Backing Up Firmware from Dallas Semiconductor DS5002FP

Peter Wilhelmsen Morten Shearman Kirkegaard

2017-07-16

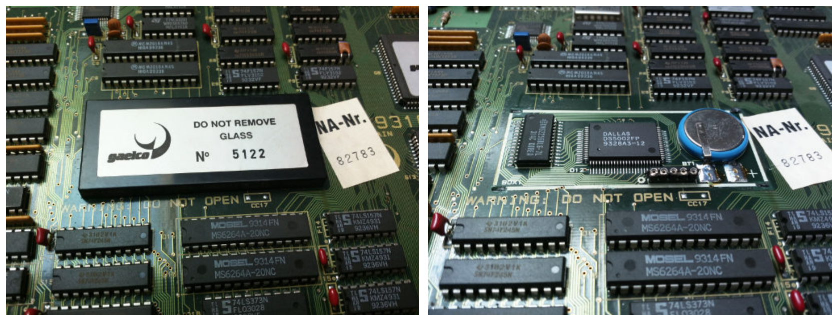
Abstract

Secure embedded systems (e.g. arcade games) may use a Dallas Semiconductor DS5002FP for a protected subsystem. It is a "secure" reimplementation of the Intel 8051. The firmware is kept encrypted in SRAM, powered by a 3V lithium cell. It is possible to take control of the DS5002FP, and recover the plaintext version of the firmware. In order to do that, we connect an FPGA to the address bus and the data bus. We route the chip-select of the RAM through the FPGA. Each time the DS5002FP CPU requests data from the RAM, the FPGA can either respond, or chip-select the original RAM. We inject code to read the original firmware, and output it on a port. To some extend, this is a reimplementation of Dr. Markus G. Kuhn's attack[1] from 1998.

1 Hardware

1.1 Getting Physical Access

On some systems, the DS5002FP, its RAM, and battery, are inside a small plastic cap. We remove the plastic cap, in order to get physical access to all the parts.

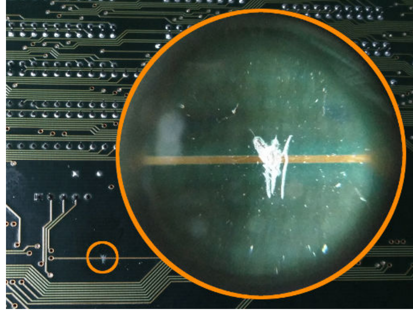


Under the cover is the RAM on the left, the DS5002FP in the middle, and the 3V lithium cell on the right. In front is a UART connector, which can be used to push new firmware to the system. That will generate a new key, so it is not useful in this attack.

[1] Markus G. Kuhn. "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP." IEEE Transactions on Computers, Vol. 47, No. 10, October 1998.

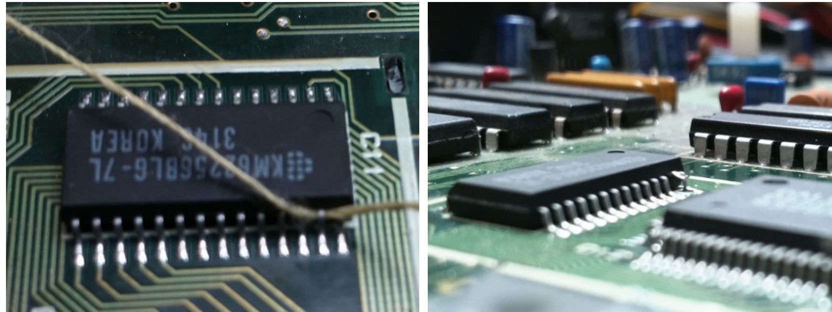
1.2 Enabling Reset

Some systems have the RST pin of the DS5002FP pulled low via an inverter IC (7404). In order for us to control when the CPU is reset and when it is not, we cut this trace.



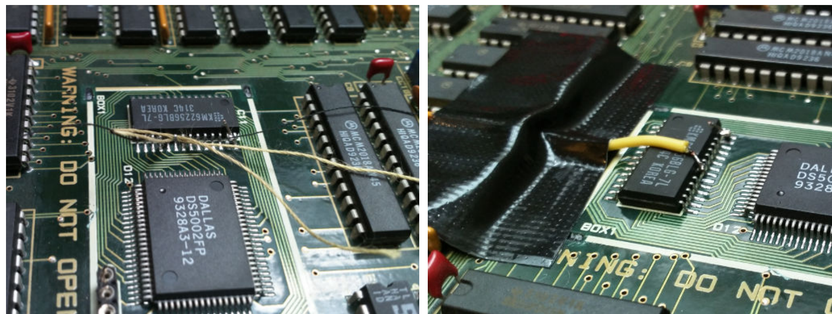
1.3 Disabling RAM Writes

In order to make sure that we don't overwrite the firmware, we lift the RAM's $\overline{\mathtt{WE}}$ (pin 27). The system must obviously be off while doing this. We use very thin sewing thread, and use that to pull Kevlar thread through the pin spacing. This is done with a steady hand, and no tools, in order to not connect any other pins, at any point. We then use a gas powered soldering iron to heat the pin, while pulling the thread out below it. A regular soldering iron would be ground connected, and could cause the RAM to lose all its data. Using a gas soldering iron solves this issue. This will disconnect the pin, and allow us to lift it up, and connect it directly to the RAM's $\mathtt{V}_{\mathtt{CC}}$ (pin 28).



1.4 Rerouting RAM Chip-Select

The attack requires some of the CPU's memory requests to be served by the FPGA, and other requests to be served by the RAM. We make the setup so that the FPGA decides who responds to a given request. We do this by lifting the RAM's $\overline{\texttt{CS}}$ (pin 20), and connecting it to an output pin of the FPGA. As soon as we desolder that pin, the RAM might function as if it was chip-selected. This can empty a lithium cell very fast. In order to make sure that the RAM still has power to keep the content, we solder on wires to a power supply, before desoldering the $\overline{\texttt{CS}}$. The lithium cell is not supposed to be parallel with a power supply, but as long as the voltage is very close, it is not a problem.

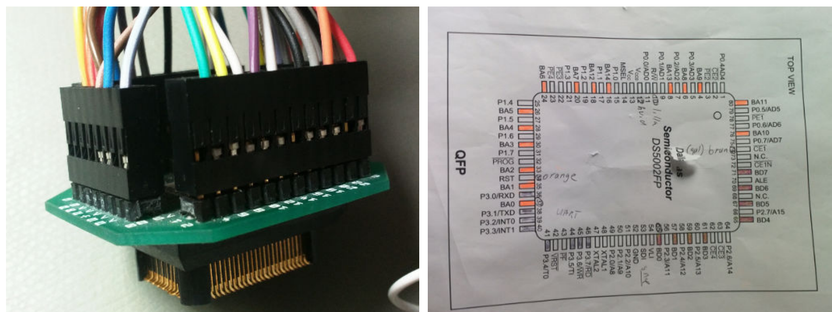


1.5 Connecting the CPU to the FPGA

Using a clip with 80 pins, we are able to access all CPU pins, without soldering. The interesting pins are `GND`, `BA` (the RAM address), `BD` (the RAM data), $\overline{\texttt{CE1}}$ (the RAM chip-enable), `P3` (for output), `R/`$\overline{\texttt{W}}$, and `RST`.

The FPGA is on a board with 74LVXC3245 transceivers, which allows us to connect a relatively modern FPGA to a 5V target.

We listen on the `R/`$\overline{\texttt{W}}$ pin, to make sure that the FPGA only replies when the CPU is making a read, not a write. On top of this, we use 270Ω resistors between the FPGA output and the RAM data bus. This is to make sure that we do not destroy the FPGA, CPU, or RAM, if we misinterpret who is to output—just to be sure.



2 Software

2.1 Firmware for the FPGA

We have written a relatively simple firmware in VHDL. In total it is 565 lines of source code[2]. Since debugging VHDL is not as easy as debugging regular software, we want to keep it as simple as possible.

The code is a simple state machine, which receives a task from the PC via UART, executes it against the DS5002FP, and writes the result back to the PC.

We use a tag, so we can make sure that the result is from the task we sent. Each element in the task is either a byte in hexadecimal, which the FPGA writes on the data bus, or `XX` when the FPGA should chip-select the real RAM.

For example:

```
QWHS XX XX XX 41 XX
```

This will select the real RAM for the first three read accesses, then have the FPGA reply `41` for the fourth access, and select the real RAM again for the fifth memory access. The response begins with the tag, and then for each memory access, it gives us the RAM address, RAM data, and P3 data:

```
QWHS 1399:E1:FF 1C84:1B:FF 7EBA:67:FF 6963:41:FF 6BE4:E6:FF
```

2.2 DS5002FP Simulator

In order to be able to develop the attack without physical access to a target board, we developed a simple FPGA and DS5002FP simulator. It only implements the relevant instructions, but that was enough to implement the attack. It is meant to be used as a network service. The simulator is 288 lines of source code[3].

3 The Real Attack

As attackers, we control the data returned to the CPU, every time it requests RAM data. This is ciphertext, and we don't know the key, so we can not just control the instructions executed. On top of that, some instructions take one or more clock cycles after the code is fetched from RAM. The DS5002FP uses that to access RAM data, which is not related to the instruction, in order to make it more difficult to figure out. Another protection is the fact that the interrupt vector is kept in internal RAM—supposedly to make sure that attackers can't tell when the CPU gets interrupts.

[2] `http://www.afdelingp.dk/files/articles/ds5002fp/top.vhd`
[3] `http://www.afdelingp.dk/files/articles/ds5002fp/sim.c`

When we power up the CPU and pull down `RST`, it is going to start executing. We keep doing this, once for each 256 possible data values for the first RAM access. If the CPU requests the same RAM address for the second cycle, no matter which value we give it in the first cycle, that RAM access was just a dummy read. We move on, and try all combinations for second RAM access, and so on, until we control an instruction.

When the DS5002FP is reset, the `P3` outputs all 1-bits (`FF`). We brute-force the next two RAM accesses, until we find the ciphertext for the `INC P3` instruction. This will output `00` on `P3`. That instruction is encoded as `05 B0`.

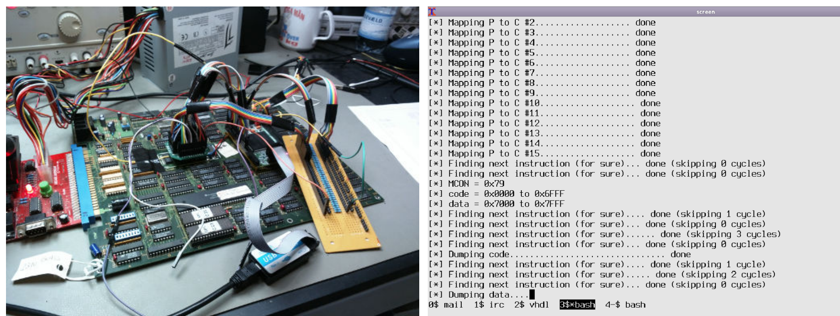
In order to make a full map of the third byte, we search for the ciphertext in the first byte, which deciphers to `MOV iram addr, #data` which is `75`. This will allow us to send the 256 combinations of `75 B0 XX`, mapping between ciphertext and plaintext on the third byte.

We can then brute-force the first byte (where we had `75`), so we find a `NOP`-like instruction, and the second byte (where we had `B0`) so we get `75` there. Since we have created a map for the third byte, we can look up `B0`. This will give us a `NOP`, followed by `75 B0 XX`. We can then create a map for the fourth byte.

At this point, we have two ciphertext/plaintext maps (third and fourth memory accesses). We can brute-force yet another `NOP`-like instruction, this time for the second byte. Once we have that, we can use the existing maps to execute `75 B0 XX` with the `XX` being on the fifth memory access. Once that is done, we can prepend a `NOP`, move the `MOV iram addr, #data` one byte, and make yet another ciphertext/plaintext map. We can keep doing that, until we have enough maps to encipher the code to dump the memory.

The code we use to find the boundary between the code and the data part is:

```
E5 C6      ; MOV A, MCON
F5 B0      ; MOV P3, A
```



The code we use to dump the firmware's code part is:

```
90 13 37  ; MOV DPTR, 0x1337
74 00     ; MOV A, 0x00
93        ; MOVC A, @(A+DPTR)
F5 B0     ; MOV P3, A
```

The code we use to dump the firmware's data part is:

```
90 73 31  ; MOV DPTR, 0x7331
E0        ; MOVX A, @DPTR
F5 B0     ; MOV P3, A
```

We keep resetting the DS5002FP, and executing the code with different values for `DPTR`, thereby dumping all the firmware. Bruting the first instructions takes around two minutes. Dumping 32768 bytes of firmware takes around four minutes.

The source code[4] for the PC side of the attack is 886 lines.

This should allow you to create backups of any DS5002FP firmware, as long as you have physical access.

We would like to thank David Haywood for working with us, and contributing to our understanding of this microprocessor.

Revision: 1.1

[4] `http://www.afdelingp.dk/files/articles/ds5002fp/attack.c`