

BACHELOR'S THESIS COMPUTING SCIENCE



RADBOD UNIVERSITY NIJMEGEN

Extracting firmware from flash memory chips

Author:
Rick Litjens
s1042847

First supervisor/assessor:
Dr I.R. Buhan

Daily supervisor:
MSc A. Adhikary

Second assessor:
Dr E. Poll

June 19, 2023

Abstract

Nowadays more and more devices are connected to the Internet. All these devices have firmware. Having vulnerabilities in this firmware can result in serious security and privacy risks. In this thesis, our aim is to extract the firmware of three identical portable routers with flash chips from different manufacturers and find out if the manufacturer is important in this process. We show how we can do this in three different ways. Using Flashrom, a utility for exactly this purpose, using alternatives to Flashrom in case a flash chip is not supported by it and by extending the flash chip support of Flashrom. By doing this, we answer our research question: Is the manufacturer important for extracting the content of different flash chips of modern routers? We found that the manufacturer is not important for extracting firmware from flash chips.

Contents

1	Introduction	3
2	Preliminaries	5
2.1	Routers	5
2.1.1	Winbond 25Q64JVSIQ	5
2.1.2	Gigadevice 25Q64CSIG	6
2.1.3	XMC QH64AHIG	6
2.2	Programs	7
2.2.1	Flashrom	7
2.2.2	PuTTY	7
2.2.3	PySerial	7
2.2.4	Ghidra	7
2.3	Serial protocols	8
2.3.1	SPI	8
2.3.2	I ² C	8
3	Research	9
3.1	Dumping firmware of supported chips	9
3.1.1	Identifying the chip and pins	9
3.1.2	Communicating with the chip	11
3.2	Alternatives to Flashrom	11
3.2.1	PuTTY	11
3.2.2	PySerial	12
3.3	Manually adding support to Flashrom	13
3.3.1	Changing files	13
3.3.2	Recompiling Flashrom	15
4	Related Work	16
5	Conclusions and future work	18
A	Python script read_image.py	22
B	Ghidra's decompile function	27

Chapter 1

Introduction

Modern society relies heavily on the Internet. It is being used almost everywhere: from education and online banking to hospitals and streaming services. We increasingly want to be online everywhere around the world and several solutions exist to fulfill this need. One of them being portable routers. A portable router is a small device that you can take everywhere you go to create a wireless network to connect your devices to the Internet. All it needs is power and a SIM card to connect to mobile networks such as 4G and 5G. However, these devices can not always be trusted since there may be vulnerabilities in their firmware. To discover if such vulnerabilities exist, we can for example look at the firmware currently on the router and compare it to the manufacturer's latest version to see what has changed.

Finding vulnerabilities in the firmware of different portable routers is out of scope of this thesis. Instead, we will look into how we can extract the current firmware of different portable routers. Extracting data from such a device can yield interesting security findings, such as backdoors, encryption keys, secret accounts, and so on.[13] This will be done with a Bus Pirate, which is an open-source hacker multi-tool that talks to most chips from a PC serial terminal. Many serial protocols are supported at 0-5.5V, such as SPI and I²C. [1] We will be using the Bus Pirate in combination with Flashrom, which is explained further in Chapter 2. While doing this, we try to answer the research question: Is the manufacturer important for extracting the content of different flash chips of modern routers? To accomplish this, we look into the following:

- We show how to extract firmware via the SPI protocol from a flash chip supported by Flashrom.
- We look at two alternatives to Flashrom, PuTTY and PySerial and demonstrate that PySerial is a viable alternative.
- We look at extending Flashrom's flash chip support.

We start with some preliminary knowledge on used hardware, software and serial protocols in Chapter 2. Next, we look at the actual research conducted in Chapter 3. Then, we compare our work with existing work (Chapter 4) and conclude this thesis with a summary of the research and a look at future research in Chapter 5.

Chapter 2

Preliminaries

2.1 Routers

In this paper, three routers with various flash chips were investigated. Here we will briefly describe them and discuss their similarities and differences.

2.1.1 Winbond 25Q64JVS1Q

Our first router has the Winbond 25Q64JVS1Q flash chip where its firmware is stored, which is visible on the right image in Figure 2.1. It has the Mediatek MT7628NN router as SoC (System-on-a-Chip), the EMST M14D5121632A as RAM and the GROUP-TEK HST-0041JAR as Ethernet transformer chip. Such a transformer chip is required to send and receive Ethernet signals over twisted pair cables (these are often used for Ethernet connections), because otherwise the cables are susceptible to interference, noise and voltage differences. These are all visible on the left image in Figure 2.1.

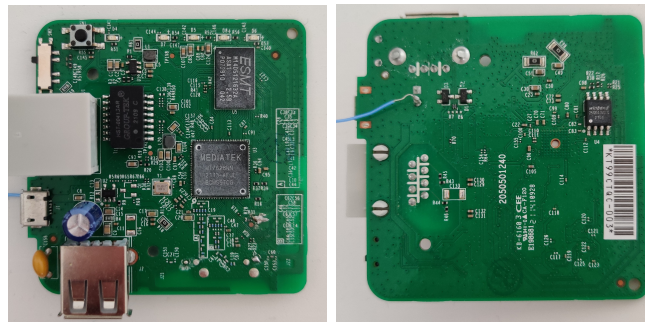


Figure 2.1: Router including the Winbond chip

2.1.2 Gigadevice 25Q64CSIG

Our second router has the Gigadevice 25Q64CSIG flash chip as storage for its firmware, as can be seen in the right image in Figure 2.2. It also has the Mediatek MT7628NN router as SoC. Furthermore, it has the Zentel A3R12E40DBF as RAM and the Bourns SMD 16001 as SMD fuse that provides one-time fuse protection. This is different from the previous router since that one has an Ethernet transformer chip at the place of the fuse. However, this is not important for our thesis since it does not influence the flash chip. All these chips can be seen in the left image in Figure 2.2.

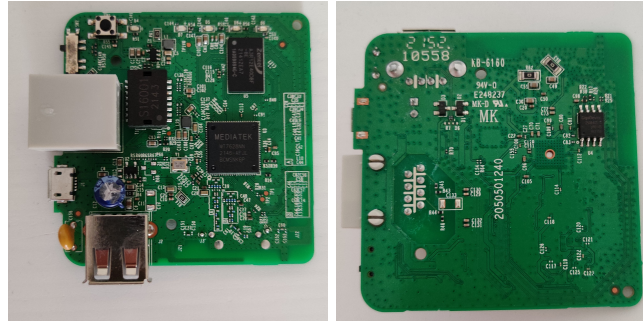


Figure 2.2: Router including the Gigadevice chip

2.1.3 XMC QH64AHIG

Our third and last router has the XMC QH64AHIG flash chip as firmware storage, this is shown in the right image in Figure 2.3. Like the other two routers, it also has the Mediatek MT7628NN router as SoC, and the JWD S16037G as Ethernet transformer chip. These can be seen in the left image in Figure 2.3. There is no RAM chip visible.

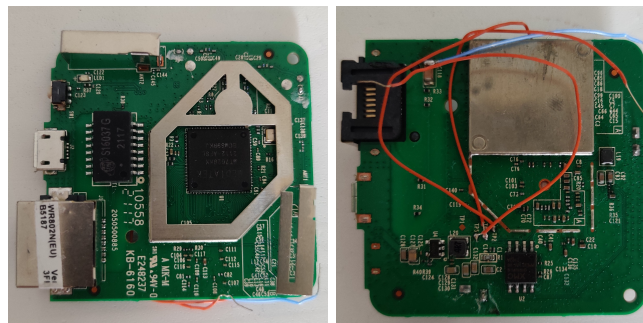


Figure 2.3: Router including the XMC chip

2.2 Programs

In this thesis, several open-source programs are being used. They are briefly described here.

2.2.1 Flashrom

Flashrom is a utility for identifying, reading, writing, verifying and erasing flash chips. It is designed to flash BIOS/EFI/coreboot/firmware/optionROM images on mainboards, network/graphics/storage controller cards, and various other programmer devices. [5] It is used to read the contents of flash chips.

2.2.2 PuTTY

PuTTY is a free implementation of SSH and Telnet for Windows and Unix platforms, along with an xterm terminal emulator. It is written and maintained primarily by Simon Tatham. [7] It can be used to establish a serial connection with the Bus Pirate and send commands directly over this connection. We used PuTTY 0.78, which is the most recent version at the time of writing.

2.2.3 PySerial

PySerial is a Python module that encapsulates the access for the serial port. It provides backends for Python running on Windows, OSX, Linux, BSD (possibly any POSIX compliant system) and IronPython. [11] It is used in a Python script to read the contents of flash chips.

2.2.4 Ghidra

Ghidra is a software reverse engineering (SRE) framework created and maintained by the National Security Agency Research Directorate. This framework includes a suite of full-featured, high-end software analysis tools that enable users to analyze compiled code on a variety of platforms including Windows, macOS, and Linux. Capabilities include disassembly, assembly, decompilation, graphing, and scripting, along with hundreds of other features. [4] Its decompilation functionality is used in this paper to check if a flash chip reading was successful. An example of this is shown in Appendix B.

2.3 Serial protocols

2.3.1 SPI

SPI is a communication protocol that transmits data between peripherals and microcontrollers. Found in popular hardware like the Raspberry Pi and Arduino, it is a synchronous communication protocol, which means it can transfer data faster than I²C and UART. Often it is used for short-distance communications in places where read and write speeds matter, such as in Ethernet peripherals, LCD displays, SD card readers and the memory chips on almost any IoT device. [13]

2.3.2 I²C

Pronounced “I squared C”, I²C is a serial communication protocol for low-speed devices. Philips Semiconductors developed I²C in the 1980s for communications between components on the same circuit board, but you can also use it between components connected via cable. In the IoT world, you will often find it in microcontrollers, I/O interfaces like keyboards and buttons, common household and enterprise devices and sensors of all types. Crucially, even the sensors in many Industrial Control Systems (ICS) use I²C, making its exploitation high stakes.

The main advantage of this protocol is its simplicity. Instead of the four wires that SPI uses, I²C has a two-wire interface. In addition, the protocol allows hardware without built-in I²C support to use I²C through general-purpose I/O pins. But its simplicity, and the fact that all data travels over the same bus, makes it an easy target if you want to sniff or inject your own data. The reason is that no authentication occurs between components in IoT devices sharing the same I²C bus. [13]

Chapter 3

Research

We looked into the following:

- First, we extracted the firmware of a flash memory chip that is already supported by the Flashrom utility. This was done using the SPI protocol. On Flashroms supported hardware page [6], we see that the Winbond 25Q64JVSQ is listed as W25Q64.V, which is the chip that we used.
- Then we looked at alternatives to Flashrom for dumping the firmware of flash chips. We looked at two options, PuTTY and PySerial. This has been done for all three flash chips we used.
- After that, we looked at extending Flashrom’s flash chip support. Since Flashrom is open-source, it is possible to add support for new (currently unsupported) flash chips. We attempted to do this with the XMC QH64AHIG chip, but sadly we were unsuccessful.

3.1 Dumping firmware of supported chips

We started with dumping the firmware of an already supported chip. As stated in the introduction, we used the Bus Pirate v3.6a by Sparkfun in combination with Flashrom v1.3.0 for Windows. We used the book by Chantzis et al. [13] as guidance through this process.

3.1.1 Identifying the chip and pins

To be able to extract the firmware of our router, we first identified its flash memory chip. These typically have 8 or 16 pins, but can also be found by looking up the microcontroller’s datasheet online. In our case it was the Winbond 25Q64JVSQ.

To find out the pin layout, we looked at the datasheet [8] of this particular chip. A flash chip’s orientation is based on a small dot in one of the corners.

The pin at this particular corner is pin 1. In Figure 3.1 we can see that the dot is at the top-left corner. This corresponds to the pin layout in the data sheet shown in Figure 3.2:

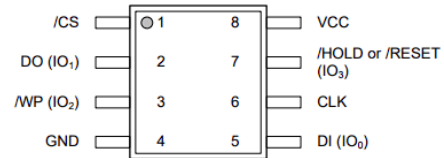
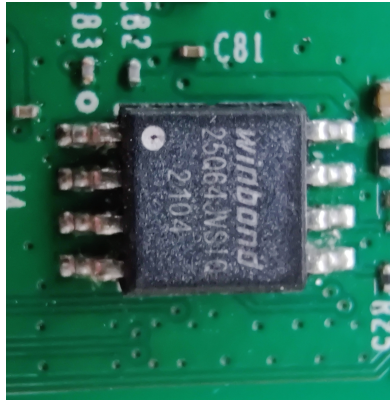


Figure 3.2: Pin layout of the Winbond chip

Figure 3.1: Flash chip inside our router

For the connection diagram, we consulted the book by Chantzis et al. [13]. Once connected, our setup looked like this:

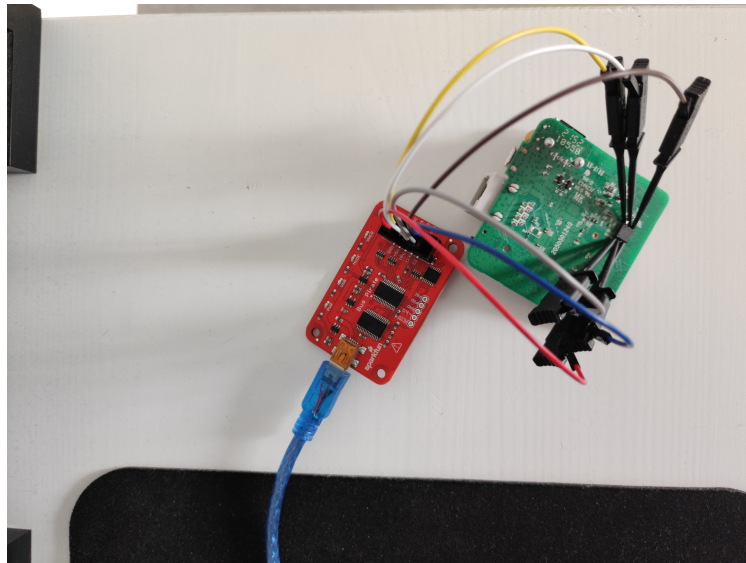


Figure 3.3: Connection between chip and Bus Pirate

3.1.2 Communicating with the chip

To be able to communicate with the Bus Pirate, its assigned device descriptor is needed. On Windows this is a COM-port. A COM-port is an I/O interface that enables the connection of a serial device to a computer. In our case the Bus Pirate was on COM6, resulting in the following command:

```
flashrom -p buspirate_spi:dev=COM6 -r out.bin
```

The `-p` flag specifies the programmer device. The Bus Pirate's name in this context is `buspirate_spi` and it is on COM6. The `-r` flag issues a read flash operation that saves to the specified file `out.bin`. This binary file can be used for reverse engineering with programs like Ghidra [9] or tools like `binwalk`. To check whether the read operation was correct, we opened the `out.bin` file in Ghidra and used its codebrowser in combination with the decompile functionality to turn the raw memory file into C-functions. This is shown in Appendix B.

3.2 Alternatives to Flashrom

Although Flashrom is probably one of the easiest programs for communicating with serial devices and acquiring their firmware, there are several alternatives. These can be used when you have a chip that is not supported by Flashrom. In this section, we take a look at these alternatives and tested both of them for all three flash chips we used.

3.2.1 PuTTY

When using PuTTY we can talk directly to the flash chip via the SPI protocol. To set up the connection, we used the correct COM-port (COM6) and baud rate (115200) and the following configuration settings:

- Speed: 1MHz
- Clock polarity: Idle low
- Output clock edge: Active to idle
- Input sample phase: Middle
- CS: /CS
- Output type: Normal (H=3.3V, L=GND)

These are all the default settings except speed, which is the highest possible.

Data can be read with the `r`-command. To use commands, first CS (chip select, one of the connected pins) needs to be pulled low, this activates the

chip and is done with [. Then the desired command is entered and finally CS needs to be pulled high again, which is done with]. This results in the following command to read one byte: [r].

When sending this, we only got 0x00 as response.

```
SPI>[r:10]
/CS ENABLED
READ: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
/CS DISABLED
```

Figure 3.4: Response when reading with PuTTY

To test if anything was working we sent the device and protocol-specific command for reading the device ID, which can be found in the datasheet for each chip, however this returned a syntax error. When sending any read commands we only got 0x00 as response as shown in Figure 3.4. This was the case for all three flash chips. This brings us to the conclusion that either the connection is not working properly or we cannot send commands in this way. To make sure that the physical connection was correct and working, we extracted the firmware of the Winbond chip again using Flashrom without problems. Since the other chips are connected in exactly the same way, we can conclude that the physical connection is working as intended. This results in the conclusion that using PuTTY we cannot send commands directly in the way we tried.

3.2.2 PySerial

When looking online into this module, we came across a Python script by Arno Wagner [20]. This is included in Appendix A for convenience. It first tests the list of candidate ports and finds the correct one. It then sets up the SPI connection in the same way as we did above. Finally, the power supplies are activated and the content of the connected chip is read. To use this script, it needed some slight adjustments because it is intended for Linux and we used Windows. The `candidates` list had to be extended since Windows uses COM ports, so in our case we had to add 'COM6' to the list. Also, we commented out the following line:

```
s = serial.Serial(port=n, baudrate=115200, timeout=0.01)
```

This statement was superfluous because `s` was defined already which resulted in trying to open the same port twice.

After this the script could be executed without problems and a small portion of the memory (512 kB) was extracted. We checked this again using Ghidra's codebrowser and decompile functions to verify that the read operation happened correctly. We repeated this for all three flash chips.

Although this worked, it was really slow and read only a small portion of the complete memory. Since we are looking into extracting the complete content of flash chips, we changed the script to read the entire memory. Only one change was necessary to be able to read the full 8 MB of our chip:

```
for adr in range(0, 0x80000, read_size):
```

↓

```
for adr in range(0, 0x800000, read_size):
```

This is sufficient because $8 \text{ MB} = 8.388.608 \text{ bytes} = 0x800000$ in hexadecimal, so this covers the entire address space of an 8 MB flash chip. We ran this script thrice, the first time with the Winbond 25Q64JVSQ chip that we used before with Flashrom and the second and third times with the XMC QH64AHIG chip and the Gigadevice 25Q64CSIG chip that we were not able to read since they were not supported by Flashrom. Using this script, we were able to read all chips completely. However, this takes quite some time. On our machine (Windows 11 Pro, Intel Core i5-9600K, 16GB RAM) it took around 2 hours to completely read one chip, whereas with Flashrom it took only a few minutes. This is why we also looked into extending the Flashrom support manually to be able to read our chips that are not supported (yet) in a way faster manner.

3.3 Manually adding support to Flashrom

Since Flashrom is an open-source software, we have access to its source code. This should allow us to extend its chip support by manually adding our own chip with the correct configuration to the list of supported chips. We tried this for the XMC QH64AHIG chip. During this process, we followed the *Adding/reviewing a new flash chip* guidelines on the Flashrom website [2].

3.3.1 Changing files

According to the guidelines, the only files that we needed to change were `flashchips.h` and `flashchips.c`. The first file contains a list of all supported chips and their corresponding IDs. After looking up the device ID in the data sheet [10], we added the following line in `flashchips.h`:

```
#define XMC_XM25QH64A 0x7017
```

This information is used by Flashrom to check which chip is connected to the Bus Pirate.

In `flashchips.c` all chip specifications are listed such as names, sizes and voltage ranges. Flashrom needs this information to communicate with the chips after they have been identified with the information from `flashchips.h`. We added the following entry to the giant array in `flashchips.c`:

```

{
    .vendor      = "XMC",
    .name        = "XM25QH64A",
    .bustype     = BUS_SPI,
    .manufacture_id = ST_ID,
    .model_id    = XMC_XM25QH64A,
    .total_size  = 8192,
    .page_size   = 256,
    .feature_bits = FEATURE_WRSR_WREN | FEATURE_OTP | FEATURE_QPI,
    .tested      = TEST_UNTESTED,
    .probe       = PROBE_SPI_RDID,
    .probe_timing = TIMING_ZERO,
    .block_erasers =
    {
        {
            .eraseblocks = { {4 * 1024, 2048} },
            .block_erase = spi_block_erase_20,
        }, {
            .eraseblocks = { {32 * 1024, 256} },
            .block_erase = spi_block_erase_52,
        }, {
            .eraseblocks = { {64 * 1024, 128} },
            .block_erase = spi_block_erase_d8,
        }, {
            .eraseblocks = { {8 * 1024 * 1024, 1} },
            .block_erase = spi_block_erase_60,
        }, {
            .eraseblocks = { {8 * 1024 * 1024, 1} },
            .block_erase = spi_block_erase_c7,
        }
    },
    .printlock = spi_prettyprint_status_register_plain,
    .unlock    = spi_disable_blockprotect,
    .write     = SPI_CHIP_WRITE256,
    .read      = SPI_CHIP_READ,
    .voltage   = {2700, 3600},
},

```

We found that the XMC chip is actually quite similar to our Winbond chip and even has the same device ID, so we copied the entry for the Winbond 25Q64JVS1Q (included in Appendix C for convenience) and adjusted it to the specifications of the XMC QH64AH1G. The following entries were changed:

- Obviously the vendor and name have to be changed:

```

.vendor = "Winbond"    → .vendor = "XMC"
.name   = "W25Q64JV-.M" → .name   = "XM25QH64A"

```

- We derived the manufacture and model id from other XMC entries in the array:

```
.manufacture_id = WINBOND_NEX_ID  → .manufacture_id = ST_ID
.model_id = WINBOND_NEX_W25Q64JV  → .model_id = XMC_XM25QH64A
```

- Our XMC chip was not tested to work with real hardware yet:

```
.tested = TEST_OK_PREW → .tested = TEST_UNTESTED
```

- The printlock function is a function to print (write) protected address ranges of the chip, but also to print the values of the status register(s) in a clear way. According to the guidelines we should reuse an existing function if possible, so we use the same as for all the other XMC chips:

```
.printlock = spi_prettyprint_status_register_bp2_tb_bpl
                ↓
.printlock = spi_prettyprint_status_register_plain
```

- The unlock function is called before Flashrom wants to modify the chip's contents to disable possible write protections. We used the same reasoning here as the printlock function:

```
.unlock = spi_disable_blockprotect_bp2_srwd
                ↓
.unlock = spi_disable_blockprotect
```

3.3.2 Recompiling Flashrom

To use the extended files, Flashrom needed to be recompiled from source. We used the Flashrom Windows installation page [3] as guidance for this step. In order to build Flashrom and various of its dependencies, we needed a UNIX-like environment on Windows, which is provided by MinGW/MSYS. After installing all the necessary programs and dependencies, we rebuilt Flashrom as explained on the Flashrom Windows installation page.

However, the newly built Flashrom executable did not recognize our XMC chip. Also, during the build process many warnings about missing dependencies appeared. Since Flashrom's support for Windows is very limited, we also rebuilt it on an Ubuntu Linux virtual machine using our altered files. This gave no errors, however also this newly built executable did not recognize the XMC chip. Thus, we were unable to successfully extend Flashrom's support.

Chapter 4

Related Work

A lot of work has been conducted on the security of Internet-connected devices, especially in the last years in the field of the Internet of Things. Previous work looked at the state of the security of this wide range of devices [16] [21] [15], and also found a wide range of vulnerabilities when analyzing the firmware of these devices [14] [18]. However, much of this previous work looked at firmware downloaded from the Internet, while we are interested in obtaining it from a device directly. This results in the hardware security of these types of devices getting less attention than the security of their software.

However, there has been done some work on hardware security as well. For example, the paper by Vasile et al. [19] looks at different methods to extract the firmware of IoT devices. They show that extracting firmware from IoT devices is possible through a variety of low-cost methods, with over 45% of the considered devices vulnerable to extraction through a simple UART¹ connection. They also show that this problem exists throughout the industry, affecting high-profile devices like the first generation Amazon Echo as well as home hubs and alarm systems with significant security and privacy implications.

The paper by Su and Ranasinghe [17] also looks at hardware security. They focus on demonstrating the: i) exploitation of debug interfaces, often left open after manufacturing; and ii) the exploitation of exposed memory buses. They illustrate a person could commit such attacks with entry-level knowledge, inexpensive equipment, and limited time (in 8 to 25 minutes).

In paragraph 6.3 of the paper by Borg and Francke [12] dumping firmware directly from a device is discussed. They use a similar approach to our paper,

¹Universal Asynchronous Receiver-Transmitter; a computer hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable. It sends data bits one by one, from the least significant to the most significant, framed by start and stop bits so that precise timing is handled by the communication channel. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

utilizing a Bus Pirate and Flashrom, communicating over the SPI protocol. A Python script is used as well to read the contents of the EEPROM in a somewhat similar manner as we did, the biggest difference being that I²C is used instead of SPI.

Chapter 5

Conclusions and future work

In this thesis, we extracted the firmware of three routers with different flash chips using a Bus Pirate. This was done in several ways.

First, we dumped the firmware using the Flashrom utility. We did this with the Winbond flash chip since that chip was already supported by Flashrom.

Then, we looked at alternatives to Flashrom for extracting firmware from flash chips and at two options in particular. The first one was direct communication over the SPI protocol using PuTTY. This did not work and returned only 0x00 and syntax errors. The second one was using a Python script that utilizes the Python module PySerial. After slight modification, this worked for all three flash chips.

However, it was really slow which is why we also looked at extending the flash chip support by Flashrom. It is an open-source software, so we have access to its source code. This allowed us to manually add our own XMC chip to the list of supported chips, but despite our effort Flashrom was still unable to identify the XMC chip.

A summary of our findings is in the table below. It shows for each flash chip and each method whether we were able to perform a successful read or not:

	Flashrom	PuTTY	PySerial	Extended Flashrom support
Winbond 25Q64JVSIQ	Yes	No	Yes	Not tested
Gigadevice 25Q64CSIG	No	No	Yes	Not tested
XMC QH64AHIG	No	No	Yes	No

This brings us to the answer to our research question “Is the manufacturer important for extracting the content of different flash chips of modern routers?”: no, the manufacturer is not important. We were able to read all

chips using the Python script and PySerial module since all flash chips use the same protocol for communication.

Our work focused on extracting firmware from devices. This is done for the purpose of finding vulnerabilities in this firmware. Future research could utilize our work to extract firmware from different devices and look for vulnerabilities in them.

Bibliography

- [1] Dangerous prototypes doc website. http://dangerousprototypes.com/docs/Bus_Pirate.
- [2] Flashrom development guidelines page. https://www.flashrom.org/Development_Guidelines.
- [3] Flashrom windows installation page. <https://www.flashrom.org/Windows>.
- [4] Ghidra github page. <https://github.com/NationalSecurityAgency/ghidra>.
- [5] Flashrom website. <https://www.flashrom.org/Flashrom>.
- [6] Flashrom supported hardware page. https://www.flashrom.org/Supported_hardware.
- [7] Putty page. <https://www.chiark.greenend.org.uk/~sgtatham/putty/>.
- [8] Datasheet 25q64jv. <https://datasheetpdf.com/pdf-file/1462312/Winbond/25Q64JVS1Q/1>.
- [9] Ghidra page. <https://ghidra-sre.org/>.
- [10] Datasheet xm25qh64a. https://datasheet.lcsc.com/lcsc/1811072025_XMC-XM25QH64AHIG_C328461.pdf.
- [11] Pyserial documentation page. <https://pyserial.readthedocs.io/en/latest/pyserial.html>.
- [12] Alexander Borg and Carl Aston Francke. Iot pentesting: Obtaining the firmware of a smart lock, 2020.
- [13] F. Chantzis, I. Stais, P. Calderon, E. Deirmentzoglou, and B. Woods. *Practical IoT Hacking: The Definitive Guide to Attacking the Internet of Things*. No Starch Press, 2021.

- [14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 95–110. USENIX Association, 2014.
- [15] Deepa V Jose and A Vijyalakshmi. An overview of security in internet of things. *Procedia Computer Science*, 143:744–748, 2018. 8th International Conference on Advances in Computing & Communications (ICACC-2018).
- [16] Chaira Mahmoud and Sofiane Aouag. Security for internet of things: A state of the art on existing protocols and open research issues. In *Proceedings of the 9th International Conference on Information Systems and Technologies, ICIST 2019, Cairo, Egypt, March 24-26, 2019*, pages 40:1–40:6. ACM, 2019.
- [17] Yang Su and Damith C. Ranasinghe. Leaving your things unattended is no joke! memory bus snooping and open debug interface exploits. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events, PerCom 2022 Workshops, Pisa, Italy, March 21-25, 2022*, pages 643–648. IEEE, 2022.
- [18] Sam L. Thomas, Flavio D. Garcia, and Tom Chothia. Humidify: A tool for hidden functionality detection in firmware. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, volume 10327 of *Lecture Notes in Computer Science*, pages 279–300. Springer, 2017.
- [19] Sebastian Vasile, David F. Oswald, and Tom Chothia. Breaking all the things - A systematic survey of firmware extraction techniques for iot devices. In Begül Bilgin and Jean-Bernard Fischer, editors, *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers*, volume 11389 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2018.
- [20] Arno Wagner. Python scripts for the buspirate. https://www.tansi.org/bus_pirate/index.html.
- [21] Zhi-Kai Zhang, Michael Cheng Yi Cho, Chia-Wei Wang, Chia-Wei Hsu, Chong Kuan Chen, and Shiuhyng Shieh. Iot security: Ongoing challenges and research opportunities. In *7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014*, pages 230–234. IEEE Computer Society, 2014.

Appendix A

Python script read_image.py

```
#!/usr/bin/python3.1

# scans for BusPirate port, and reads 4Mbit flash image
# NOTE: Will overwrite target file without warning!

# Version 1.0 11.12.2012

# Copyright (C) 2012, Arno Wagner <arno@wagner.name>
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# version 2, or a later version at your choice, as published by the
# Free Software Foundation.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301

import serial, time, sys

dumpfile='flash_data.img'

candidates = ['/dev/ttyUSB0', '/dev/ttyUSB1', '/dev/ttyUSB2', '/dev/ttyUSB3',
              '/dev/ttyUSB4', '/dev/ttyUSB5', '/dev/ttyUSB6', '/dev/ttyUSB7',
              '/dev/ttyUSB8', '/dev/ttyUSB9', '/dev/ttyUSB10', '/dev/ttyUSB11',
              '/dev/ttyUSB12', '/dev/ttyUSB13', '/dev/ttyUSB14', '/dev/ttyUSB15']
```

```

def test_if(name):
    global wait_time
    try:
        s = serial.Serial(port=name, baudrate=115200, timeout=0.01)
    except Exception as e:
        return None
    else:
        return s

def test_bp(s):
    if s == None:
        return None
    else:
        s.write(b'#\n')
        i = s.read(10000)
        sl = i.splitlines()

        if len(sl) < 4 or \
            ( sl[2][0:10] != b'Bus Pirate' and \
              sl[3][0:10] != b'Bus Pirate' ):
            return None;
        return True

def find_bp():
    # tests list of candidates. Returns serial instance if exactly one found.
    # If several found, aborts with exception.
    # If none found, returns None
    cnt = 0
    s_bp = None
    n_bp = None
    n_str = ''
    for n in candidates:
        s = test_if(n)
        is_bp = test_bp(s)
        if is_bp:
            cnt += 1
            s_bp = s
            n_bp = n
            n_str += n + '\n'
    if cnt > 1:
        raise IOError('More than one Bus Pirate found! Interfaces: \n'+n_str)
    return (s_bp, n_bp)

def adr24(n):
    # takes an int and returns a 24 bit hex address MSB,...,LSB

```

```

# e.g. 0x01 0x00 0x23
b1 = n//65535
b2 = (n//256)%256
b3 = n%256
s = '0x{:02X} '.format(b1) + '0x{:02X} '.format(b2) + '0x{:02X}'.format(b3)
print('adr24: ',n,' -> ', s)
return(s.encode('ascii'))

def safe_exit(msg):
    # exit with BP reset
    print(' ** ERROR -- safe_exit() called **')
    print(' MSG: ',msg)
    s.write(b'#\n')
    i = s.read(100)
    # sl = i.decode('ascii').splitlines()
    # print(sl)
    # Note: seems reset from SPI does sometimes not echo the #...
    # if sl[0] != 'RESET' and sl[1] != 'RESET':
    #     print(' ** error exit reset failed! **')
    sys.exit(-1)

a = find_bp()
(s,n) = a
#print('device:      ', n)
#print('seri_inst: ', s)

# Start interaction
s = serial.Serial(port=n, baudrate=115200, timeout=0.01)

# reset just in case
s.write(b'#\n')
i = s.read(200)
#sl = i.decode('ascii').splitlines()
#print(sl)
#if sl[1] != 'RESET': safe_exit('initial reset failed')

# m set SPI
s.write(b'm\n') # 'm'
i = s.read(100)
sl = i.decode('ascii').splitlines()
#print(sl)

# SPI
s.write(b'5\n') # Menu item 5 = SPI
i = s.read(100)
sl = i.decode('ascii').splitlines()
#print(sl)

```

```

s.write(b'4\n') # 1=30khz, 2=125kHz, 3=250kHz, 4=1MHz
i = s.read(100)
sl = i.decode('ascii').splitlines()
#print(sl)

s.write(b'1\n') # 1= clock idle low (default) 2: clock idle high
i = s.read(100)
sl = i.decode('ascii').splitlines()
#print(sl)

s.write(b'2\n') # 1:idle to active 2:active to idle (default)
# note: for the 25x40 the datasheet seems to indicate 1 is
# correct, but 2 works also. But for a a25l040, 1 fails while
# 2 works. So likely 2 is really the usually correct choice.
i = s.read(100)
sl = i.decode('ascii').splitlines()
#print(sl)

s.write(b'1\n') # sample signal 1: middle 2:end
i = s.read(100)
sl = i.decode('ascii').splitlines()
#print(sl)

s.write(b'2\n') # 1:CS 2:/CS
i = s.read(100)
sl = i.decode('ascii').splitlines()
#print(sl)

s.write(b'2\n') # 1: H=Hi-Z 2: H=3v3 Note: Pullups are problematic
i = s.read(100)
sl = i.decode('ascii').splitlines()
#print(sl)
if sl[1] != 'Ready': safe_exit('SPI detail setting failed')

s.write(b'W\n') # Activate power
i = s.read(100)
sl = i.decode('ascii').splitlines()
#print(sl)
if sl[1] != 'Power supplies ON' and \
    sl[1] != 'POWER SUPPLIES ON':
    safe_exit('power ON failed')

# give is a bit of time to come up and settle
time.sleep(0.200)

# read data
outf = open(dumpfile, 'wb')

```

```

read_size = 128

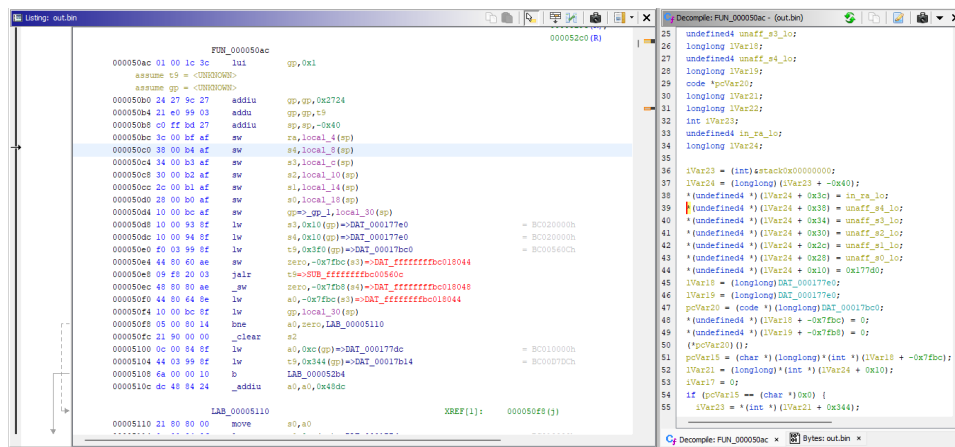
for adr in range(0, 0x80000, read_size):
    address = adr24(adr)
    cmd = b'[0x03 ' + address + b' r:' + str(read_size).encode('ascii') + b']\n'
    print('cmd: ', cmd)
    s.write(cmd) #
    i = s.read(2000)
    sl = i.decode('ascii').splitlines()
    #print(sl)
    r = sl[6]
    if r[0:6] != 'READ: ': safe_exit('read failed')

    ds = r[6:]
    # print('ds: ', ds)
    dss = ds.split()
    # for n in dss: print(int(n, 16), end=' ')
    # print()
    print('len: ', len(dss))
    for n in dss:
        c = int(n, 16)
        b = bytes([c])
        outf.write(b)
outf.close()

# Final reset
s.write(b'#\n')
i = s.read(200)
#sl = i.decode('ascii').splitlines()
#print(sl)
# Note: Seems reset from SPI does sometimes not echo the '#'
#if sl[0] != 'RESET' and sl[1] != 'RESET': safe_exit('final reset failed')

```


Ghidra's decompile function



In the left window, all memory addresses including the values stored at these addresses are shown. In the right window is Ghidra's decompile function, turning the low-level information stored in memory into high-level C code.

Appendix C

Winbond 25Q64JVS1Q specification

```
{
    .vendor          = "Winbond",
    .name            = "W25Q64JV-.M",
    .bustype         = BUS_SPI,
    .manufacture_id = WINBOND_NEX_ID,
    .model_id        = WINBOND_NEX_W25Q64JV,
    .total_size      = 8192,
    .page_size       = 256,
    /* supports SFDP */
    /* QPI enable 0x38 */
    .feature_bits    = FEATURE_WRSR_WREN | FEATURE_OTP | FEATURE_QPI,
    .tested          = TEST_OK_PREW,
    .probe           = PROBE_SPI_RDID,
    .probe_timing    = TIMING_ZERO,
    .block_erasers   =
    {
        {
            .eraseblocks = { {4 * 1024, 2048} },
            .block_erase = spi_block_erase_20,
        }, {
            .eraseblocks = { {32 * 1024, 256} },
            .block_erase = spi_block_erase_52,
        }, {
            .eraseblocks = { {64 * 1024, 128} },
            .block_erase = spi_block_erase_d8,
        }, {
            .eraseblocks = { {8 * 1024 * 1024, 1} },
            .block_erase = spi_block_erase_60,
        }, {
            .eraseblocks = { {8 * 1024 * 1024, 1} },
            .block_erase = spi_block_erase_c7,
        }
    }
}
```

```

    },
    .printlock = spi_prettyprint_status_register_bp2_tb_bpl,
    .unlock    = spi_disable_blockprotect_bp2_srwd,
    .write     = SPI_CHIP_WRITE256,
    .read      = SPI_CHIP_READ,
    .voltage   = {2700, 3600},
},

```