

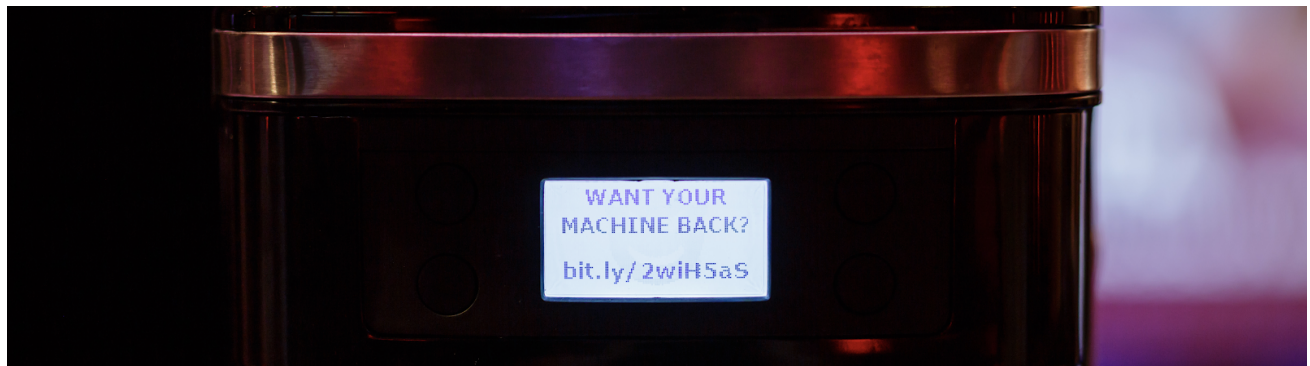
The Fresh Smell of ransomed coffee

 decoded.avast.io/martinhrn/the-fresh-smell-of-ransomed-coffee/

September 25, 2020



by [Martin Hron](#) September 25, 2020 28 min read



We turned a coffee maker into a dangerous machine asking for ransom by modifying the maker's firmware. While we could, could someone else do it too? As you might expect, the answer is: Yes. Follow us on a journey where we show you that firmware is the new software.

Firmware is the new software

Some research is so fun that it confirms why I do this work. I was asked to prove a myth, call it a suspicion, that the threat to IoT devices is not just to access them via a weak router or exposure to the internet, but that an IoT device itself is vulnerable and can be easily owned without owning the network or the router. I also bet that I could make that threat persist and present a true danger to any user. We often say that your home network, thought of as a chain of trust, is only as strong as its weakest link, but what if the same were true at the device level? What would that mean?

Let's say you have an IoT device that is well protected with functions that can be accessed through a well-defined API; even if you can control the device through the API, you probably can't do too much harm. Firmware, the programming inside the device has logical constraints that don't allow you, for example, to close garage doors while someone is in the way of them or overheat a device so that it combusts.

We used to trust that hardware, such as a common kitchen appliance, could be trusted and could not be easily altered without physically dismounting the device. But with today's "smart" appliances, this is no longer the case.

My colleagues often hear me say that "firmware is a new software." And that software is very often flawed. We see it everywhere. CPU flaws, and cryptographic chips generating weak keys that can be easily broken. The weakened state of IoT security is due in large part to the fact that, nowadays, it is more convenient and cheap to place a processor inside a device which controls and orchestrates all hardware parts, motors, sensors, heating elements, etc. based on a short program called firmware. This solution is not only cheap, but has also one important property – it can be updated

Back in the day, if there was a so-called design flaw in a piece of hardware, the rigid design (often hardwired) meant the vendor would need to replace the whole component or logic board or even replace the entire device. Manufacturers would have to change the manufacturing process and at potentially great financial losses. In the era of firmware, this can be easily mitigated just by issuing a firmware update.

The process of updating firmware can vary greatly, from connecting to the special device using a special tool (which still requires the vendor's physical interaction) to the more and more popular way of OTA (over the air) updates. In this case, a vendor doesn't have to be physically present and the whole process is done either automatically over the internet or semi-automatically after a user's notification and approval of the update.

Making your first geeky coffee

So let's see what we have. We have a coffee maker that allows you to make coffee the old fashioned way by pressing a few buttons or via a mobile phone or tablet using an app. The maker operates with Wi-Fi and when unboxed you have to connect it to your network through a companion app on your mobile phone. When turned on for the first time, the coffee maker works in a local mode and it creates its own Wi-Fi network that the hopeful coffee drinker first connects to in order to set up the device.

It's worth to mention that this coffee maker is no longer supported by the vendor as they moved to a more secure platform in 2017 and technology because of all the flaws illustrated in this research. But this also shows a general problem we have with abandoned IoT devices.

When we downloaded the companion app, we saw that it allows you to create a network of any devices of this particular vendor and connects these devices to the home network and then allows you to control all the functions of your coffee maker or smart kettle. It also allows you to check the firmware version of the device and update it if needed.

The protocol that this device speaks has already been documented on the internet by [several other researchers](#). As expected, it's a simple binary protocol with hardly any encryption, authorization or authentication. Communication with machines takes place on `TCP` port `2081`. The format of the command is very simple:



In response (if there is a response) the coffee maker sends back:



`response_type` differs based on the command, but the general rule is: If the response contains data `response_type = command + 1`, if it's just a status then `response_type=3` and then there is only one `data` byte which contains resulting status, where `1` means success. The complete list of commands is in the [GitHub repository](#).

So just for illustration, by issuing this command:



“default” settings

You'll make yourself a nice cup/cups of coffee based on the default settings of the coffee maker. As you can see, there is no security, so anyone who has access to the network and is able to reach the IP address of the coffee maker can control it. What is more interesting, is that all these commands are also available through that default opened Wi-Fi network when the device is not joined to a home network yet.

Update the future

Let's return to our main goal of hijacking the coffee maker for nefarious purposes. How secure is the updating process? Can we break into it? Can we even change the firmware to do something else than was originally intended? Can we turn the device into a physically dangerous device? Can we do it remotely? As I said in the beginning, the weakest link always compromises the whole system. Either it's a network or a device. The goal is answering all the above questions and prove that IoT devices could be also compromised at the firmware level.

To start, we wanted to learn how the update process works. We have several options to do this, but we already have a recipe for this based on similar past research we have done. The general rule is to make it as simple as possible; here is our to-do list when trying to reverse the update process of a piece of firmware.

1. Get it (the file with the firmware)
2. Unpack it (if it is packed/encrypted)
3. Reverse engineer it (translate those zeroes and ones into meaningful code)
4. Modify it (add the malicious content)
5. Upload it (and push it back to the device)

It's simple right?

Finding the firmware

First, we need to get the firmware somehow, and again there are several options. Let's stick to the rule, the simplest option first:

1. Google it – as obvious as it seems, sometimes security researchers omit this step which could save them a lot of work. Firmware is often available on the internet to be downloaded.
2. Capture and analyze network traffic – if the protocol is unencrypted the easiest way is to just capture the network traffic, in that case, we have three more options where to look
 - traffic between the device and the internet
 - traffic between device and companion app, if there is any
 - traffic between the companion app and the internet
3. Analyze and reverse engineer the companion app. Before we dive deeply into the device, it's advisable to first peek into the device's companion app, it's usually the easiest solution for grabbing the communication protocol and commands. No luck with any of the above? It's time to dismount the device, trace the board to identify all the components, get datasheets to find debugging ports, and possibly dump the firmware directly from the chip. This is really an adventure in hardware and not for everyone.

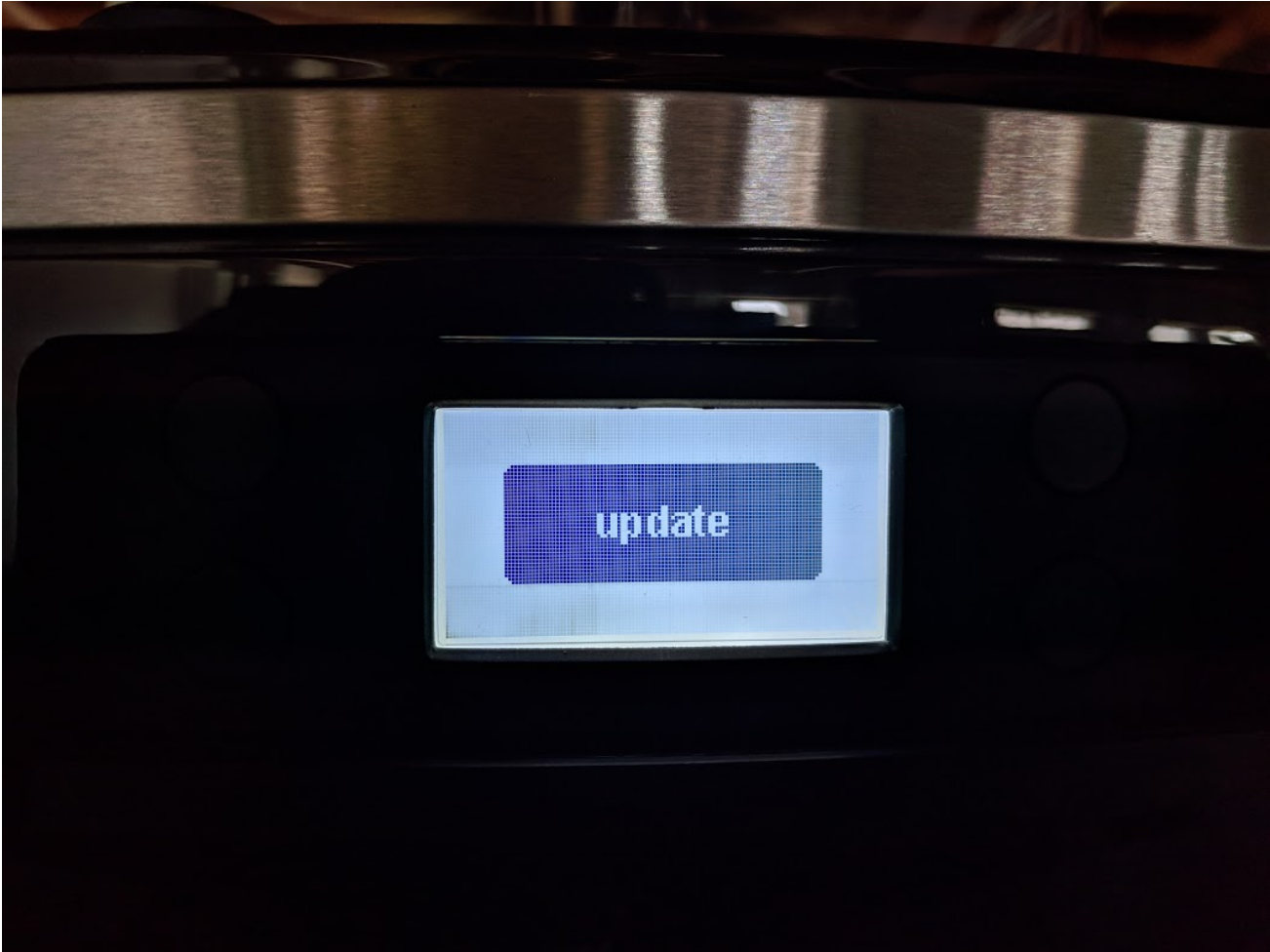
Life is not so simple, and neither is reverse engineering IoT devices. It turned out that we had to use a combination of all the aforementioned techniques. First, we googled it, and as the commands have already been documented, we found a command that says `"update the firmware."` But the command itself (its parameters and format) had not been documented. So by issuing:

0x6D	terminator 0x7E
------	-----------------

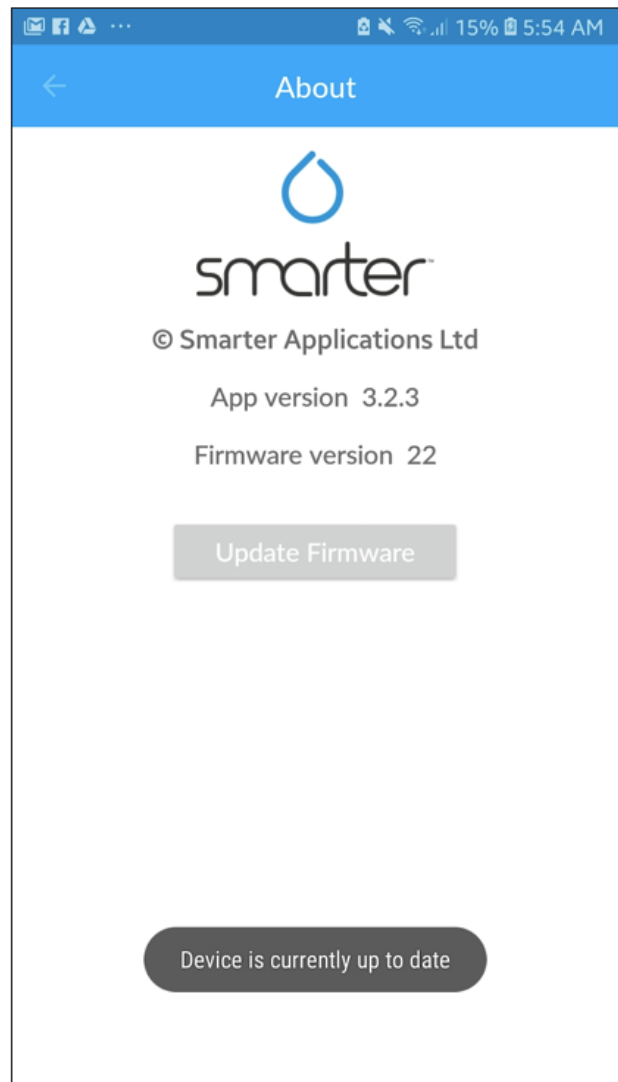
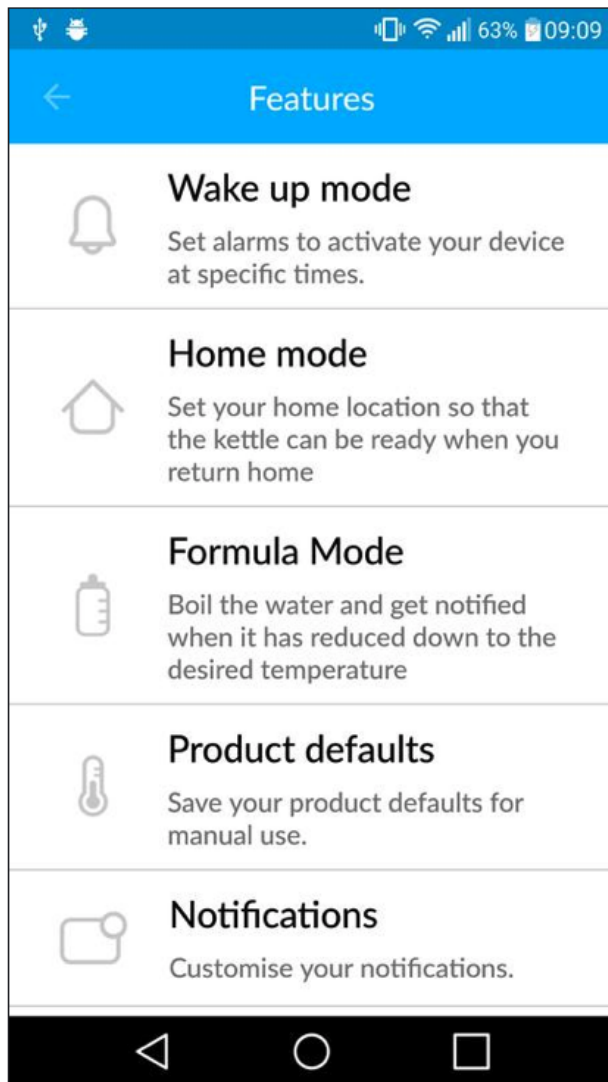
this simple (two bytes) command

switches coffee maker into update mode

the coffee maker goes into an update mode (on the newer firmware you have to push a button to actually start the update, but this is not the case with older versions !)



Hmm nice, but nothing happens. By analyzing network traffic, we concluded that there is nothing to analyze, as there is no traffic coming out of the coffee maker at this stage. So we tried the Android companion app.



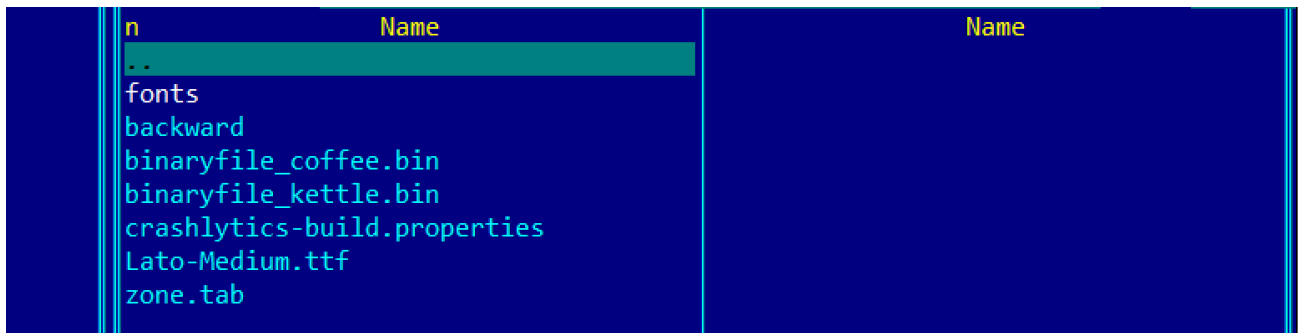
Android application for smart coffee maker, right picture shows an update screen
 You can see the update button is grayed out, so obviously, the firmware is up-to-date and there is no easy option to push the firmware update to be able to see what's in the network traffic. What's interesting here is what's missing. There was no communication to the internet neither from the coffee maker nor from the app. So how is it possible that the app knows that the coffee maker has the latest firmware? The only packets of data that went through were those between the machine and app when the app had been asking the machine for the version of the firmware.



Typical traffic analysis of IoT with companion app: we are interested in traffic between app

and internet and app and device.

This is strange, and it seemed to tell us that the firmware is probably not on the internet and must be part of the app. So we opened the .apk file as easy as a .zip file. What we found there, proved our assumption.



n	Name	Name
-	-	
	fonts	
	backward	
	binaryfile_coffee.bin	
	binaryfile_kettle.bin	
	crashlytics-build.properties	
	Lato-Medium.ttf	
	zone.tab	

Files inside the apk: You can see firmware for both products are contained in the two files with suffix .bin

The firmware is part of the Android app and it also means that new versions of the firmware always come with new versions of the app. This makes perfect sense if you think about it for a second. The new firmware usually adds new functionality, which has to be reflected somehow inside the user interface of the app, and it allows us to find a file containing the firmware without even touching the device. That's nice and not very common.

Reverse engineering

In the next step of my research we try to figure out what the file contains. The first thing any reverse engineer would do would be to just eyeball the file and see what it contains. What we saw there was a bunch of strings that actually made sense. From this, we could deduce there is no encryption and the firmware is probably a “plaintext” image that is uploaded directly into the **FLASH** memory of the coffee maker.

```

4C 45 0D 0A 00 00 00 00 AGE HANDLE
20 31 20 49 6E 76 61 6C Schedule 1 Inval
64 75 6C 65 20 32 20 49 id: Schedule 2 I
53 63 68 65 64 75 6C 65 nvalid: Schedule
69 64 3A 00 53 63 68 65 3 Invalid: Sche
6E 76 61 6C 69 64 3A 00 dule 4 Invalid:
65 66 61 75 6C 74 73 3A Coffee defaults:
72 20 62 75 66 66 65 72 clear buffer
49 50 53 54 4F 3D 31 38 AT+CIPSTO=18
52 20 42 55 46 46 45 52 0 CLEAR BUFFER
45 53 54 4F 52 45 0D 0A AT+RESTORE
57 4A 41 50 5F 44 45 46 AT+CWJAP_DEF
57 4A 41 50 5F 44 45 46 ? AT+CWJAP_DEF
49 50 4D 55 58 3D 31 0D =" AT+CIPMUX=1
46 50 4F 57 45 52 3D 35 AT+RFPOWER=5
20 43 4F 4E 4E 45 43 54 0 WIFI CONNECT
4F 52 59 20 52 45 53 45 ED FACTORY RESE
4B 5F 43 41 52 41 46 45 T CHECK_CARAFE
64 75 6C 65 20 49 6E 66 Schedule Inf
20 43 4F 4E 4E 45 43 54 o: WIFI CONNECT

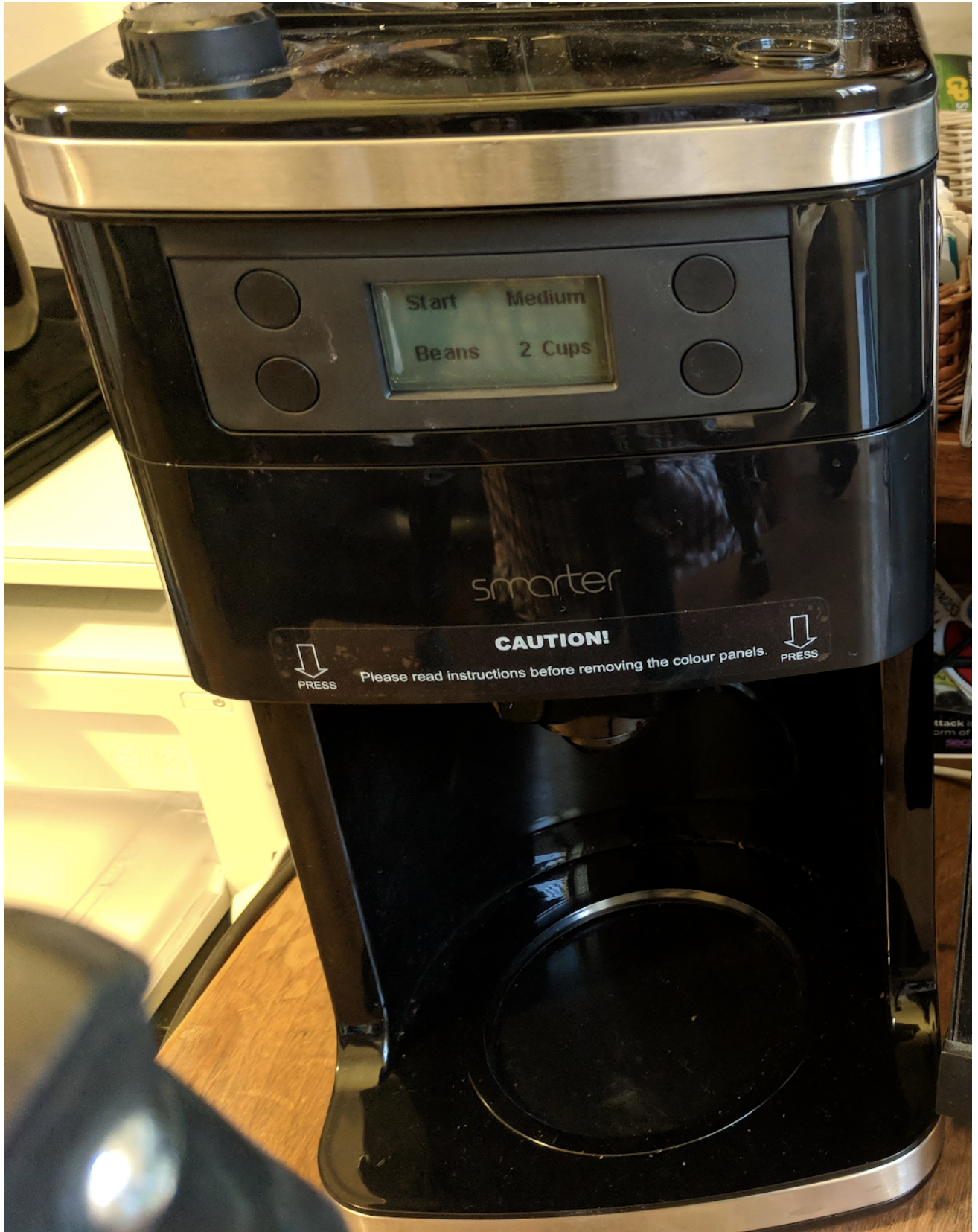
```

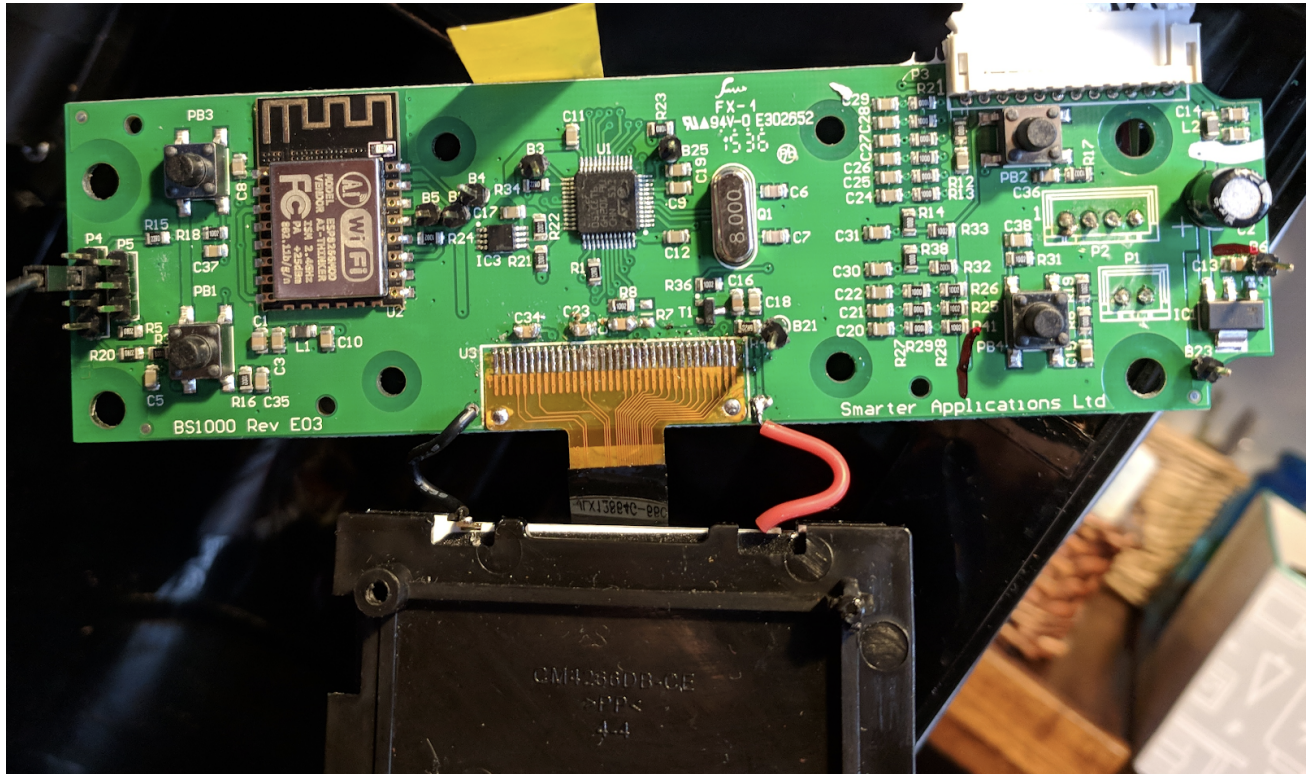
Content of the file tells us that there is no encryption involved and also that there is a probably extra WiFi modem module

To be able to understand what the firmware does we had to decode the binary into an assembly, that means we needed to know the processor or architecture this firmware is targeting.

We can guess or we can experiment. Or we could dismount the device to get information about the hardware the old-fashioned way. So after unscrewing a few screws and taking off a few plastic covers, we finally got our hands on a circuit board. This is what it looks like:

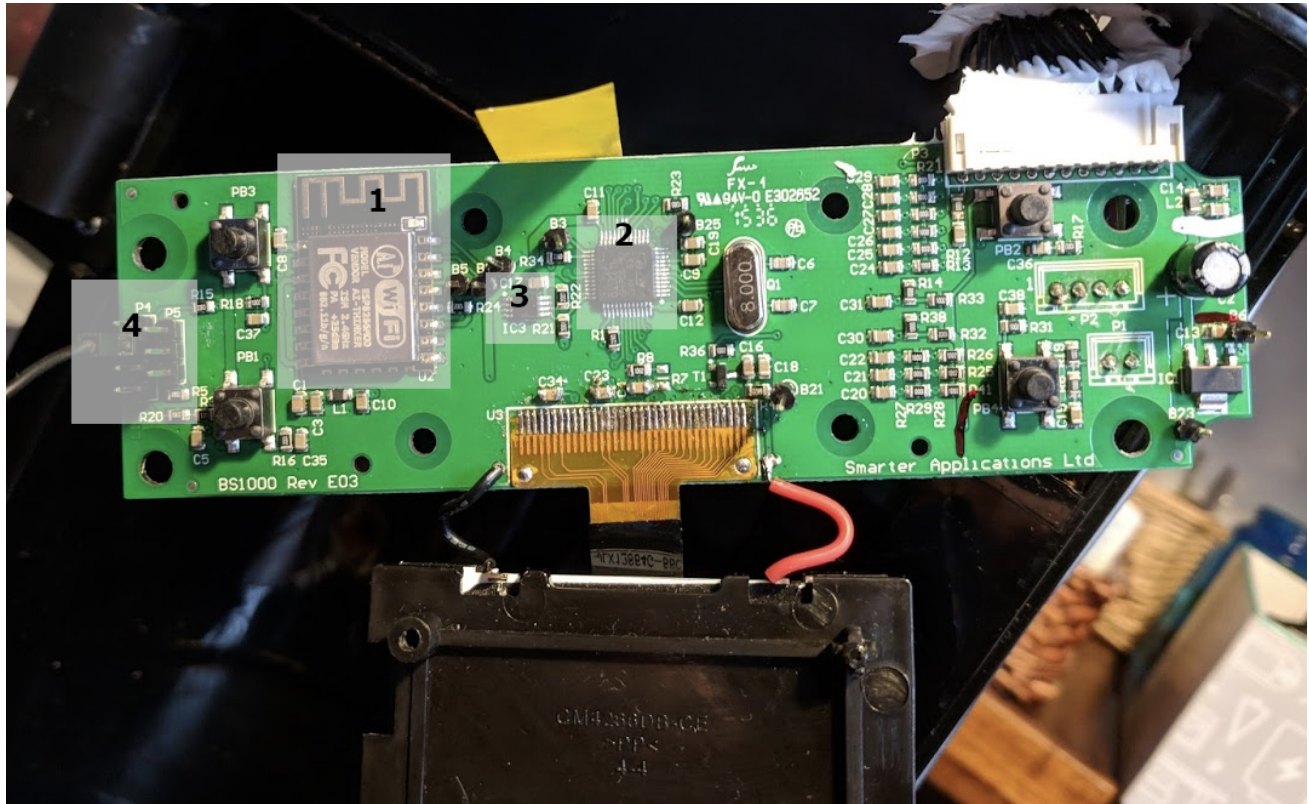






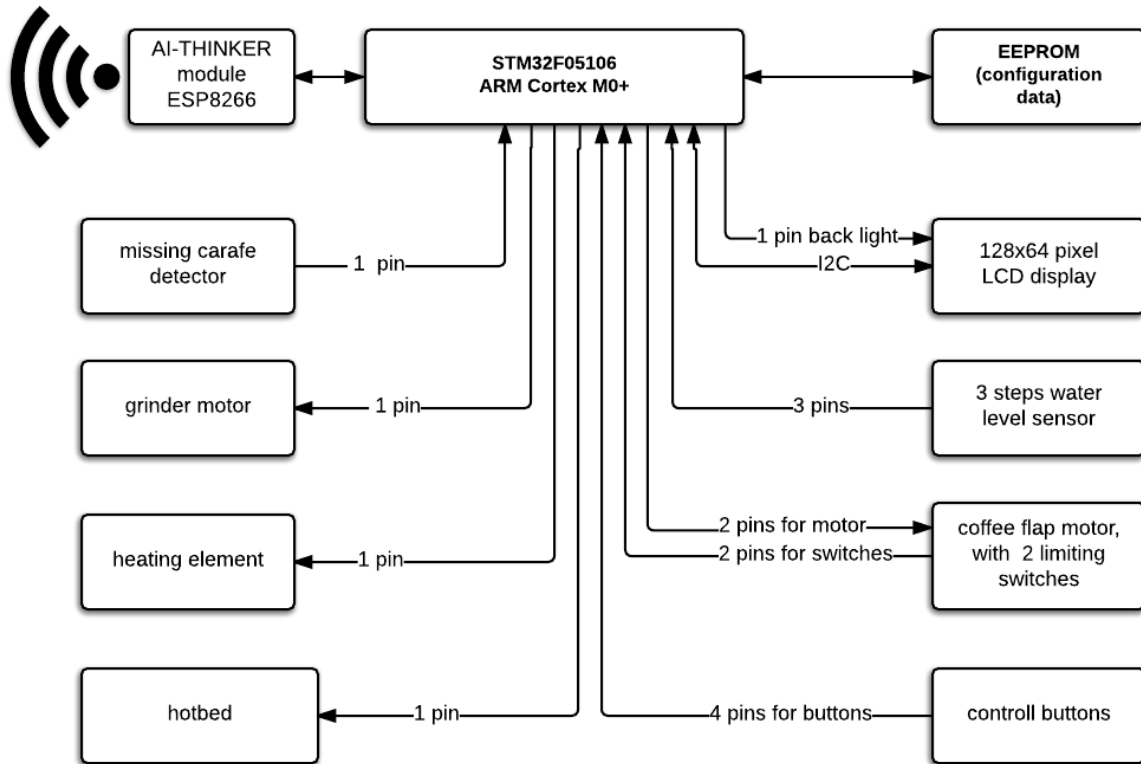
coffee maker and its control board

We can immediately see, there is one component we are already very familiar with. The **ESP8266** module. This is a very common chip that provides Wi-Fi capabilities and is widely used in many devices by manufacturers as a Wi-Fi modem or even as the main CPU of the device. That's not the case this time. We identified other chips that seemed important and came to a different conclusion.



1 – ESP8266 with AT modem firmware, 2 – STM32F05106 ARM Cortex M0 – main CPU that glues everything together, 3 – I2C EEPROM with configuration, 4 – debug ports and programming interface

The main CPU in this case, and the real heart of the whole coffee maker, is an ARM Cortex M0 processor that controls all the connected gadgets (coffee grinder, coffee flap, water level sensor, hotbed heater, main heater, display, buttons and “missing carafe” sensor). To make it easier to understand, we came up with the block diagram of the whole maker:



A block diagram of coffee machine's main components

We can see that the **ESP8266** is only used as a WiFi modem that provides WiFi capabilities to the main CPU (this could sound funny as the **ESP8266** is many times more powerful and has a much more memory than the main CPU, but it's true). Anyway, we have what we came for. Now that we know the exact type of CPU, we can download the data sheet and extract some basic information that will help us figure out where to begin with disassembly.

Quick View

The STM32F051xx microcontrollers incorporate the high-performance ARM®Cortex®-M0 32-bit RISC core operating at up to 48 MHz frequency, high-speed embedded memories (up to 64 Kbytes of Flash memory and 8 Kbytes of SRAM), and an extensive range of enhanced peripherals and I/Os. All devices offer standard [Show more +](#)

Key Features

- Core: ARM®32-bit Cortex®-M0 CPU, frequency up to 48 MHz
- Memories
 - 16 to 64 Kbytes of Flash memory
 - 8 Kbytes of SRAM with HW parity checking
- CRC calculation unit
- Reset and power management
 - Digital and I/O supply: $V_{DD} = 2.0\text{ V to }3.6\text{ V}$
 - Analog supply: $V_{DDA} = \text{from } V_{DD} \text{ to } 3.6\text{ V}$
 - Power-on/Power down reset (POR/PDR)
 - Programmable voltage detector (PVD)

[Show more +](#)

 [Download Datasheet](#)

[Resources](#) 

[Tools & Software](#) 

[Sample & Buy](#) 

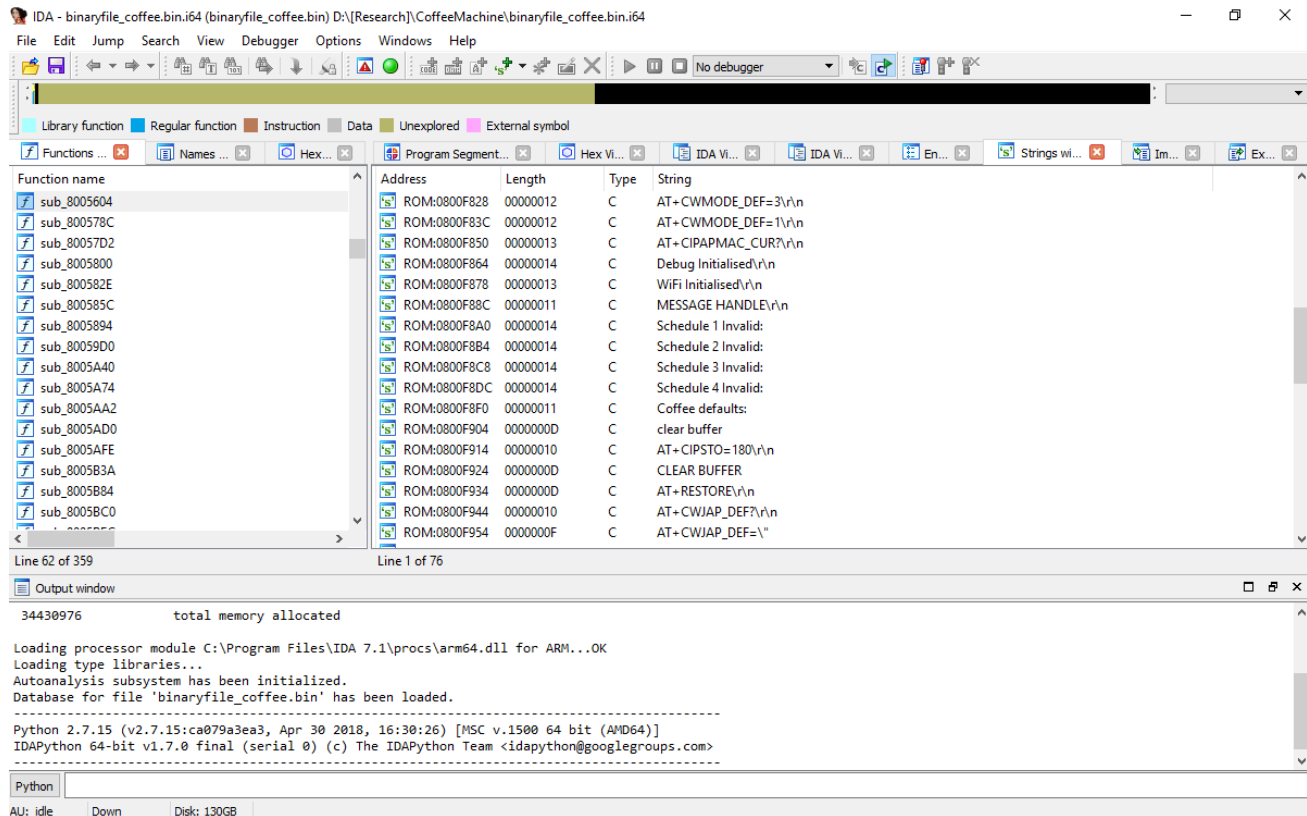
[Quality & Reliability](#) 

Product Image

System	48 MHz ARM Cortex-M0 CPU	32-Kbyte Flash memory	Analog
Power supply 1.8 V internal regulator POR/PDR/PVD		8-Kbyte SRAM HW parity checking	
Xtal oscillators 32 kHz + 4-32 MHz	Nested Vector Interrupt Controller (NVIC)	20-byte backup data	1x 12-bit ADC 15 channels / 3 MAPS 2x analog comparator Temperature sensor
Internal RC oscillators 40 kHz + 8 MHz			
Internal RC oscillator 48 MHz (auto trimming on exit syndrome)	1x SPI (with PS mode)	1x 16-bit motor control PWM Synchronized RC timer	
PLL			1x I2C with FastMode Plus
Clock control	2x USART with modem control (1x with LIN peripheral) I2C		
Calendar RTC			APB bus
SysTick timer	3-channel DMA		
2x watchdogs (independent and master)			Touch-sensing Up to 16 keys
20,000 SIOs			
Cyclic Redundancy Check (CRC)			

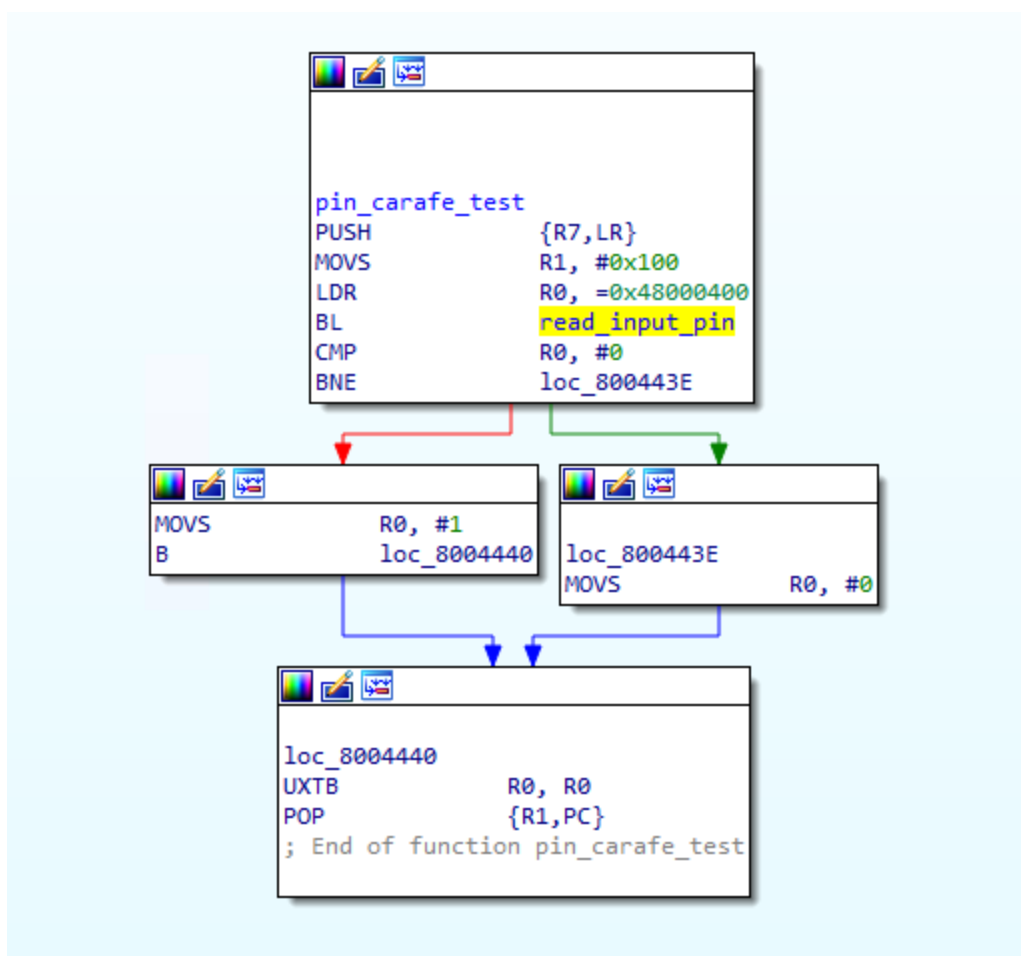
basic specs as can be found on STMicroelectronics web site

We know that this CPU is rudimentary and resource-limited enough that there won't be an operating system which usually makes RE easier, as there is not much code to go through. In this case, we used IDA (interactive disassembler) and after some trial and error attempts, we loaded the binary onto the starting address `0x08004000`. Normally, the CPU starts its execution from a `0x08000000` address, but later we found out that at the beginning of the address space there is a second custom bootloader (a piece of code that is there to allow updates of the rest of the code on chip) that takes care of the upgrade process and is always running as a safe place in the case something goes wrong during the update process. When you find yourself in the unfamiliar land of unknown firmware, you usually get something like this in IDA (that's the tool we use a lot in our threat labs to reverse engineer malware and unknown code generally):



IDA environment displaying functions and strings views

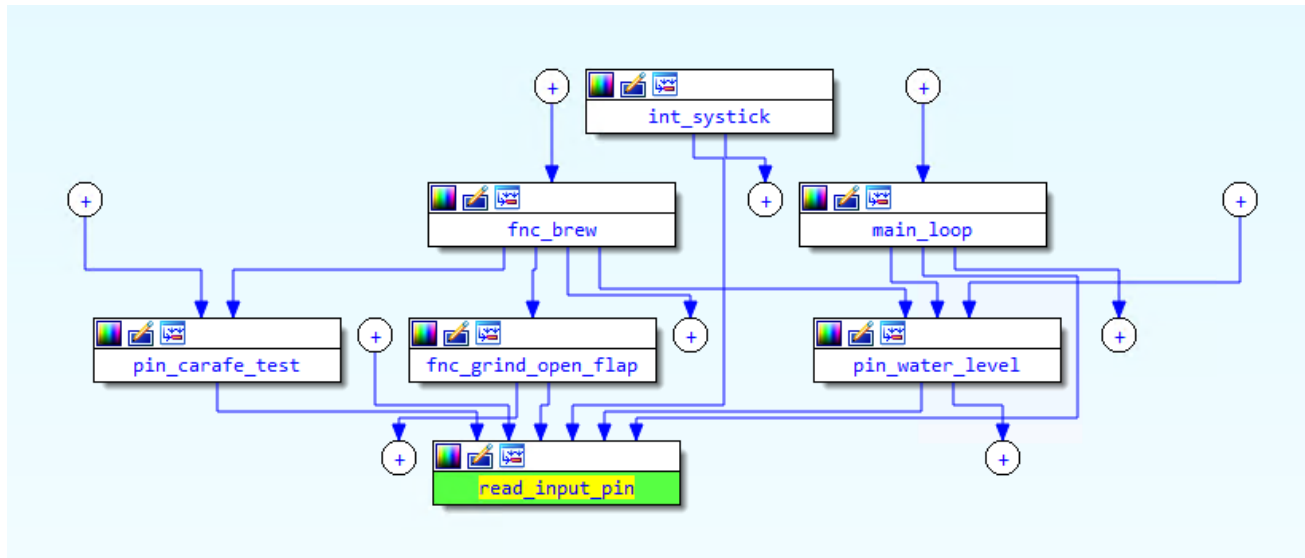
We see a number of unnamed routines and a list of strings. Now, the real work begins. You have to find which routine does what and slowly piece together how the whole thing works. It's a matter of preference, but I usually start with the strings. From a string you can usually deduce what functions use the string and based on that reference, you can usually uncover what the function does. It also helps to identify the most common and most referenced functions first. Specifically in the case of IoT, a good clue could be also to identify memory mapped I/O pins, so when you see reference to such memory, you know that function touches some HW component attached. Let's skip tedious parts of analysis and let us show you a few picks.



Using architecture

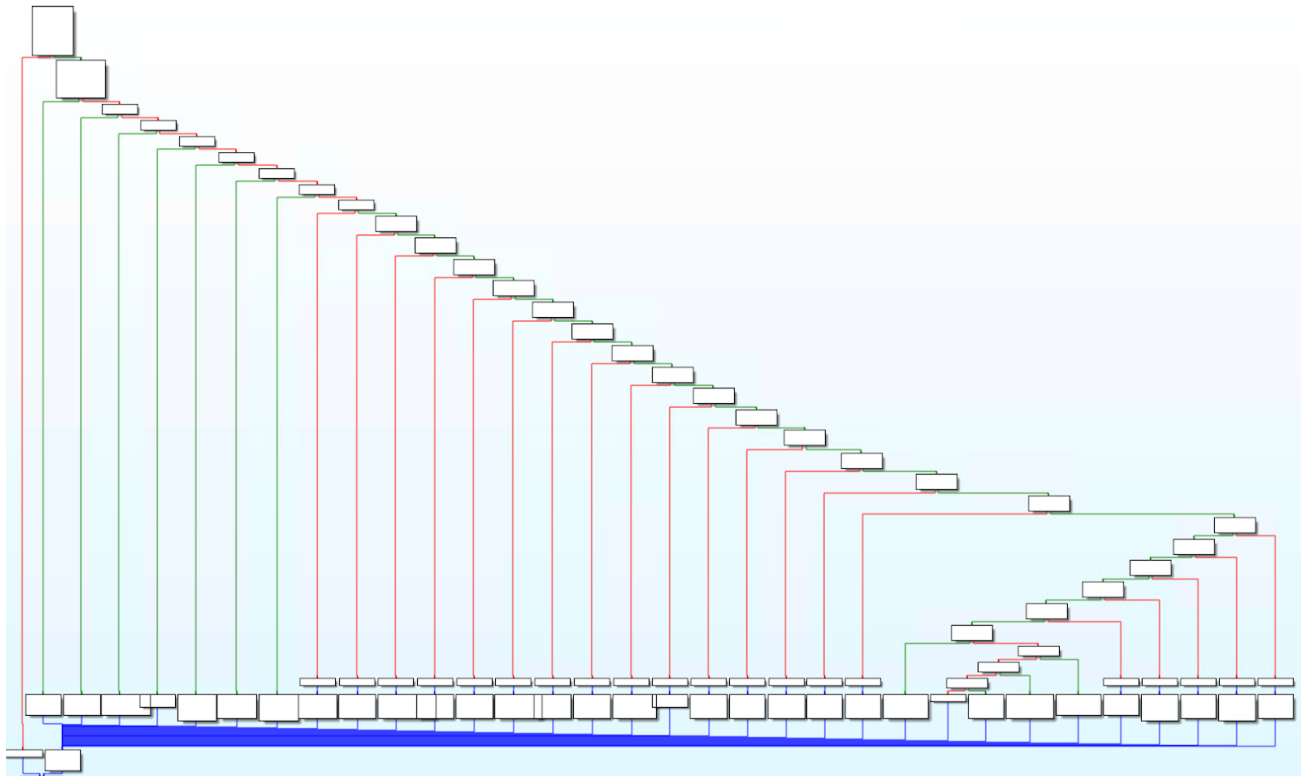
knowledge to understand code: From address seen we deduced that this must be function accessing missing carafe switch, thus function being called is generic function for reading pin and function calling will be part of 'brew' command

Although we didn't reverse all the functions, we got most of the important ones. We also discovered the global variables that store the state of buttons and received data over Wi-Fi, as well as the routines that control the display and all the gadgets. We also found the function for allocating and freeing memory, beeping, delay, etc. We found the main loop of a program that executes all the subfunctions as well as the main command routine – that is the function which based on the received packet over the network performs the command. This allowed us to create a list of all the remote commands that the coffee maker is able to perform and match them with previous research and even extend it.

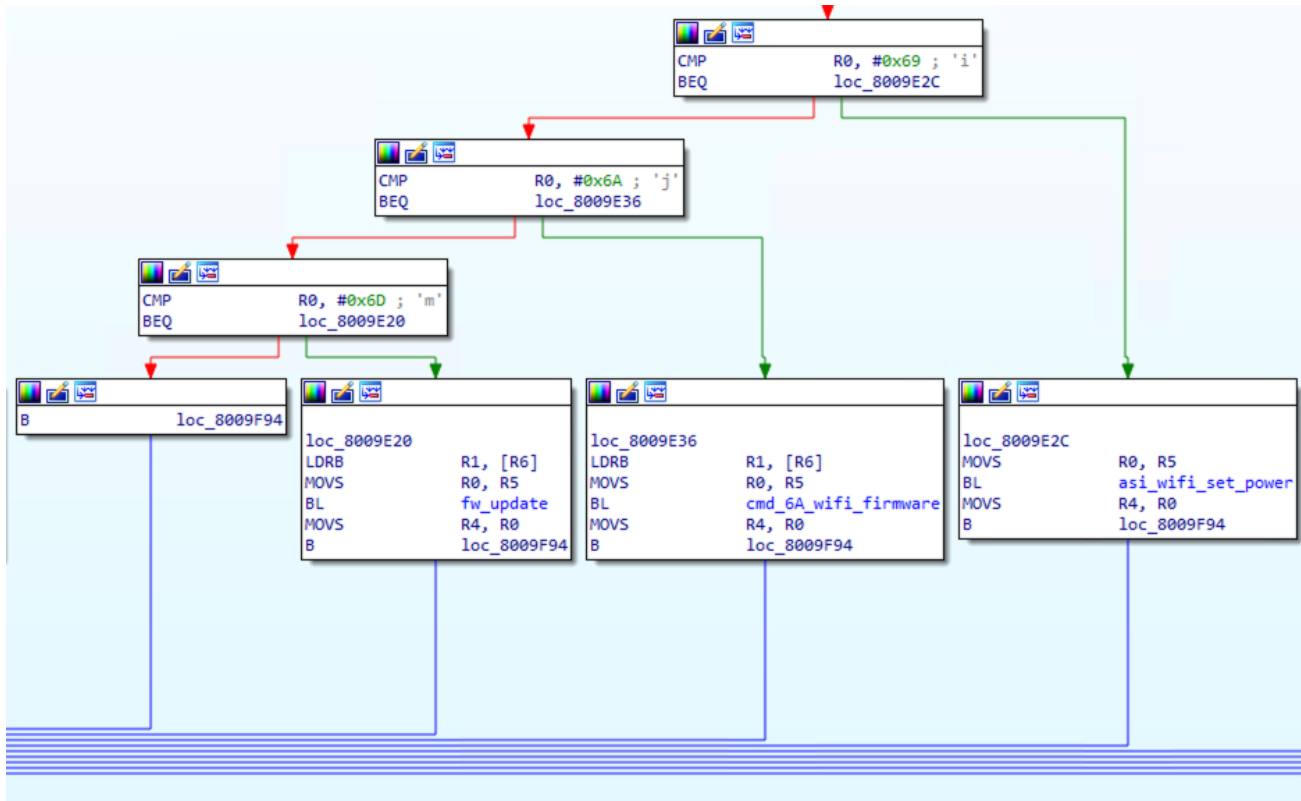


By cross referencing function calls and addresses we are slowly uncovering what each function does

After reversing the entire firmware, we knew exactly where every peripheral connects, how to control it, and all the commands that the coffee maker is able to perform. The most interesting was the firmware upgrade routine itself. After a closer look into the function that handles the command that induced the “update” screen, it turned out that the coffee maker notes something into the eeprom memory and performs the reboot. So now it was clear that the update process is handled by the bootloader which is not a part of a firmware package. Also when the “update” message is on the screen, the name of the Wi-Fi changes, from “Smarter Coffee:xx” where xx is the same as the lowest byte of the MAC address of the device. So obviously even if your coffee maker is connected to your home network when it comes to updating, it’s always done over the Wi-Fi in “Access Point” mode.



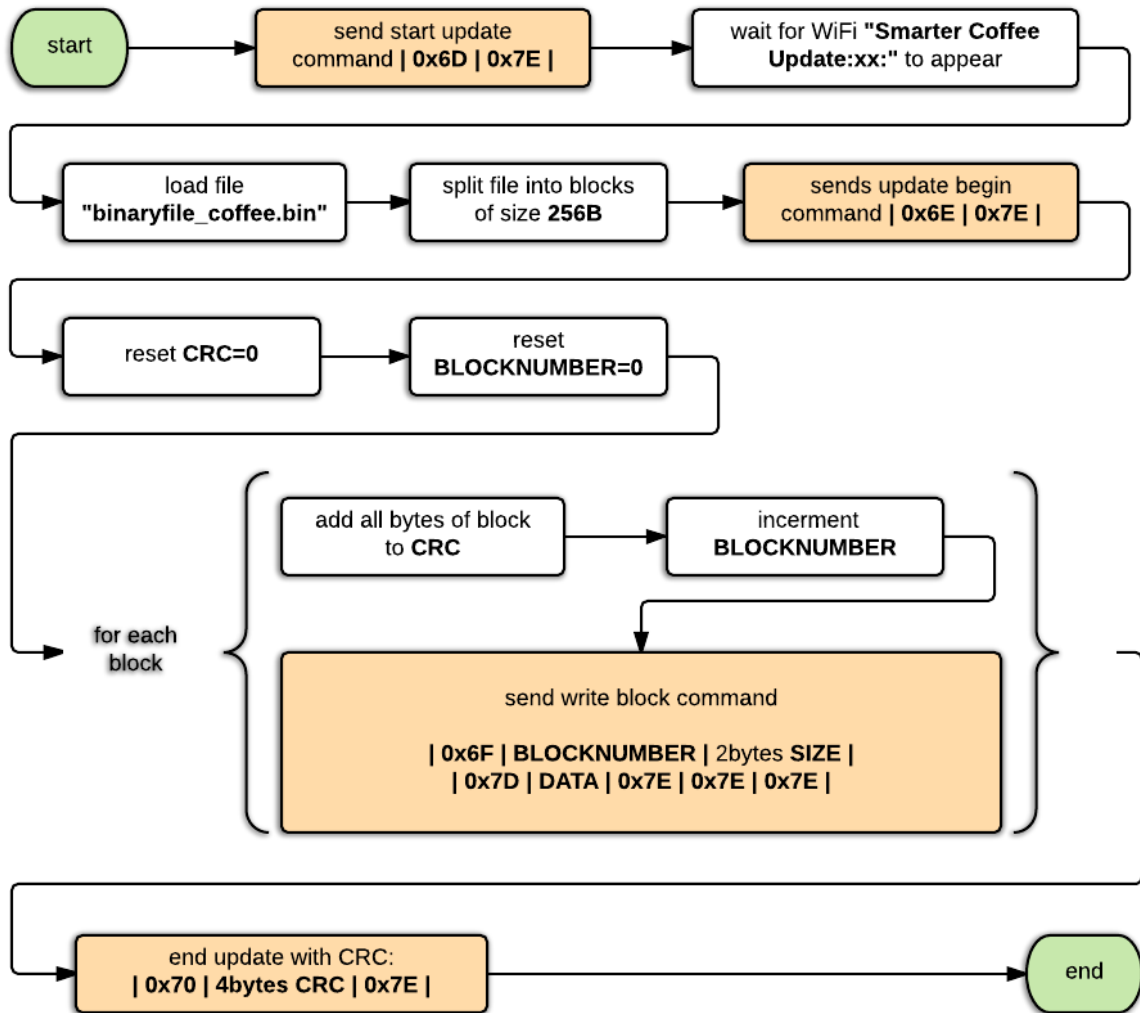
spotting main function that processes all commands based on appearance of the call graph



close-up of cluster of WiFi related functions

When we tried to reverse the update protocol we had a few different options. Since we knew what processor was inside, we could try to dump the whole firmware (including bootloader) out of the chip and reverse engineer the bootloader. We went with a more

simple approach and decided to reverse engineer the Android app and find the update part as it is usually easier to read decompiled Java code than CPU assembly language. It took us a while, but we identified and documented the whole process:



process of an update obtained by reversing the apk file

What is so surprising here is that the update procedure doesn't use any encryption or signature of the firmware. EVERYTHING is transmitted in PLAINTEXT over an UNSECURED WiFi connection, the only check is CRC at the end.

First update and modification

Now we have all the information we need to try our first update. Using Python we implemented a simple uploader, which works in two steps. First, it tries to find the coffee maker's Wi-Fi and then connects to it (by default, the coffee maker is always reachable on `192.168.4.1` port `2081`). By sending the "start update command", the coffee maker switches itself into the update mode.

```
0x6D terminator 0x7E
```

this simple (two bytes) command

switches coffee maker into update mode

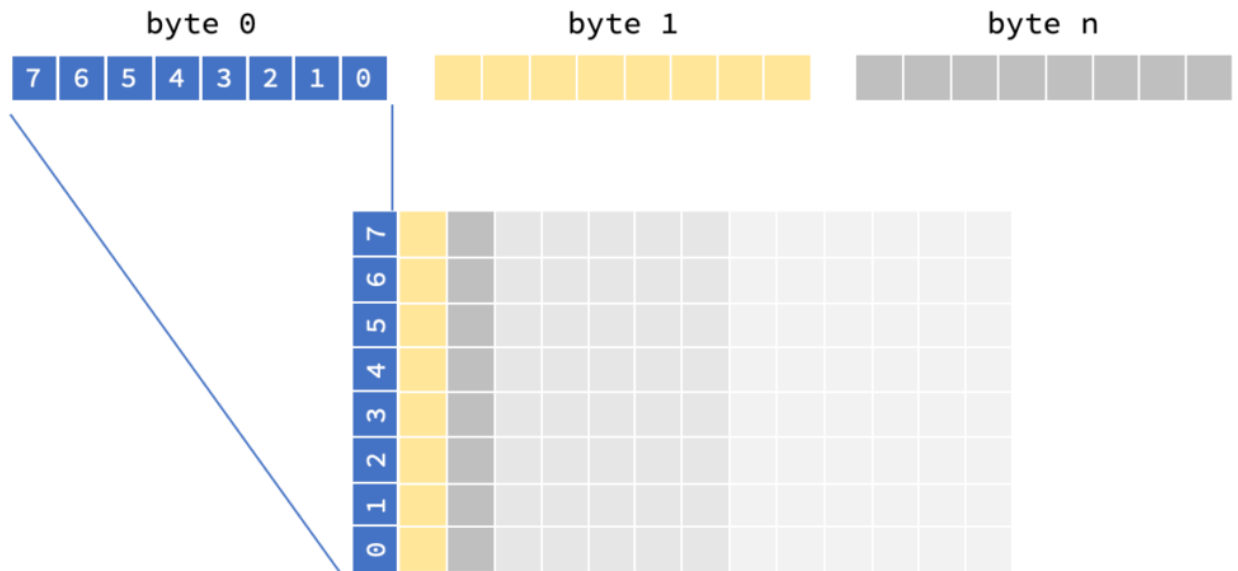
The script waits for the Wi-Fi to change from "Smarter Coffee:xx" to "Smarter Coffee Update:xx" . Once that happens, it connects again to the same address and sends new firmware using the aforementioned procedure. For starters, we just sent a slightly updated version of the firmware, which instead of displaying "missing carafe" displayed this:



So the whole procedure of replacing the firmware takes approx. 15-25 seconds and is completely automatic.

Everything is an image here

Interestingly, there is no font or textual output for the display, everything you see on screen is stored in the firmware as an image. So, there are, for example, images for the number of cups (1-12). The images are black and white and the encoding is pretty simple, as you can see in the following diagram:



black & white image encoding

Creating modified firmware

Now we have everything we need to create our own modified firmware. Originally, we wanted to prove the fact that this device could mine cryptocurrency, considering the CPU and architecture it is certainly doable, but at a speed of **8MHZ** it doesn't make any sense as the produced value of such a miner would be negligible.

We decided to turn the coffee maker into a ransomware machine where a certain trigger initiates the ransom message. It looks completely innocent and operates normally until the trigger is hit by an attacker making it even more surprising.

To do so, We hijacked one of the commands that originally served to connect the maker to the network so that when a user tries to connect the machine to home network a user can trigger the ransom behavior themselves unintentionally.

We used the unused memory space at the very end of the firmware to create the malicious code. By using the ARM assembler we created ransomware that when triggered renders the coffee maker unusable and asks for ransom, while at the same time turning on the hotbed, water dispensing heating element, permanently and spinning up the grinder, forever, displaying the ransom message and beeping. We thought this would be enough to freak any user out and make it a very stressful experience. The only thing the user can do at that point is unplug the coffee maker from the power socket.

```

27 .equ dwCoffeeStrengthButton, 0x200004D4
28 .equ dwStartButton, 0x200004D0
29
30
31
32
33 // ----- ENTRY -----
34 // hijacked command for WiFi connection
35 _start:
36 _reset:
37     PUSH        {R7,LR}
38
39     // turn on hotplate (controls pin directly)
40     LDR         R0, =0x48000414
41     LDRH        R0, [R0]
42     MOVS        R1, #1
43     ORR         R1, R0
44     LDR         R0, =0x48000414
45     STRH        R1, [R0]
46
47     // turn on heater (controls pin directly)
48     LDR         R0, =0x48000414
49     LDRH        R0, [R0]
50     MOVS        R1, #4
51     ORR         R1, R0
52     LDR         R0, =0x48000414
53     STRH        R1, [R0]
54
55
56     MOVS        R6, #0
57 _wait:
58     // reset watchdog
59     BL          reset_watchdog
60     // backlight on
61     BL          backlight_on
62     // display ransom
63     LDR         R0, =img_ransom_image
64     BL          disp_big_images

```

snippet of “malicious”

code: this is the part that displays ransom on the display

Here you can see it in action:

Plan(n/t)ing attack vectors

So now that we proved it's possible to change the coffee maker's firmware without touching it, let's have a look at how you can get your hands on the coffee maker as an attacker. What possibilities do we have and can we overcome the "security" measure of the newer firmware version, where the device owner is required to press the start button before the update happens.

Attack vector 1: Passer-by.

In this approach we need physical access to the device to initiate the update.

- + Easy to perform
- Need physical proximity, the coffee maker must be in an unconfigured state (not connected to a home network)

This attack option probably has the least impact. To perform it, you need access to the device or need to at least be in range of its Wi-Fi signal. If the device is already connected to the home network, you can use a deauthentication attack as the device is also vulnerable to "default-fallback to Wi-fi Access Point mode", when it's unable to connect to a pre-set WiFi network. Let's look at the scheme of this attack:

The attacker issues an "update start" command and then just after that, the Wi-Fi changes its name to "Smarter Coffee Update:xx". The attacker then starts sending modified firmware to the machine.

Attack vector 2a: Breaking the perimeter.

In this scenario we use the network to get to the coffee maker.

- + Can be done remotely
- Need to break into the network first, and need a Wi-Fi enabled device/router inside the network, if the coffee machine is running on a newer firmware version, the user needs to confirm the update by pressing the button.

This is a very common scenario where an attacker first breaks into the home network. The crucial part of this attack is to find a device via which you can control the Wi-Fi. An ideal candidate is, of course, a Wi-Fi router.

As the coffee maker always turns itself into a Wi-Fi access point while upgrading its firmware, you need a device inside the network you can use to connect to it. This is usually a pretty common function of any router, so for a short time, you can easily reconfigure the router to be a client, instead of a Wi-Fi access point. Because the update process is so short, the outage should be hardly noticeable. Some routers are even capable of being a client and access point at the same time using a single radio.

Attack vector 2b: Breaking the perimeter.

This attack is a variant of the previous attack vector, the only difference is that it fools the user into pressing the update button without any concerns.

- + Reach, overcome the user interaction problem
- Requires more planning and access to the network, not an instant effect

The novelty here is that in the exact moment when the app asks for the version of the firmware, the attacking script (running on the router) just spoofs the response and sends back a response indicating the coffee maker firmware is out-of-date, which leads to a prompt in the app that it needs to be updated. The rest is easy – you can now replace the firmware as it flows from the app to the coffee maker.

This attack could be also performed in a “passer-by” manner, when you have proximity to the coffee maker and it’s not joined to a network.

Attack vector 3: Social engineering. An android app as a mediator.

In this attack scenario, we trick the user into downloading a fake app to control the coffee maker.

- + No interaction with the network or protocol needed
- Social engineering, outcome is not guaranteed

This is a classic attack scenario. You just need to replace the firmware inside the coffee makers official companion Android app and re-sign it with a stolen certificate and try to push it into the Google Play Store or any other third party app store. You can use any social engineering technique to push the user into downloading the fake app, as there is no signature in the firmware, the coffee maker gladly accepts it. Basically, any form of creating a fake version of the app and pushing the user to install it works.

Found vulnerabilities and issues

During our analysis, we found several weaknesses and one critical vulnerability. As you can already imagine, the most serious vulnerability is the one that enables an attack to replace the coffee maker’s firmware remotely.

There are differences between firmware versions, the oldest one didn’t need any user interaction to be updated, whereas newer versions of the coffee maker’s firmware ask the user to press the “start” button on the machine to begin the update. However, even if it requires the user to press the button before the update, you can still use social engineering attacks by repeatedly sending the start update packet, which will cause the user to see the same prompt over and over, even when pressing “cancel”.

Because the update happens over an unencrypted network, you can also spoof the firmware version and make the companion app believe the coffee maker actually needs to update and then inject malicious code. In this case users will likely happily press update.

Another weakness is the possibility to permanently dissociate the device from the home Wi-Fi network. Even if it is already connected to a secure Wi-Fi network you can disconnect the machine from the network by using a deauthentication attack. Strangely enough, it will make no attempt to reconnect, so it loses connection forever, or at least until someone resets it. If you keep deauthing the device during its restart, it switches itself back into “Access Point” mode with open Wi-Fi waiting for the attacker to connect.

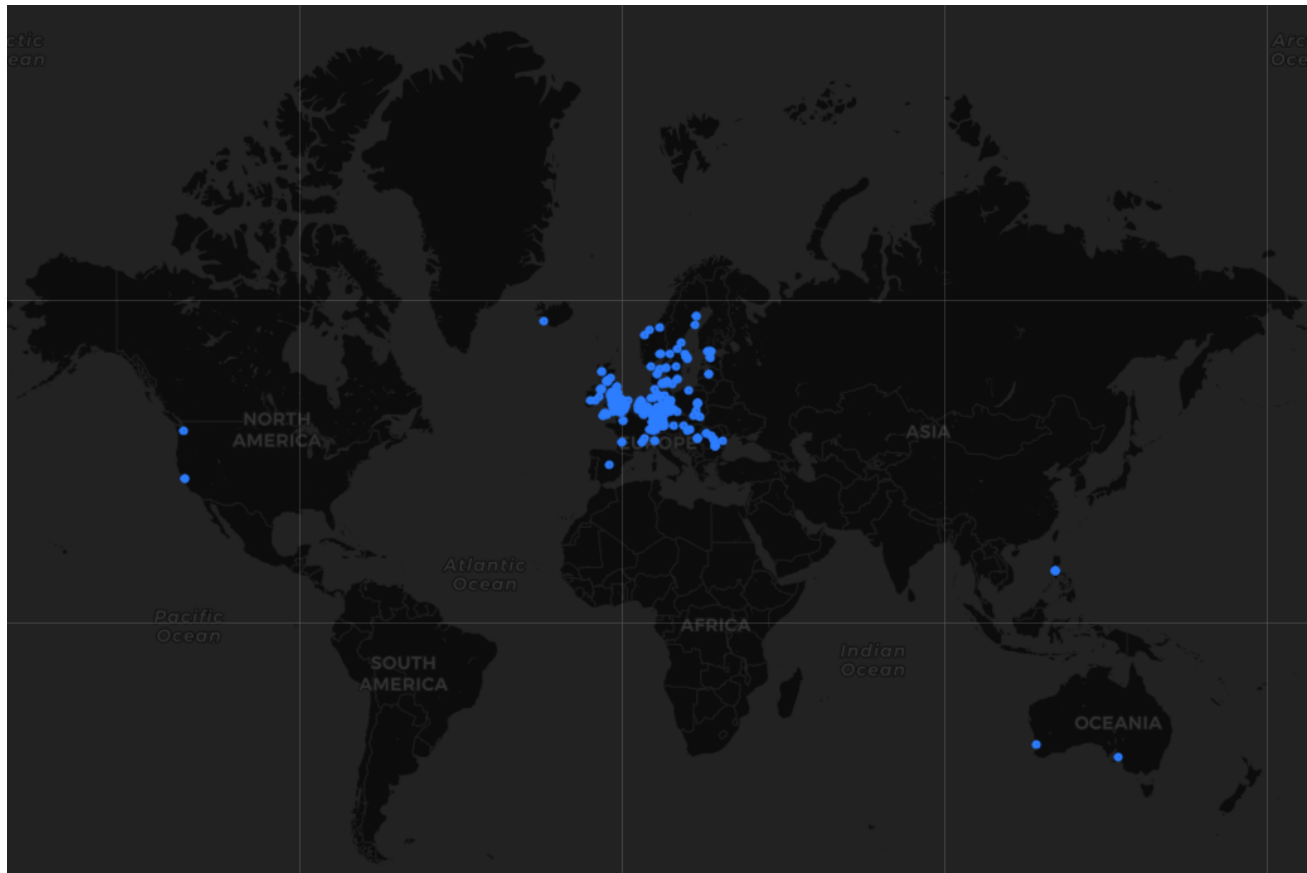
The disclosure and fix

Even if we were to contact the vendor, we would likely get no response. According to their website, this generation of coffee maker is no longer supported. So users should not expect a fix.

Final thoughts

Are you still interested in a smart coffee maker after all this? As shown on the map below (source: [wagle](#)) there are nearly **570 smart coffee makers** from this vendor out there that have not been set up. That means the smart functionalities are not being used by these **570** machines. However, the owners of these machines have unintentionally made it easier to hack their devices, as each of these coffee makers makes itself an access point to which anyone in range can connect and misuse.

Additionally, this case also demonstrates one of the most concerning issues with modern IoT devices: **“The lifespan of a typical fridge is 17 years, how long do you think vendors will support software for its smart functionality?”**. Sure, you can still use it even if it’s not getting updates anymore, but with the pace of IoT explosion and bad attitude to support, we are creating an army of abandoned vulnerable devices that can be misused for nefarious purposes such as network breaches, data leaks, ransomware attack and DDoS.



Unconfigured Smarter Coffee machines across the globe

As we said at the very beginning, in the security domain we used to consider software as an untrusted part of the ecosystem, while considering the hardware as secure and trusted. More and more often, we see how this trust is being broken. Unfortunately, many vendors make firmware attacks more viable by just leaving security behind and making it wide open to attackers. For cybercriminals this opens and the whole new world of attack surfaces to abuse. It may not be that easy to write and replace firmware, but the advantages of stealthiness and persistence you can achieve are just so tempting.

We live in a world where things talk to things, and where the number of smart things is slowly outnumbering the number of computers. These devices, for the most part, have no screen and can therefore mask malicious activities running in the background from their owners.

IoCs, CVE + artifacts

CVE-2020-15501 Smarter Coffee Maker before 2nd generation allows firmware replacement without authentication or authorization. User interaction is required to press a button. **NOTE: This vulnerability only affects products that are no longer supported by the maintainer.**

Distinct firmware images

SHA256

1eff6702b158b1554284f3ef6eb9d05748f43ba353d60954f21c6f20fd71e6ce

650a7bc7a55162988c77df34235c8e87eda9c8e2fcedd72b74c5f69e3edd088c

Github repository with firmware images, IDA database, uploader

[Avast IoC repository](#).

References to similar research:

[Reversing the Smarter Coffee IoT Machine Protocol to Make Coffee Using the Terminal](#)

<https://www.pentestpartners.com/security-blog/another-unsmart-smarter-app/>

github.com/Tristan79/iBrew(opens in a new tab).

<https://www.pentestpartners.com/security-blog/hacking-a-wi-fi-coffee-machine-part-1/>

Tagged [asanalysis](#), [cve](#), [hardware](#), [reversing](#), [vulnerability](#).