AuthWait=1
DeleteKey=00000000000000000000000000000000

[Control Parameter]
SFlashSecret=1
```

Our hope at this point is that the firmware will be decrypted by the update app and not by the board itself. So we will have to reverse the update process.

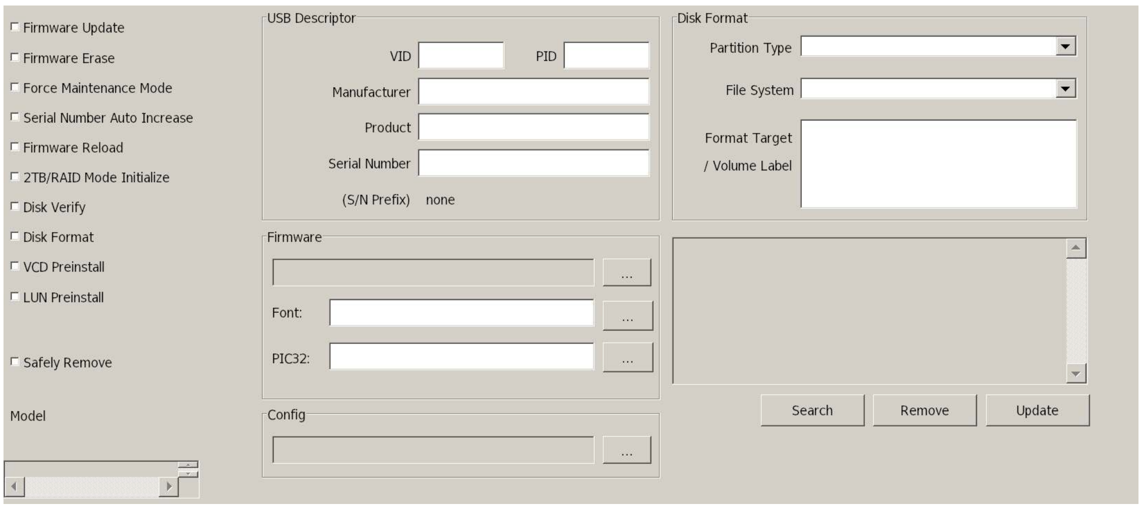
4.1.2 Reversing the update process

Since we are interested only by the hardware reversing process, we will not detail the reversing of the firmware update application.



```
VE400_firmware_1-48(FAT).bin
0000 0000: 6A 18 29 93 6F C7 4E 7D  51 A6 60 A3 D7 29 BD 2D  j.).o.N} Q.`..).-
0000 0010: DC 73 60 2F 97 06 7A 98  A7 F6 05 D1 72 36 5A 45  .s`/..z. ....r6ZE
0000 0020: A4 2B 01 00 02 00 00 00  AA C8 01 A7 67 D6 C1 18  .+...... ....g...
0000 0030: 4C 4E 93 1B D4 0F 39 8F  57 D2 0F 18 94 5F 20 D1  LN....9. W...._ .
0000 0040: 10 FB 0A D7 6D CB 4F 13  05 8A C3 32 F7 6E 21 38  ....m.O. ...2.n!8
0000 0050: 2A C1 36 9E A6 1F 6A BF  84 F1 ED BC 7B 5F A1 6C  *.6...j. ....{_.l
0000 0060: D0 A7 5B B1 93 18 81 E2  1F 90 A4 FC FA 9E 00 18  ..[..... ........
0000 0070: 50 9C 63 D0 83 D1 E3 9A  C9 1E 16 88 6A E0 67 F1  P.c..... ....j.g.
0000 0080: 00 89 7F A4 69 8F 3D A9  E4 95 0B 76 B6 C7 18 BA  ....i.=. ...v....
0000 0090: 3F 0C 42 55 14 39 91 A8  EC 80 E0 7B 23 E5 27 E0  ?.BU.9.. ...{#.'.
0000 00A0: 2D 8D 4B 90 AC A0 54 32  80 3A 13 0A 75 1A DE E0  -.K...T2 .:..u...
0000 00B0: 0E 03 31 07 5D 6B EC EA  4F 02 DA 9C 3A 01 0F 55  ..1.]k.. O...:..U
0000 00C0: B6 3E 19 19 E1 C8 85 41  AD E4 92 15 9F F2 CA 77  .>.....A .......w
0000 00D0: 6C D3 BE 77 63 17 0A 85  88 14 2E 49 3E 22 F5 05  l..wc... ...I>"..
0000 00E0: 96 B0 C1 3A 93 23 4C 51  7C 7A BB CD C3 19 13 7F  ...:.#LQ |z......
VE400_firmware_1-48(NTFS).bin
0000 0000: E8 22 6E C9 71 C0 40 D9  1D 62 1B 80 63 CA 8F 52  ."n.q.@. .b..c..R
0000 0010: 42 70 3E F9 C2 4A F6 4A  7D DE 0E D3 00 D7 47 4F  Bp>..J.J }.....GO
0000 0020: C4 2B 01 00 C2 00 00 00  AA C8 01 A7 67 D6 C1 18  .+...... ....g...
0000 0030: 4C 4E 93 1B D4 0F 39 8F  57 D2 0F 18 94 5F 20 D1  LN....9. W...._ .
0000 0040: 10 FB 0A D7 6D CB 4F 13  05 8A C3 32 F7 6E 21 38  ....m.O. ...2.n!8
0000 0050: 2A C1 36 9E A6 1F 6A BF  84 F1 ED BC 7B 5F A1 6C  *.6...j. ....{_.l
0000 0060: D0 A7 5B B1 93 18 81 E2  1F 90 A4 FC F6 9E 00 18  ..[..... ........
0000 0070: 4C 9C 63 D0 83 D1 E3 9A  DD 1E 16 88 5E E0 67 F1  L.c..... ....^.g.
0000 0080: 14 89 7F A4 65 8F 3D A9  E8 95 0B 76 82 C7 18 BA  ....e.=. ...v....
0000 0090: C3 0B 42 55 00 39 91 A8  EC 80 E0 7B 23 E5 27 E0  ..BU.9.. ...{#.'.
0000 00A0: 21 8D 4B 90 AC A0 54 32  80 3A 13 0A 75 1A DE E0  !.K...T2 .:..u...
0000 00B0: 0E 03 31 07 5D 6B EC EA  4F 02 DA 9C 3A 01 0F 55  ..1.]k.. O...:..U
0000 00C0: A2 3E 19 19 F5 C8 85 41  B9 E4 92 15 9F F2 CA 77  .>.....A .......w
0000 00D0: 6C D3 BE 77 6F 17 0A 85  88 14 2E 49 3E 22 F5 05  l..wo... ...I>"..
0000 00E0: 96 B0 C1 3A 93 23 4C 51  7C 7A BB CD C3 19 13 7F  ...:.#LQ |z......
```

Figure 4: Binary diff of firmware updates



☐ Firmware Update
☐ Firmware Erase
☐ Force Maintenance Mode
☐ Serial Number Auto Increase
☐ Firmware Reload
☐ 2TB/RAID Mode Initialize
☐ Disk Verify
☐ Disk Format
☐ VCD Preinstall
☐ LUN Preinstall
☐ Safely Remove

Model

USB Descriptor
VID PID
Manufacturer
Product
Serial Number
(S/N Prefix) none

Firmware
Font:
PIC32:

Config

Disk Format
Partition Type
File System
Format Target / Volume Label

Search Remove Update

Figure 5: Firmware updater settings dialog extracted from ressources

Just looking at the dialog boxes contained in the resources section is interesting, figure 5 shows that "micom" file most probably is used in the PIC32 field. Which validates our initial hypothesis.

The only interesting result from the reverse process is that the settings read from the ini files are stored in several binary structures that are then transferred to the USB device using standard SCSI-over-USB commands.

The firmware itself is never decrypted, as a basic check with Wireshark and USBPcap could have shown from the start.

Since we could not get the code directly from software, we will have to get it from the hardware itself.

We will start with the PIC, which is easier, in theory.

Summary 4 - Failure, next step

As the firmware updater only contains encrypted firmwares transfered "as-is" to the hardware, we must attack the hardware.

4.2 PIC32MX

Every modern micro-controller contains flash or EEPROM memory to store program code and data. Programming is done by putting the controller in a special mode, usually by running a specific sequence on some pins. Most of them also include a way to protect the code against external reading so that manufacturer can ship microcontrollers to customers without having the code read, mostly to protect against clones.

PIC32MX microcontrollers support two different ways of interacting: standard JTAG and ICSP, the Microchip proprietary *In-Circuit Serial Programming* interface. ICSP has the advantage of only requiring 2 pins for data exchange.

Trace analysis showed that the J3 6-pads header on the back (figure 2) is connected to pins that are used for ICSP access.

The pinout is "standard":

```
 2 o  4 o  6 o          1 : MCLR     2 : VCC (3,3v)
                        3 : GND      4 : PGED
 1 o  3 o  5 o          5 : PGEC     6 : unused
```

Of course, Microchip provides a way of protecting the code stored into the PIC: the configuration bits include a *Code-Protect bit*[5] which "*Prevents boot and program Flash memory from being read or modified by an external programming device.*". This bit can only be cleared to enable the protection, to disable it, the chip must be fully erased.

We will now try to use this ICSP header to check if the PIC is protected.

4.2.1 PICkit 2

We need a PIC programmer to interact with our target from the host. One of the most interesting options is the PICkit 2, which is a cheap (25€) USB programmer for PIC microcontrollers. As its design is open, it is very well supported by open source programs but is now unsupported by Microchip.

So, in theory, we should use the more cumbersome PICkit3 to interface with PIC32MX but, thanks to the opensource `pic32prog` tool [4], we can directly use a PICKit 2 to interact.

As show on the figure 2, the pad/test point[6] for J3 header are big enough to solder wires.

Once small wires have been soldered[7] to the J3 header and interfaced with the PICkit, we can check the *code-protection* status. `pic32prog` displays the dreaded message : "`Device is code protected and must be erased first`" So, protection is enabled. It is even impossible to read the configuration registers :(

Before admitting defeat, we read *PIC32 Flash Programming Specification* datasheet[3] thoroughly but could not find any attack.

[5]DS60001168F-page 226

[6]A portion of exposed metal on the surface of a board usually used for testing or programming during manufacturing.

[7]First time we really have to touch hardware !

4.2.2 Other options

As we know from reversing that the firmware update contains encrypted code for the PIC, its core is upgradable. It most probably uses a bootloader to be reflashed. While we could not find any publicly available bootloaders that implement encryption, some commercial offers exist and may have been used. Potential vulnerabilities may help recover the code.

While some logical attacks exists on smaller PICs, mostly the PIC18F family, which involve erasing parts of the code, no public attack exists for PIC32MX.

In last ressort, as the PIC32MX is not a *secure* chip, it is always possible to use physical attacks to access the code. They usually involve chemical decapping. Quite a few companies offer such services for a fee of several thousands USD.

4.3 Fujitsu MB86C311A

The datasheet of the SoC states that the code can be loaded from an external SPI flash chip. We will look at this option in the next subsection.

Another possibility is an possible JTAG/SWD port. Unfortunately, the datasheet we have does not mention such port. While some pins are named `PTST/TMODE`, which could indicate a *test port*, looking closely at the documentation suggests they are really used to switch between two modes and do not indicate a debug port.

4.4 SPI flash chips

Since we are unable to access the code directly from the chips, we will dump the SPI (Serial Peripheral Interface) flash chips to see if their content is also encrypted.

The Serial Peripheral Interface (SPI) bus is a synchronous serial communication interface specification. SPI devices communicate in full duplex mode using a master-slave architecture with a single master. The SPI bus specifies four logic signals:

- SCLK : Serial Clock (output from master).
- MOSI : Master Output, Slave Input (output from master).
- MISO : Master Input, Slave Output (output from slave).
- SS : Slave Select (active low, output from master).

Unfortunately, dumping SPI flashes directly from the board using a SOIC clip is usually hard, as the current will also power surrounding chips that will interfere. So, unsoldering is needed. However, it is interesting to re-solder the flash on a SOIC to DIP connector in order to keep the board functionnal and usefull for subsequent analysis (see figure 6).

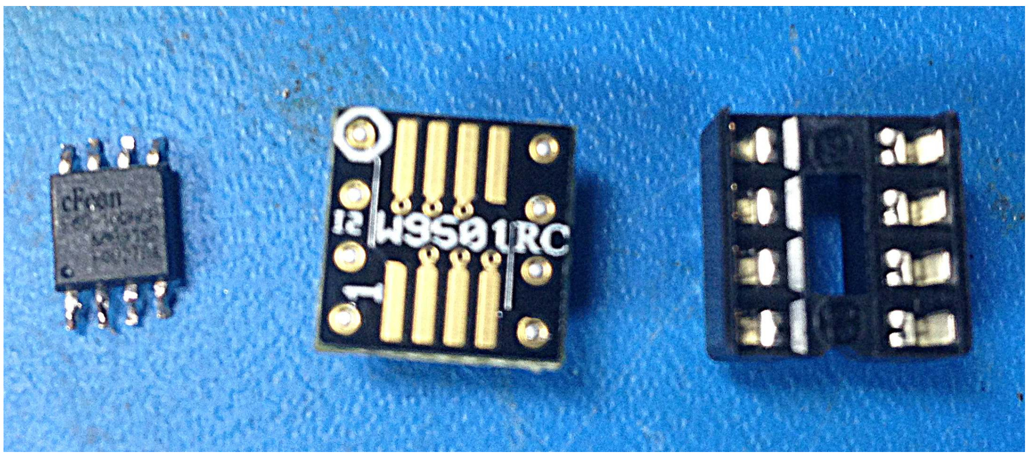


Figure 6: Unsoldered flash

It is important to note that when you unsolder the flash, you should pay attention to the temperature. If the temperature of your soldering iron is too high, metal traces on the PCB could separate from the board and loose connectivity.

Another mistake often made is unsoldering pins one by one by using something to raise the pin. But it frequently leads to breaken pins or to traces disconnected from the PCB. A good solution is to use tweezers soldering iron, as on figure 7.

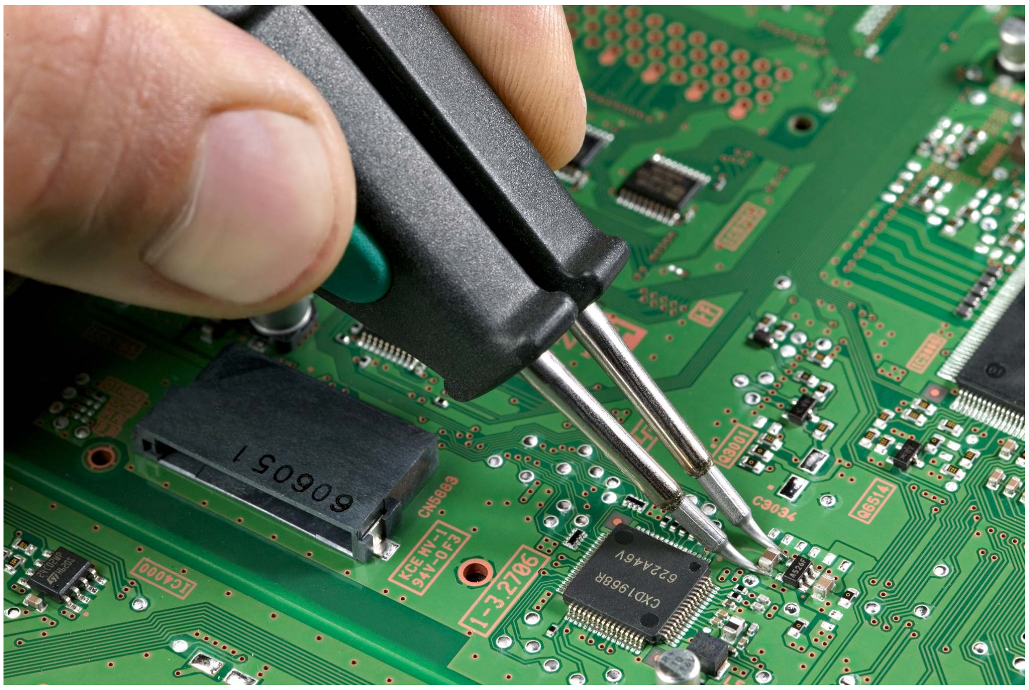


Figure 7: Tweezers soldering iron / Credit : Aisart on Wikipedia (CC-BY-SA 3.0)

So after unsoldering, dumping the chip is just a matter of using a:

- GoodFET with *goodfet.spiflash*
- Bus Pirate via SPI port
- RaspberryPI with *spi.dev*

Just do not forget to put the `HOLD` pin at VCC, otherwise commands may be ignored.

After dumping, we unfortunately note that both chips contain the same encrypted data that we had in the firmware updater:

- the SoC SPI contains : a configuration block, USB descriptors, unknown data, encrypted firmware, font (for LCD display), encrypted PIC firmware
- the PIC SPI contains: font (for LCD display), encrypted PIC firmware

While looking for information on the Fujitsu SoC, we came across information related to the PlayStation 4, which uses a very similar controller, the MB86C311B, which doesn't include encryption. [5] links to a dump of its flash.

Comparing the dump from the PS4 flash with ours shows that:

- the configuration block at the beginning is mostly the same
- USB descriptors differ
- only our dump contains high-entropy data @ 0x1000
- the firmware code seems to be encrypted with the same key

So, we probably have encryption related data at address 0x1000.

Unfortunately, we are stuck, since we could not get any code from our dumps.

Summary 5 - Fail ! Next step

Dumping code from the SoC and PIC failed.
$\Longrightarrow$ We must move to black box, by analyzing communications.

5 Blackbox analysis

Since we have two chips with very different uses, we can probably gain valuable information by sniffing:

- communications between them
- accesses, both read and write, to SPI flashes

This section will focus on describing the process of placing probes and analyzing the signals. Everything will be done using a logic analyzer. We used the Saleae Logic Pro 16, which is both convenient and performant.

5.1 Placing probes

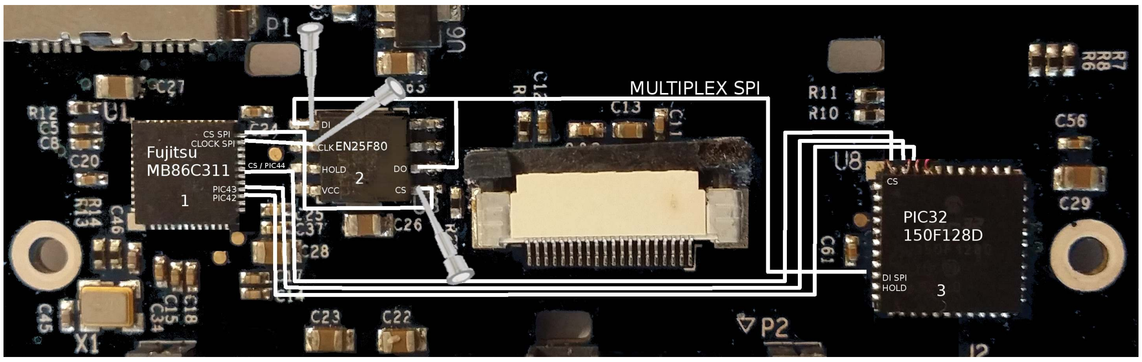


Figure 8: Physical traces and probes placement

Since section 3.2 we have a good idea of which pins are used for communication between the SoCs and for flash access. Some are used for both, which is quite handy because the logic probes are small enough to be directly attached to the SPI pins. Figure 8 shows the important traces and where the probes have been connected: *Chip Select*, *Clock* , and *DI/DO* pins.

Other pins will have to be connected by soldering thin wires (strap wire) to the relevant PIC pins.

Figure 9 shows the kind of wire we used and figure 1 shows the end result when soldered on the board.

Figure 10 shows what it looks like when the probes are placed on the PCB.

5.2 Sniffing and decoding

Once the probes are placed, we just have to setup a trigger, which will determine when to start recording. Obvious options include: the *Chip Select* pin of an SPI flash, or the *clock* pin. We will use the latter.

Before going into details, some words on SPI: it is a simple serial protocol, which uses 4 wires for transfering data [9]:

- /CS: *chip select*, held low to activate the chip we want to talk to
- CLK: the clock, used to sample bits
- DO/MISO: *Data Output* or *Master Input Slave Output* : data coming out of the chip

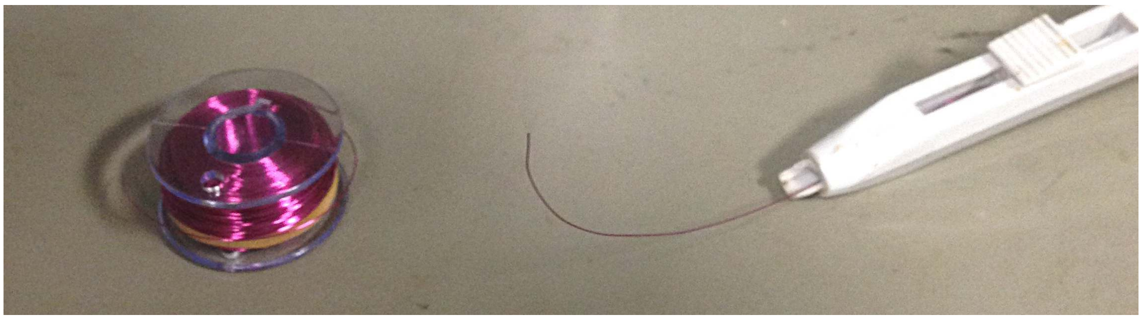


Figure 9: Strap wires

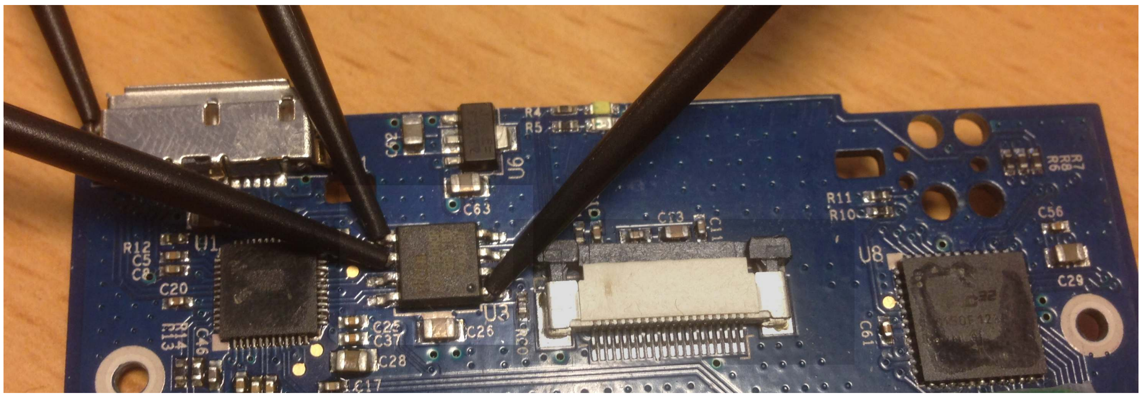


Figure 10: Logic analyzer probes placed

- DI/MOSI: *Data Input* or *Master Output Slave Input*: data arriving to the chip

We know that the bus is shared for communications with the SPI flash and the PIC, so our capture will probably include both.

Figure 11 shows the result in the analyzer window.

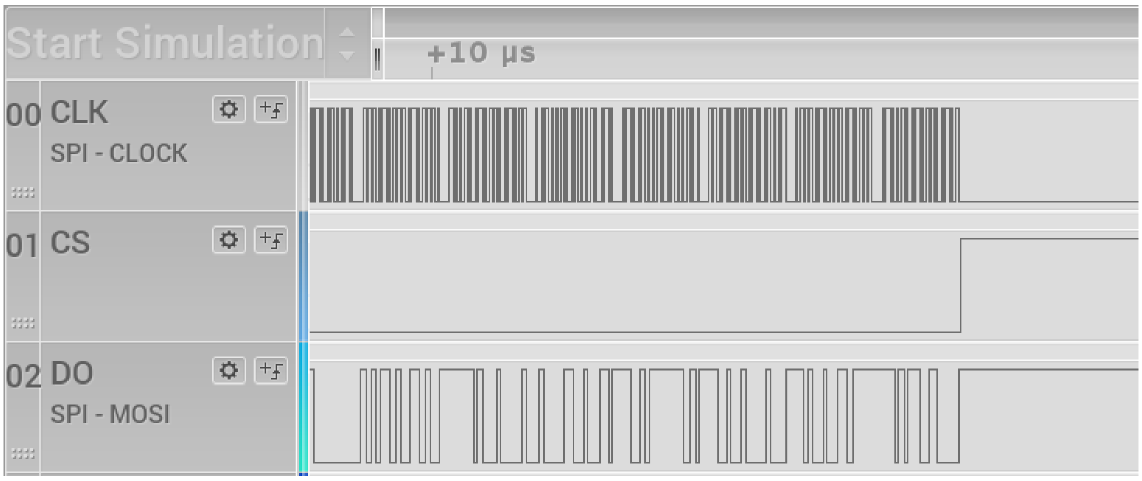


Figure 11: Logic analyzer results

Figure 12 shows a trace with both communications. The red block, on the left, clearly contains communications from the SoC to the flash as the CS pin is brought low (not visible here) for each transfer. While the blue block, on the right, is different: the CS pin is maintained high while the pins CLK, DO, 44 and 43 are active.

Figure 13 shows the beginning of the blue block in details. Analysis is rather straightforward:

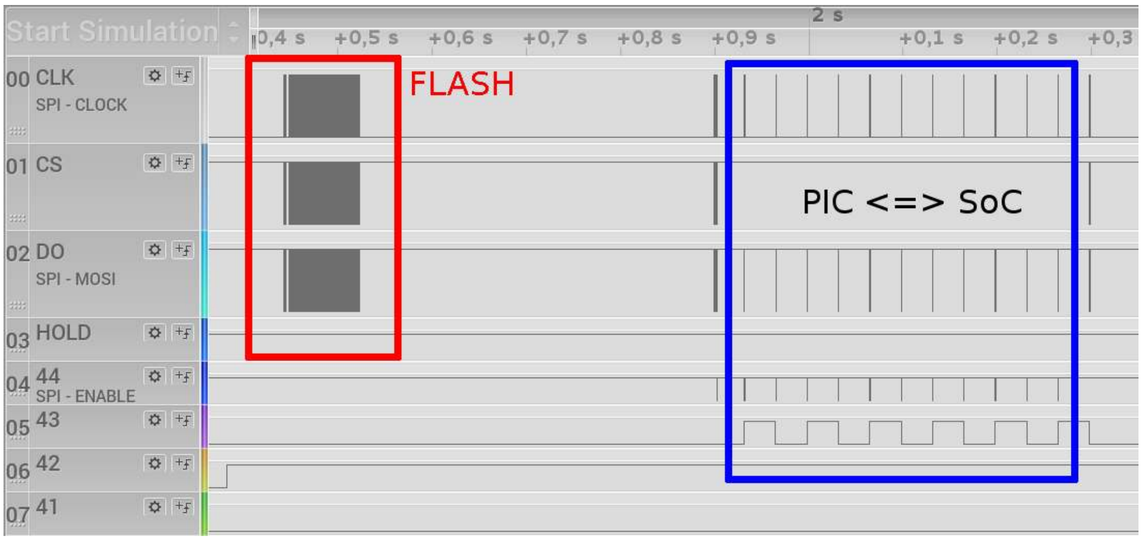


Figure 12: Trace of both flash and PIC↔SoC communications

- CLK (named from the SPI flash pin) is obviously also a clock
- DO is also used for data transfer
- pin 44 is used as a CS for the PIC
- pin 43 is used as a "broader" CS, as seen on figure 12

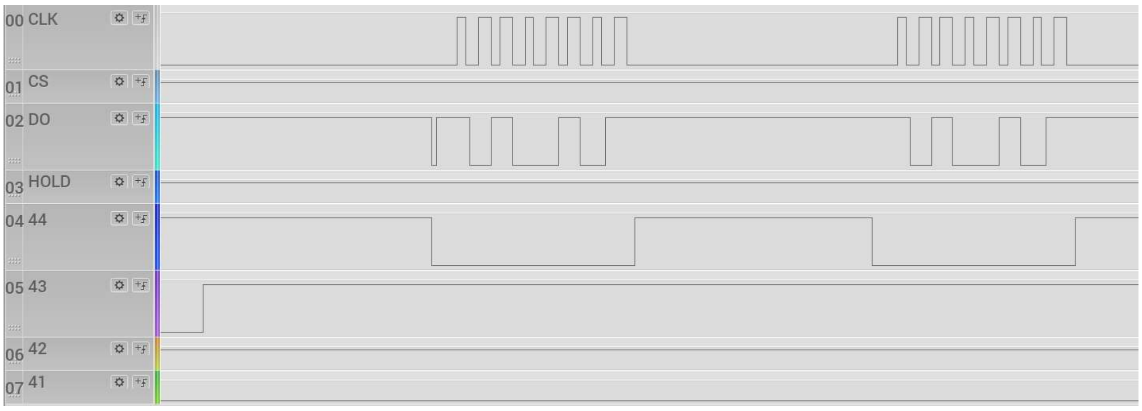


Figure 13: Detail of PIC↔SoC communications

Since the protocol looks like SPI, we will also use the SPI decoder to export the trace to CSV for further analysis.

5.3 Analysing the traces

5.3.1 SPI Flash chips

SPI serves as the underlying protocol for flash memory access. While no official standard seems to exists, most SPI flash memories support the same simple procotol, as specified in the datasheet [1].

While the Saleae software includes protocol decoders, including SPI, it does only decode the transport, and not the flash commands. However, we can export the decoded SPI frames to CSV and decode the flash commands from this input:

Data 2 - Raw SPI flash commands

```
Time [s],Packet ID,MOSI,MISO
0.000000000000000,0,'5' (0x05),'5' (0x05)
0.000000510000000,0,'0' (0x00),'0' (0x00)
```

into commands: READ, RSR, WSR, etc.

A little Ruby script will parse the CSV file using a simple state machine and output the actual commands. The only detail we have to pay attention to is the boundary of commands, in particular for the READ answer which can return an arbitrary amount of data. The CS pin is used in hardware, but as it is not exported in the CSV, we use timing information to detect the end of transfers. The result looks like this:

Data 3 - Interpreted SPI flash commands

```
0.116114 : WREN
0.116118 : PP @ 0x11b8
0x74,0x38,0xac,0x67
0.116124 : RSR : 3
[...]
0.116139 : RSR : 0
---------------------
0.116171 : READ @ 0x1000
0x00,0xff,0x40,0x00,0x26,0xf9,0x9b,0xf3,0x6b,0x45,0xf7,0x11
0.116183 : READ @ 0x100c
0xd4,0x83,0x1b,0x7a,0xce,0x5a,0x2e,0xa2,0x2f,0xe1,0xf2,0x69
```

The following commands are present :

- WREN: *write enable*
- PP: *page program* (write)
- RSR : *read status register*
- READ : read

Since we can decode the addresses of READ commands and get the returned data, we can also dump the binary data to a file, which can be compared with the dumps we acquired by reading the flash chips directly.

5.3.2 PIC

Contrary to the flash chips, we do not know the upper layer protocol used to communicate:

Data 4 - Raw SoC↔PIC protocol

```
Time [s],Packet ID,MOSI,MISO
0.00000000,,'170' (0xAA),'170' (0xAA)
0.00000274,,'170' (0xAA),'170' (0xAA)
0.00000549,,'170' (0xAA),'170' (0xAA)
0.00000823,,'170' (0xAA),'170' (0xAA)
0.00001098,,U (0x55),U (0x55)
0.00001389,,'8' (0x08),'8' (0x08)
0.00001663,,3 (0x33),3 (0x33)
0.00001938,,'20' (0x14),'20' (0x14)
0.00002213,,'1' (0x01),'1' (0x01)
0.00002487,,'1' (0x01),'1' (0x01)
0.00002762,,'16' (0x10),'16' (0x10)
0.00003037,,'1' (0x01),'1' (0x01)
0.00003311,,L (0x4C),L (0x4C)
0.00003586,,'1' (0x01),'1' (0x01)
0.00003860,,'0' (0x00),'0' (0x00)

0.00007927,,'165' (0xA5),'165' (0xA5)
0.00008135,,'165' (0xA5),'165' (0xA5)
0.00008343,,'165' (0xA5),'165' (0xA5)
0.00008556,,Z (0x5A),Z (0x5A)
0.00008775,,'21' (0x15),'21' (0x15)
0.00008988,,'0' (0x00),'0' (0x00)
0.00009201,,'0' (0x00),'0' (0x00)
0.00009415,,'20' (0x14),'20' (0x14)
0.00009628,,'6' (0x06),'6' (0x06)
0.00009841,,'0' (0x00),'0' (0x00)
0.00010055,,'1' (0x01),'1' (0x01)
0.00010268,,'0' (0x00),'0' (0x00)
[...]
```

We have to do a black box analysis of the protocol. We can readily identify two interesting patterns: 0xAA,0xAA,0xAA,0xAA,0x55 and 0xA5,0xA5,0xA5,0x5A which are at the start of each transmission block (pay attention to the timings).

Such patterns are usually syncwords/preambles used to indicate the beginning of a frame. Since the DATA line is used for both directions, each preamble probably corresponds to one direction.

Comparative analysis of several traces allows us to determine the following frame structure:

- preamble (4 or 5 bytes)
- *length* (1 byte)
- *type* (1 byte)
- *frame id* (1 byte)
- *data* (*length-2* bytes)
- *checksum* (2 bytes)

Putting all together in a Ruby script gives us:

Data 5 - Interpreted SoC↔PIC protocol

```
0.0000000 S->P [0x00,0x08]       T: 0x33, ID: 0x14 | 01,01,10,01 (4c01)
0.0000386 P->S [0x0e,0x15]              RESP: 0x14 | 06,00,01,00,61,3f,
                                              8a,27,0e,00,00,00,
                                              44,99,c7,d5 (a300)
```

which shows a request of type `0x33` from the SoC to the PIC (`S->P`) with ID `0x14`, which is answered by the PIC.

Summary 6 - Next step

We are now able to decode most of the communications:

- with the two SPI flash chips
- between the SoC and PIC32

$\Longrightarrow$ we must dig into the details to understand the actual logic

6 Sniff ALL THE THINGS

Since we cannot read the code, we will read the communications and try to determine how the following works :

- valid PIN check
- invalid PIN check (1st try)
- invalid PIN check (Xth try)
- encryption enabling
- encryption disabling
- boot with encryption

Of course we will probably be unable to have the fine details, but we may be able to understand the role of each chip and how the storage capabilities are used.

6.1 PIN checks

6.1.1 Basic check

To understand how PIN verification works, we boot the disk and only start sniffing before pressing the "enter" key. The traffic on the bus after entering "11111111" and "12345678" as PINs is:

Data 6 - PIN validation frames

```
S->P T: 0x33, ID: 0x14 | 01,01,10,01
P->S        RESP: 0x14 | 06,00,01,00,61,3f,8a,27,0e,00,00,00,44,99,c7,d5

S->P T: 0x33, ID: 0x14 | 01,01,10,01
P->S        RESP: 0x14 | 06,00,01,00,8a,62,8f,82,0e,00,00,00,4c,63,d9,65
```

We see here that the communication consists in the SoC asking something to the PIC, which replies with some data.

Looking at the previous trace, the roles of each message is not obvious. Looking at the full trace may help identifying messages, in particular, which message transmits the PIN:

Data 7 - More complete PIN validation frames (good PIN)

```
0.00000000 S->P T: 0x33, ID: 0x14 | 01,01,10,01
0.00003861 P->S        RESP: 0x14 | 06,00,01,00,09,4d,01,cb,
                                    0e,00,00,00,89,0f,3a,7a
1.10261073 S->P T: 0x37, ID: 0x15 | 07,01,01,01
1.10266848 P->S         ACK: 0x15
3.10530806 S->P T: 0x37, ID: 0x16 | 07,02,02,01
3.10534667 P->S         ACK: 0x16
3.10545591 S->P T: 0x37, ID: 0x17 | 05,01,01,01
3.10549452 P->S         ACK: 0x17
3.73653834 S->P T: 0x37, ID: 0x18 | 09,01,01,01
3.73657695 P->S         ACK: 0x18
4.02558845 S->P T: 0x37, ID: 0x19 | 07,41,01,01
4.02562706 P->S         ACK: 0x19
4.02575436 S->P T: 0x33, ID: 0x1a | 01,01,10,01
4.02579297 P->S        RESP: 0x1a | 06,00,01,00,08,07,00,00,
                                    05,00,00,00,00,01,00,00
```

Data 8 - More complete PIN validation frames (bad PIN)

```
0.00000000 S->P T: 0x33, ID: 0x14 | 01,01,10,01
0.00003861 P->S        RESP: 0x14 | 06,00,01,00,77,bd,f8,14,
                                    0e,00,00,00,86,bb,34,9e
0.00392331 S->P T: 0x37, ID: 0x15 | 05,01,01,01
0.00396192 P->S         ACK: 0x15
0.00409148 S->P T: 0x37, ID: 0x16 | 09,04,01,01
0.00413009 P->S         ACK: 0x16
1.00720932 S->P T: 0x37, ID: 0x17 | 0a,01,01,01
1.00724793 P->S         ACK: 0x17
2.01064556 S->P T: 0x37, ID: 0x18 | 0a,01,01,01
2.01068417 P->S         ACK: 0x18
```

It seems that the SoC is requesting the PIC for the PIN (interestingly, we do not see any traffic before we press the Enter key on the keyboard, we do not know why). However, we see that only 8 bytes differ in the response.

Also, data sent by the PIC seem like a deterministic hash of the PIN, as collected samples show:

PIN	Hash
3333	094d01cb890f3a7a
1111	d123a2b469452914
0111	3faf7e23c912063c
0011	5177ab50c9030491
0000	9d11fea0fcc89c77
00000	f1a6b81046b0d834
1234	15fb1e2bd24ecb0b
123456	a4250a552009475e
11111111	613f8a274499c7d5

As the hash is transmitted to the SoC, it is probably in charge of verifying the PIN. The message giving the result (good) to the PIC could be either `07,01,01,01` or `09,01,01,01`. We can also see the different response times : almost instant for bad PIN, a second for a good one.

6.1.2 Bad PIN delays

Sniffing a verification where a bad PIN is entered for the Xth time is interesting:

Data 9 - Bad PIN with delay trace

```
0.00000000 S->P T: 0x37, ID: 0x15 | 0a,01,01,01 (4d01)
0.00003861 P->S         ACK: 0x15 (9100)
0.22340134 S->P T: 0x33, ID: 0x16 | 01,01,10,01 (4a01)
0.22343995 P->S        RESP: 0x16 | 06,00,01,00,3e,c5,
                                    ab,0a,0e,00,00,00,
                                    03,01,34,df (9c00)
0.22733179 S->P T: 0x37, ID: 0x17 | 05,01,01,01 (5001)
0.22737040 P->S         ACK: 0x17 (8f00)
0.22750210 S->P T: 0x37, ID: 0x18 | 09,04,01,01 (4701)
0.22754071 P->S         ACK: 0x18 (8e00)
1.23121708 S->P T: 0x37, ID: 0x19 | 0a,01,01,01 (4901)
1.23125569 P->S         ACK: 0x19 (8d00)
2.23464766 S->P T: 0x37, ID: 0x1a | 0a,01,01,01 (4801)
2.23468627 P->S         ACK: 0x1a (8c00)
3.23808799 S->P T: 0x37, ID: 0x1b | 0a,01,01,01 (4701)
3.23812660 P->S         ACK: 0x1b (8b00)
4.24152672 S->P T: 0x37, ID: 0x1c | 0a,01,01,01 (4601)
4.24156533 P->S         ACK: 0x1c (8a00)
```

The PIN hash is transmitted right away but the PIC is "pinged" by 0x37 frames while the wait is on going.

So by sniffing, we can tell if the PIN was good or bad right away and restart the disk to bypass the increasing delay (see [6] for a similar attack).

Summary 7 - PIN check results

- the SoC is doing the verification
- we cannot easily do a bruteforce attack because of the unknown hash
- check result is known right after entry, delay is bypassable

6.2 Monitoring the SPI flash

While checking the communications between the SoC and the PIC is interesting, combining this information with read and writes to the flashes allows us to have a better overview of the external interactions of both chips.

6.2.1 Good PIN entry

Notably, this is the trace of a successful PIN entry, showing both inter-chip comms and SoC flash accesses :

Data 10 - Good PIN, trace of interchip and flash communications

```
0.00000000 S->P T: 0x33, ID: 0x14 | 01,01,10,01
0.00003861 P->S        RESP: 0x14 | 06,00,01,00,09,4d,01,cb,
                                    0e,00,00,00,89,0f,3a,7a
0.000218 : READ @ 0x1000 (0x94 bytes)
0.000890 : READ @ 0x1094 (0x128 bytes)
0.005659 : READ @ 0x1000 (0x94 bytes)
0.006848 : SECTOR_ERASE @ 0x1000
0.096850 : WREN
0.096854 : PP @ 0x1000
[... repeated sequential PP ... ]
0.099916 : PP @ 0x11b8 (total : 0x1bc bytes)
0.100391 : READ @ 0x1000 (0x1BC bytes)
1.10260833 S->P T: 0x37, ID: 0x15 | 07,01,01,01
1.10264694 P->S         ACK: 0x15
3.10529433 S->P T: 0x37, ID: 0x16 | 07,02,02,01
3.10533294 P->S         ACK: 0x16
3.10546138 S->P T: 0x37, ID: 0x17 | 05,01,01,01
3.10549999 P->S         ACK: 0x17
3.57679572 S->P T: 0x37, ID: 0x18 | 09,01,01,01
3.57683433 P->S         ACK: 0x18
```

When a good PIN is entered, the SoC rewrites 0x1BC bytes at 0x1000, and nothing happens when a bad PIN is entered. So maybe this block contains some crypto related data ?

6.2.2 Enabling encryption

Checking what happens while enabling encryption can give clues. Interestingly the trace show the same behaviour as entering a good PIN :

1. the hashed PIN is sent to the SoC
2. the block at 0x1000 is erased...
3. ...then rewritten

While enabling encryption implies using the last 10 sectors for the enclosure's internal use, the size presented by the drive is still the same. However, trying to read or write those sectors just doesn't work so the SoC probably filters access.

6.2.3 Disabling encryption

Disabling encryption give the following :

1. read of data at 0x1000
2. hashed PIN is sent to the SoC
3. sector erase of 0x1000

Of course the block stored at the end of the HDD is also zeroed and normal access is restored.

6.2.4 SPI block at 0x1000

Analysing the block written at offset 0x1000 in the flash could help understanding its role.

Data 11 - Hexdump of flash block

```
00000000  00 ff 40 00 26 f9 9b f3  6b 45 f7 11 d4 83 1b 7a
00000010  ce 5a 2e a2 25 24 ff e0  00 69 a6 8b ac d5 e9 b7
00000020  66 3e 59 48 3f 52 89 bc  a3 22 84 ab 51 55 ec 1b
00000030  b5 c5 8e 50 ca dc 89 9d  1e 55 78 4f 8d d4 fd 18
00000040  5f 06 17 1b b7 d7 a2 51  c0 0e 17 4c be 98 e1 28
00000050  d5 45 8a 9d 7f 21 04 40  5e 13 91 b8 69 55 be 0c
00000060  8a 35 7d f8 98 0b e0 91  84 f5 53 d9 fb 4b 9c a7
00000070  3b 23 eb d6 da ca 89 ed  81 52 c5 6f 7e 47 62 51
00000080  72 de 26 b8 92 3e d0 6a  31 ad 58 df 76 8c 9e 5a
00000090  43 10 58 75 80 ff 40 00  cb 0b 83 73 47 dc 0d 0f
000000a0  ff 5b 05 92 6c a3 b0 4c  68 84 77 fc 58 ae 92 10
000000b0  28 cb 4a e8 cb 02 a1 90  59 d5 c0 e4 dd 81 c9 70
000000c0  8e 08 88 66 30 b5 a1 95  ad 9a d4 96 f6 87 a5 64
000000d0  d4 f9 e4 1a ac c2 5a a3  ef f1 b8 56 d6 0f c5 d5
000000e0  60 35 3b d7 a3 36 8f 7f  a5 aa 8e 59 cf 03 22 37
000000f0  98 0e a7 31 01 78 8c d4  c0 b9 8b cf f1 a5 a7 6c
00000100  c9 63 65 5b 25 7b 08 96  5e 89 45 4e f5 c8 a5 cc
00000110  a5 df 39 b6 54 c0 18 c5  00 00 20 00 a5 fe 03 88
00000120  63 b6 0d 20 79 63 ac 5d  ce 3c 55 94 85 eb ab bd
00000130  3d 15 6d 67 f6 b5 1c 71  02 a2 c7 79 00 00 00 00
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
00000190  00 00 00 00 00 00 00 00  00 00 00 00 79 cc 89 57
000001a0  4e 94 89 86 93 ee 16 7c  07 02 a7 57 d9 6d 35 c8
000001b0  57 f3 ae 38 14 ae d8 db  0c 2a ee b6
```

The `00 ff 40 00` and `80 ff 40 00` sequences are TLV-style delimiters, also used in the device configuration blocks generated by the firmware updater. The role for the rest of the data is not evident. The two TLV blocks are followed by 512 bits of data, which could represent the two keys needed for AES-256-XTS. We could determine that the last 32 bytes are the SHA-256 hash of the beginning of the block.

We tested the possibility that the block directly contains the AES keys by bruteforcing every possible combination of two blocks of 32 bytes as the keys for AES-XTS and by trying to decrypt sector 0 with AES-XTS. Unfortunately the result was negative.

So, as this data is written after a good PIN entry, we suspect it contains material necessary to decrypt the drive, such as the AES-XTS keys. They are however probably encrypted or obfuscated.

One hypothesis is that the SDK from Fujitsu contained an example for encrypted drives which stores the keys in the SPI flash. That code could have been reused by the developers of the drive.

Summary 8 - Current knowledge

- the SoC verifies the PIN using the hash it receives from the PIC
- if a good PIN was entered a block containing (probably) crypto material is written on the SoC SPI flash at offset 0x1000
- else nothing is written
- the block at offset 0x1000 probably contains encrypted or obfuscated keys for drive encryption

7 Attack Attack !

7.1 Theory

Summing up everything we have seen so far, the boot process seems to be:

1. custom firmware in the SATA controller check if there is an encrypted block at the end of the HDD (10 sectors before the end)
2. if a valid block is found it asks the PIC for a PIN
3. PIC gets the PIN from user
4. PIC hashes the PIN and sends it to SATA
5. SATA validates the PIN against the data previously read at the end of the disk
6. if the PIN is valid, encryption related data is written at offset 0x1000 in the SATA SPI flash
7. encryption processing is enabled

So, since the data necessary to decrypt the disk is probably stored in the SATA SPI flash, we conceived the following attack:

1. attacker setups an enclosure and drive with PIN (named A)
2. get a victim HDD, copy flash + HDD (named V)
3. write flash V to attacker SoC flash
4. make the attacker SPI flash read only
5. build HDD with : victim data and attacker "encryption block" (containing PIN/key A)
6. enter PIN A
7. SoC reads block A to verify PIN A : success
8. SoC tries to write data A in SoC flash but fails
9. SoC reads back block V from SoC flash
10. disk is decrypted using key data from block V

In theory, the PIN should be verified against the data stored at the end of the disk, which we overwrote with a block containing a PIN we chose. But since the SPI flash is read-only, the actual key used for decrypting the victim HDD should remain untouched and used to access the target data.

7.2 Practice

Writing the flash and making it *read only* is just a matter of a few GoodFET commands:

```
goodfet.spiflash dump backup_SPI
goodfet.spiflash erasesector 0x1000
goodfet.spiflash flash boot_key_block 0x1000 0x11BC
goodfet.spiflash writestatus 0x1C
```

Of course, the theory never works at the first try. When we first tried our attack, the key data in the flash was overwritten even though we set it up read-only. Analysing the boot with the logic analyzer showed that the SoC actually resets the status register to 0, removing the protection.

So, to avoid this problem, we just changed the flash content and set the status register *after* boot but *before* entering our known PIN. Figures 14 and 15 show how we setup the attack using a socket for the flash.

But, unfortunately, the attack did not work, the drive did not unlock. So it most probably checked that the data written on the flash was actually written.

Summary 9 - The end

Our black box attack failed. We must access the code.

8 Conclusion

Starting with no information, we managed, in full black box, to have a good understanding of the way this encrypted drive enclosure works.

While we know the crypto design is a fail, because all the encryption related data is stored on the drive itself, with no enclosure dependent secret, we were unable to actually exploit it.

However, accessing the code running on the USB↔SATA bridge would be enough to compromise the security trivially : we could reproduce the decryption of the block at the end of the drive and access the keys or bruteforce the PIN code, which is limited to 8 chars.

To achieve this goal, several options can be considered:

- someone leaks the Fujitsu key used to encrypt the external firmware in the SPI flash, as it is shared by all Fujitsu SoCs
- a vulnerability in ARM or MIPS code : NTFS/exFAT parsing for ISO mounting, SPI parsing, USB parsing ?
- low level hardware attack: chip decapping
- side-channel attacks

Regarding the reverse engineering process we applied in this study, we showed that everything is actually doable in software using relatively cheap tools: Bus Pirate/GoodFET, Saleae, etc. The only hard thing to get is steady hands ;)

And in the end, this process is not that different from reversing an unknown network protocols in black box.

We now hope that you will feel attacking hardware is not that hard :)

References

[1] Eon : EN25F80 datasheet. `http://www.eonssi.com/upfile/p200892918920.pdf`. Accessed: 2015-04-10.

[2] Iodd 2541: firmware updates. `http://crm.iodd.co.kr/projects/iodd2541/files`. Accessed: 2015-04-10.

[3] Microchip: PIC32 Flash Programming Specification (DS60001145P). `http://ww1.microchip.com/downloads/cn/DeviceDoc/cn532867.pdf`. Accessed: 2015-04-10.

[4] pic32prog: Flash programming utility for Microchip PIC32 microcontrollers. `https://github.com/sergev/pic32prog`. Accessed: 2015-04-10.

[5] PS4 developer wiki: MB86C311B. `http://www.psdevwiki.com/ps4/MB86C311B`. Accessed: 2015-04-10.

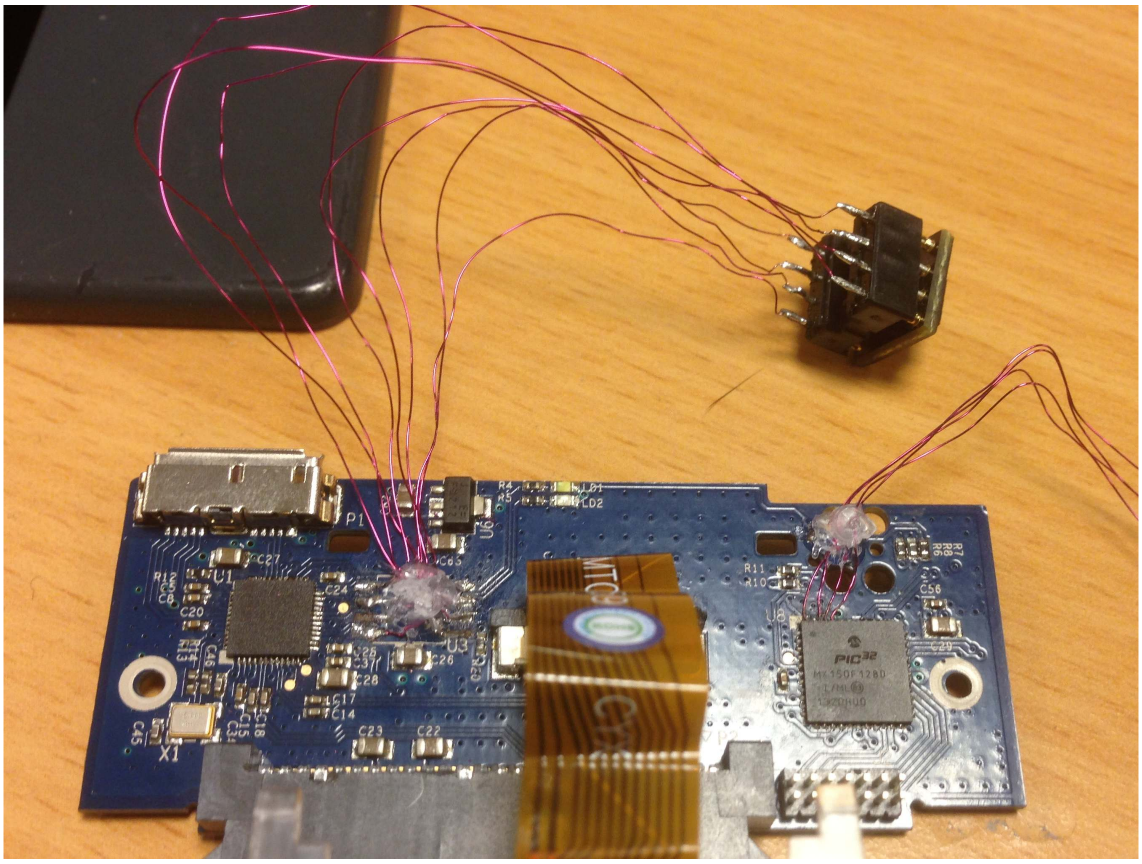


Figure 14: Soldering flash socket

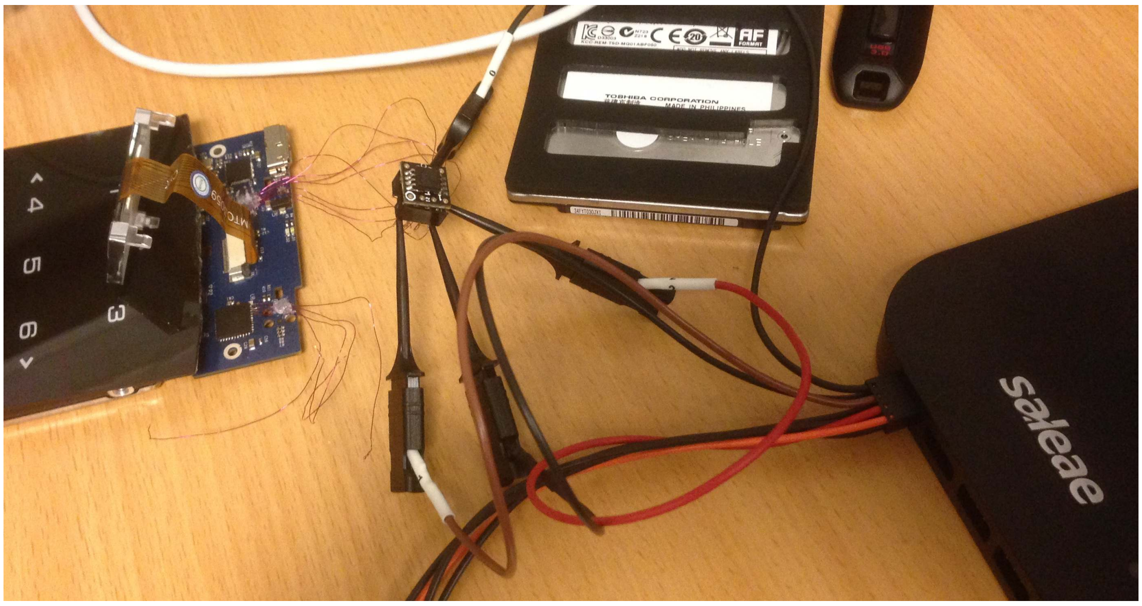


Figure 15: Attack setup

[6] SpritesMods : DiskGenie review - Timing-attack. https://spritesmods.com/?art=diskgenie&page=5. Accessed: 2015-04-10.

[7] Wikipedia: Block cipher modes of operation. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#ECB. Accessed: 2015-04-10.

[8] Wikipedia: Disk encryption theory. https://en.wikipedia.org/wiki/Disk_encryption_theory. Accessed: 2015-04-10.

[9] Wikipedia: Serial Peripheral Interface Bus. https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus. Accessed: 2015-04-10.