Networking and Interfacing IoT Platforms
Lab 2
Energy Efficient Programming

There are two parts to this assignment. Part one covers energy efficient programming on low power platforms. Part two consists of a digital reverse engineering challenge.

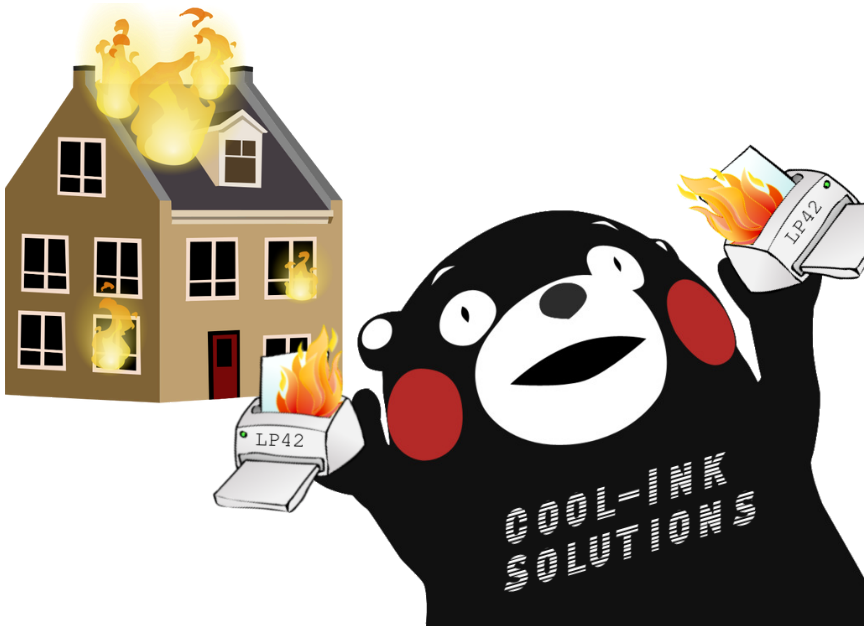
Important: Do NOT connect any circuits you create to a power source without permission from the teaching team! When you have finished creating your circuits, always have a teaching assistant come and inspect your work.

Note: Labs can be answered in Dutch or English, the important part is the contents. Paste lots of pictures/diagrams/… instead of writing walls of text. Do include all your failures in the description as well as tips, recommendations or the help you received from the teaching team! Remarks (Google Doc remarks) made by the teaching team must not be removed. Make a short movie after completion of a part and submit it as a separate file (no editing required !).

Part 1: Energy efficient programming

Assignment: LP42 on fire

The Cool-Ink printer company made a mistake in one of their latest enterprise printer designs. The Cool-Ink LP42 line printer has the capability of spitting out a whopping 100 pages per minute by using a new proprietary heating technology. Their product, however, was never fully tested for paper stalls... Whenever a paper stall occurs, the heating chamber heats the paper to combustion and thus sets the printer on fire. To make matters worse, the printer had no stall detection built-in. In case of a stall, the printer simply keeps feeding the heating chamber new paper against high speed, essentially stoking the oven.



The CoolInk printer company decided it would be more economically viable to "patch" this behaviour by somehow warning the local system administrator of a potential fire taking place. This is why they called you guys in!

Since the printer is proprietary, you can not hook into any present systems. As such you will have to design an external monitoring system using the MPC9808 (temperature sensor) that can detect a potential fire.

Cool-Ink put together the following list of (minimal) requirements:

- Deployment situations vary:
    - Some LP42 line printers are deployed under a contract stating that no more than 1 power socket may be used at location. The printer itself already uses this, so our monitoring solution may in some cases need to operated from a powerbank; the solution needs to be energy efficient!
    - Other LP42 line printers are deployed in zones where a potential fire could quickly cost lives; constant monitoring for a potential fire is a MUST in these zones.

- Some companies deploy multiple LP42 line printers, configuring all these hardcoded is a no-go; a central management system is required that can set the monitor system's parameters through a REST API
- In case of a fire, the central management system shall receive the classic "LP0 is on fire" message to at least notify the local system administrator
- The monitoring system has to be robust (i.e., what happens if a monitoring device drops out of the network, what if a monitoring device detects fire but can not contact the management system, …)
- Some basic form of security is required, only "known" monitoring devices can interact with the central management system (i.e., protection against forged messages, replay attacks, …)

Useful resources:
- https://docs.pycom.io/firmwareapi/pycom/machine/#power-functions
- https://docs.pycom.io/firmwareapi/pycom/pycom/#pycom-nvs-set-key-value
- https://github.com/JorisHerbots/micropython-mcp9808
- https://github.com/micropython/micropython-lib

Create a robust solution and be creative in your thinking! All energy related choices made must be backed up by a quantifiable statistic; there are USB testers available, use them. Document all your decisions, mistakes and thought processes. When you are done, take pictures of the finished setup(s) and film its workings.

Preliminary thoughts:
- Put PyCom in low power mode
- Let it wake up on I2C interrupt from sensor if possible, else have timed task that wakes up MCU every specified interval.
- Poll sensor and compare with threshold value
- Only if threshold exceeded, boot WiFi chip, connect to network and send predefined secure REST payload to endpoint.
- After success, shut down WiFi and go back to deep sleep.

Consummation measurements:

	Measured	Adjusted (- shield)
USB shield only	0.0151-0.0154 A	0 A
PyCom idle on REPL	0.1072 A	0,0921 A
PyCom in deepsleep	0.0167 A	0.0013 A
PyCom in sleep	0.0186 A	0.0032 A
Pycom idle on REPL with heartbeat and WLAN off	0.0455 A	0,0304 A
Pycom idle on REPL with heartbeat and WLAN on standby	0.0480 A	0.0326 A
Pycom idle on REPL with heartbeat, WLAN on standby and sensor powered	0.0480-0.0730 A	0.0326-0.0576 A

Battery life for a 1000mAh battery:

	Min	Max	Peak	Battery life when never triggering interrupt or timeout	Battery life when being awake for 1 minute a day with 5s peak
PyCom with deepsleep	0.0013A	0.0304A	0.1120A	769.23 hours	752.1 hours
PyCom in sleep	0.0032A	0.0326A	0.0576A	312.5 hours	310.24 hours

Calculations

Deepsleep
Average consumption = (1439 minutes * 0.0013A + 0.833 minutes * 0.0304A + 0.167 minutes * 0.112A) / (1440 minutes) = 0.00133A
⇒ Lifetime = 1Ah / 0.00073 = 752.1 hours

Sleep
Average consumption = (1439 minutes * 0.0032A + 0.833 minutes * 0,0326A + 0.167 minutes * 0.0576A) / (1440 minutes) = 0,00322A
⇒ Lifetime = 1Ah / 0.00073 = 310,24 hours

(see further for details about deep sleep and sleep)

We can conclude that the device can be active for at most a month when using deep sleep on a 1Ah battery, or about 13 days when using sleep.

Client

The PyCom can be in 2 states: booted for the first time or awoken by the temperature sensor.

Initializing the sensor

When booted for the first time, the WLAN will be started and will connect to the local network (assumed to have a fixed SSID and password for this scenario). It will then send a registration request to the server. Every product shipped to the customer already contains a generated private key. When connecting to the server, a secure channel is setup using an ssl socket. Encrypted data is interchanged to let the server know we are genuine. To do this, we sign our identifier (the PyCom's MAC address, fetched with `machine.unique_id()`) along with the postfix "_niip" with our private key. Both the server IP, port and endpoints are currently hardcoded in the firmware. To sign-up, a json containing our id and the signature is POSTed to the /sensors endpoint. The signature is the output of `crypto.generate_rsa_signature` with our id and private key. In this manner, we are sure that only legitimate devices know the private key and therefore can communicate with the server.

If the check passes, we are registered at the server and receive a public key. This key is later used to encrypt the messages that we send to the server. We can then fetch our settings by GETting the /config route, again with our id and signature. With the signature the server can verify that this sensor was already signed on and has received a public key. The server responds with a json containing two settings: crit_temp and crit_msg. If the sensor value ever reaches at least crit_temp, the PyCom is supposed to send crit_msg to the server on the /notify route. This payload will be encrypted with the received public key. With these settings received, the sensor is configured to trigger an interrupt on its ALERT pin when reaching the critical temperature. The required settings are also written to NVRAM so they can be restored later.

The temperature from the sensor is also measured, and if the current temperature is below the threshold, go to deep sleep, else the notify will be send.

Deep sleep

After the checking the sensor, the pyCom enters deep sleep mode, to save battery power. The PyCom will leave this mode upon an interrupt from the temperature sensor or a time interrupt.

Temperature interrupt

When the MCP9808 senses that the temperature has become higher than the critical value, it will fire an interrupt. The PyCom then wakes up and measures the machine temperature first. If this does not exceed an idle threshold (that was set on first boot) and does not exceed the threshold set in NVRAM (from the server), the PyCom will resume its slumber.

If the machine temperature does exceed the critical value, this is verified by getting the latest sensor measurement. If this is indeed larger than the threshold, the WLAN module is enabled and a connection to the server is made. We send a POST request to the /notify route with a JSON message containing our unique id and the fire message (temperature and critical message). If we are a registered sensor in the server's database, the message is handled.

If the temperature was not above the threshold, deep sleep will be entered again. An additional settings in the NVRAM could be used to not let the PyCom send multiple notifications, e.g. send one and save that temperature and only if the current temperature is back below that value, reset the setting and allow for new notifications.

Example output when waking up from PIN_INTERRUPT due to critical temperature:

```
Boot
Wake on PIN_INTERRUPT
Attempt to setup from remote...
Already verified
Checking sensor value...
Attempt to connect sensor...
Sensor OK
Notifying server...                          ⇒ Temperature exceeds threshold here
Attempt to create and connect socket...
Attempt to connect to WLAN...
Wait 1 of 10...
Wait 2 of 10...
Wait 3 of 10...
WLAN connected.
Socket OK
Sent notify: {"msg": "LP0 is on fire", "temperature": 26.25}
Going to sleep...
```

Time interrupt

This interrupt serves as a daily update for the configuration. We set this counter to 1 day (1000*60*60*24 = 86 400 000 ms or 10 seconds for our testing), so that the sensor updates it configuration each day. After receiving this timeout interrupt, the device will contact the server again and ask for a new configuration.
Another method would be a pushing system where the server triggers the PyCom to wake up when sending something to its IP address, but we did not have time to implement that. Furthermore, it is likely that the PyCom will not keep its IP address when deployed in the field and connecting to the local WiFi after being reset or rebooted.

Example output when waking up after timeout:

```
Boot
Wake on deep sleep timeout
Attempt to request CONFIG...
Attempt to create and connect socket...
Attempt to connect to WLAN...
Wait 1 of 10...
Wait 2 of 10...
Wait 3 of 10...
WLAN connected.
Socket OK
Send CONFIG request...
Got CONFIG.
Got from remote: {'crit_temp': 26, 'crit_msg': 'LP0 is on fire'}
Set t_thresh=2600
Set msg='LP0 is on fire'
Attempt to connect sensor...
Sensor OK
Sensor configured for critical alert.
```

```
Checking sensor value...
Attempt to connect sensor...
Sensor already OK
Temperature below critical value (2487 < 2600).
Going to sleep...
```

Sleep

Note that the PyCom has 2 sleep modes: `machine.sleep()` and `machine.deepsleep()`. We initially chose deep sleep because all peripherals are shut down in this mode and the power consumption is measured to be 0.0007A (without the shield). However, when booting the PyCom, after an interrupt or timeout is triggered, a consumption spike of about 0.097A is expected momentarily, after which the usage drops back down to 0,0304A (without shield). This could on average cause a higher consumption rate that needed.

The sleep() method on the other hand, just stops execution and puts only the MCU in a low power mode by simply stopping the CPU ticks. And just like deep sleep, it wakes up on interrupt, but this time just resumes execution from where the function was called. This means that there is no inherent peak consumption when resuming. Where deep sleep must reboot, sleep just resumes.

Both methods are supported in our driver code with the USE_DEEPSLEEP flag.

As can be seen in the power consumption table, deep sleep will on average consume less power. And we can conclude that the device can be active for at most a month when using deep sleep on a 1Ah battery, or about 13 days when using sleep.

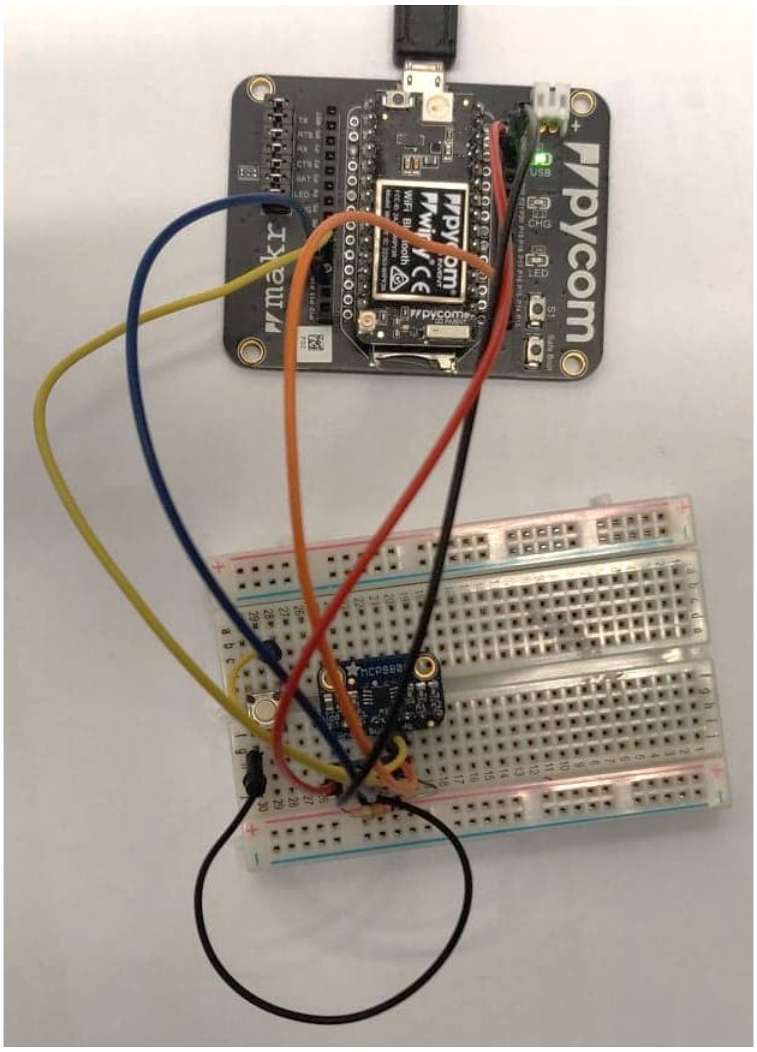
Notes

Every set-up function is called lazily: only when e.g. the WLAN interface is actually required, will it be set up. So when waking from interrupt, only the sensor is checked and only if the temperature actually exceeds the threshold is a connection made to the server. Functionality is included to retry doing an action a few times with a delay in between if it failed the first time, e.g. enable WLAN, connect and then wait and check if the connection succeeded.

Furthermore, when an error occurs in any process, the device is reset so it can try again. If this happens, the on-board LED will turn red.

Currently there is no additional way to notify anyone that is present if the printer is supposedly on fire and no network connection could be made. It is possible to include a loud buzzer or something similar to attract the attention to anyone on-site if this should occur.

Wiring



We used pin 13 as the sensor alert pin. And while everything was set up right (internal pull-up etc.), the interrupt would constantly be triggered. So as a solution we instead used an external 10K pull-up resistor. In the end we found out that we had overlooked the capabilities of pin 13: it does not support internal pull-up/downs.

Additionally we also kept the button to trigger the interrupt on demand for testing purposes without the sensor.

Server

The server was written in Python 2.7 (as supposed to Python 3.5 on the PyCom) and uses the flask and flask_restfull modules, supported by a SQLite database.

Sensors route

The first step when firing up the FireSensor will be signing up to the server. This can be done by performing a POST-command to the sensors route. In this request, we provide a payload with an identifier and a signature. With this information the server verifies if the sensor is legitimate. Verifying the signature is done by RSA-verification of the payload "DEVICE_ID + *_niip".*

After the sensor has signed up, the server will respond with the public key of the particular sensor. This public key can then be utilized to encrypt further communication between the sensor and the PyCom. The corresponding private key is saved in a database, so that the server can fetch it to decrypt the received messages.

Config route

Once the sensor is signed-up at the server, it fetches the configuration from the server. This can be done by performing a GET-request to the config route. This requests also contains the identifier and signature of the sensor, so that the server can verify the identity of the sensor.
The server then will respond with an encrypted payload containing the information about the critical temperature (when does it have to report a fire) and the message that has to be sent upon reaching the threshold.

Notifying the server

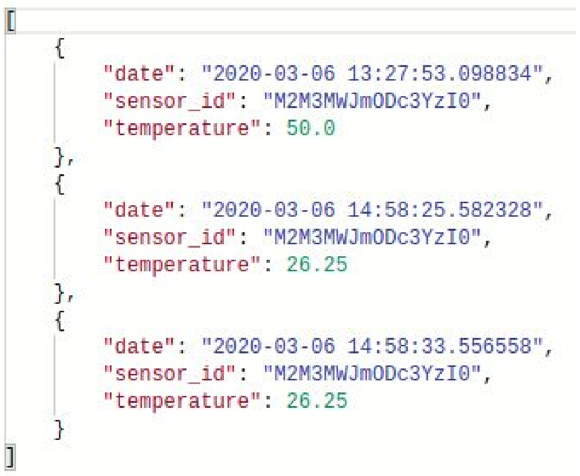
Now that the sensor is set up, it goes in monitoring mode. When sensing a fire, it will perform a POST-command to the notify-route. This command contains an encrypted payload which describes the current temperature and message to sent. Of course, there is also a signature verification at this route.

After receiving a notification from a sensor, the server should look in the database to identify the customer the particular sensor was deployed to (not in this implementation). This can be coupled to a mail service, so that the admin of the company receives an e-mail each time a fire occurs.

Fires route

Fetching the sensed fires can be done by performing a GET-request to the fires route. This will respond with a JSON-array containing the information about the past monitored fires.

An example can be seen on the right.



```
[
    {
        "date": "2020-03-06 13:27:53.098834",
        "sensor_id": "M2M3MWJmODc3YzI0",
        "temperature": 50.0
    },
    {
        "date": "2020-03-06 14:58:25.582328",
        "sensor_id": "M2M3MWJmODc3YzI0",
        "temperature": 26.25
    },
    {
        "date": "2020-03-06 14:58:33.556558",
        "sensor_id": "M2M3MWJmODc3YzI0",
        "temperature": 26.25
    }
]
```

Routes overview

Action	Method	Expected data

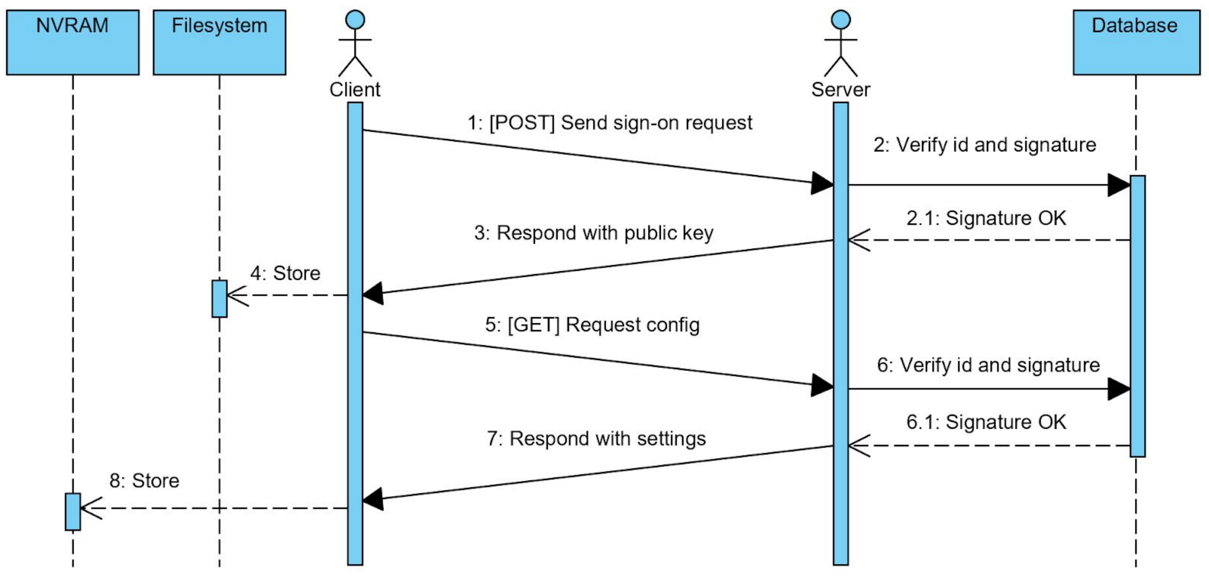
Sing up	POST /sensors	(sent to server)
		`{`
		`  "id": <sensor_id>,`
		`  "signature": <sensor_signature>`
		`}`

Get settings	GET /config	(sent from device)
		`{`
		`  "id": <sensor_id>,`
		`  "signature": <sensor_signature>`
		`}`
		
		(received from server)
		`{`
		`  "msg": <encrypted sensor settings>`
		`}`
		
		Settings take the form of:
		`{`
		`  "crit_temp": 51.23,`
		`  "crit_msg": "LP0 is on fire"`
		`}`
		
		And this json in encrypted with the devices public key.

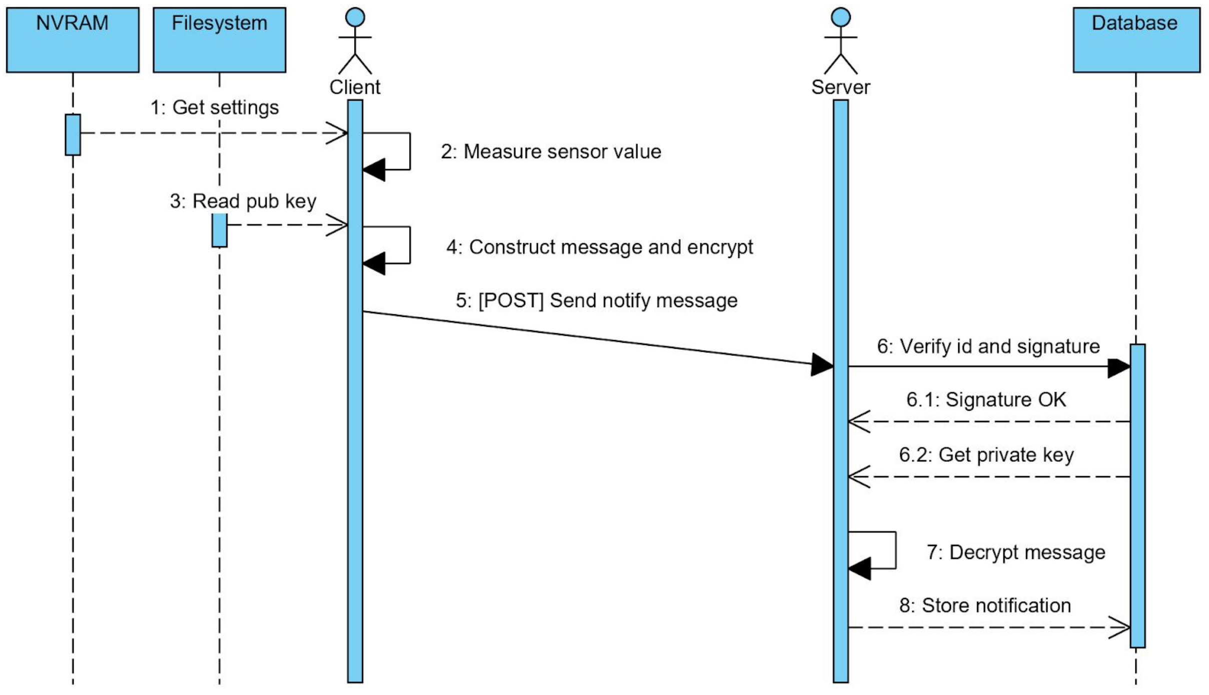
Notify fire	POST /notify	(sent from device)
		`{`
		`  "id": <sensor_id>,`
		`  "signature": <sensor_signature>,`
		`  "msg": <encrypted sensor message>`
		`}`
		
		Message contains:
		`{`
		`  "temperature": <measured temperature>,`
		`  "msg": <message on crit temp>`
		`}`

Data exchange

Setup: sign-on with POST to /sensors and GET to /config



Notify: Notify server with POST to /notify



Additional notes

By our design, the PyCom has a private key to create a signature and to decrypt payloads from the server. This key is written to the device when flashing it (so before giving it to a customer). In addition, the device id is already added to our database beforehand, so that it is possible to track which device has been send of to which customer. This is however currently not implemented.

For the public-private key encryption, we use RSA, both on the server and on the PyCom. We saw however, that the PyCom was not able to decrypt the encrypted settings received from the /config route. It turns out that Python at the server side, utilizes PCSK1 padding, which is not supported on the PyCom. The PyCom only can decrypt raw RSA ciphers, which is considered very unsafe. Python however, can only encrypt with some sort of padding. Therefore we were not able to encrypt the communication between the PyCom and the API.

Our first try to solve this issue is by implementing Textbook RSA in Python, but we could not find a proper solution. Every documentation or resource we considered, pointed to the fact that Textbook RSA is very unsafe and should not be used anymore.
Secondly, we tried to find a library which enables RSA with padding possibilities on the PyCom. But we did not find a library that suits our needs.

So, we were not able to solve this issue in the given time and we decided to add a variable to the server and the client that states if it has to use encryption or not. The implementation therefore is able to handle encrypted messages, but by default will consider all communication as plain text. The signature verification on the other hand, does work as intended.

The connection itself uses the PyCom's SSL socket, but without a certificate. generating a CA certificate for our server and putting it on each device (with its private key), would result in additional security.

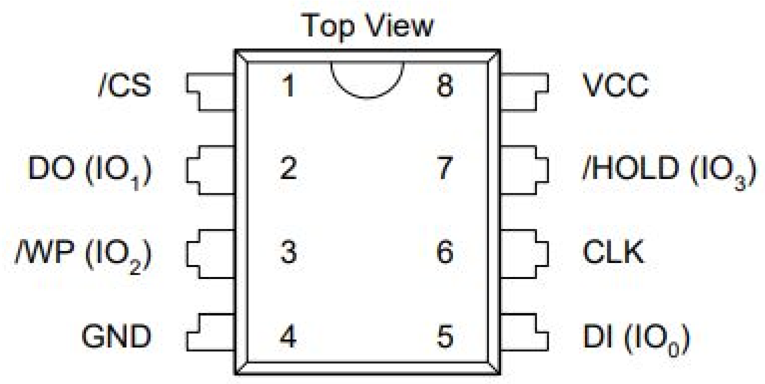
Part 2: Reverse engineering

In part 2, we will be looking at techniques to reverse engineer common components on IoT devices.

Flash

First, we will have a look at flash memory chips. These devices are currently the most popular way to store non-volatile information such as firmwares and settings. They are preferred over old-fashioned ROMs/EEPROMS/RAM because they can easily be upgraded and do not require a permanent power source to keep their state.

In this assignment, we will be looking at SPI-enabled flash chips. The workings of the SPI bus works have already been explained in the first lab. Typical SPI-enabled flash chips have a pinout like this (be sure to have the chip turned in the right direction !):



You will probably recognise some of these connections : Chip Select, Clock and DI/DO (which are equal to MOSI/MISO pins in other diagrams). For reference, you may assume that /HOLD and /WP are to be permanently connected to VCC (=3.3V).

We will assume the role of a person that wants to reverse engineer the firmware/settings of an existing device. The most difficult part has already been done for you : the physical extraction of the memory chip from the PCB. We have soldered these onto breakout boards that you can place on a breadboard.

Because a standard pc (as you know) does not have an SPI interface, we will be using a raspberry which, of which the SoC does have this capability. Alternatives to this approach are USB converters/readers that you can connect to your pc. For our purposes, a raspberry pi will do just fine.

Have a look at the documentation of the RPI on how to attach the flash memory chip to the spi bus on the raspberry pi (note : these are dedicated pins on the GPIO header of the raspberry). You are provided with a vanilla 'raspbian' image for the raspberry pi, so some settings may need to be changed.

Ask the teaching team to verify your connections before powering on, we only have 2 flash chips.

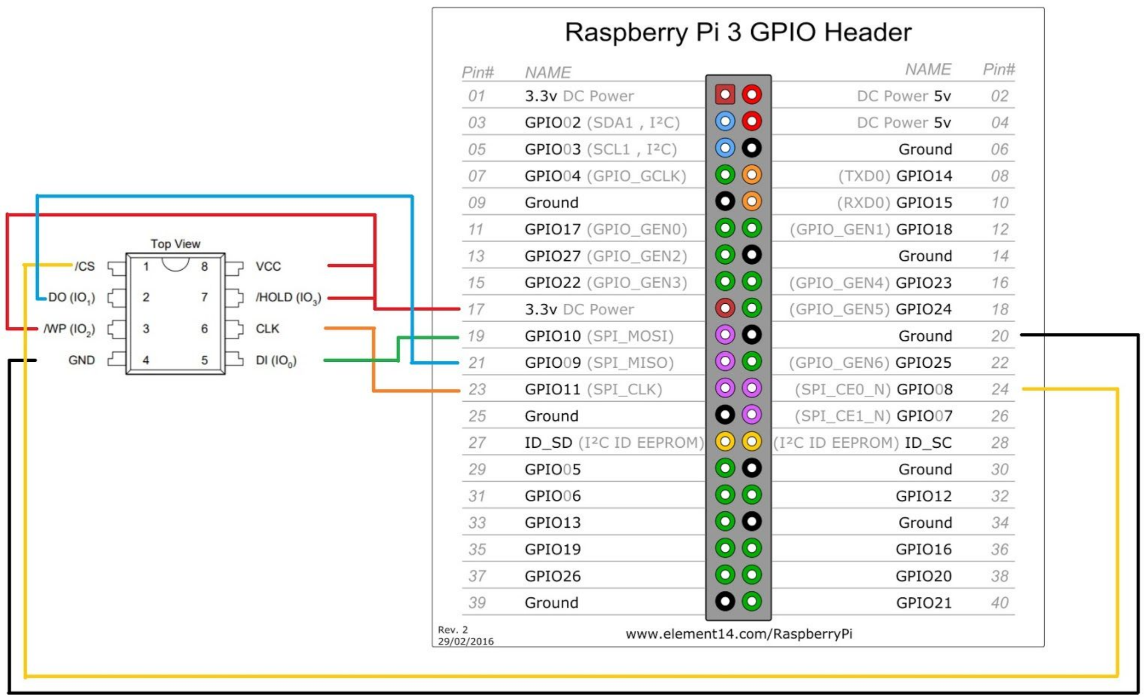
A pointer in the right direction : have a look at the 'flashrom' utility; it is compatible with the RPi and has support for the SPI flash chips we are using in this lab.

Step 1 is to extract the firmware from the 2 flash chips provided. You should end up with image files that contain a raw dump of the contents.

After verifying that the SPI port worked on the RPi with the following commands: (connect MISO and MOSI pins together)

```
sudo pigpiod
> pigs spio 0 50000 0
> 0
> pigs spix 0 10 20 30 40 50
> 5 10 20 30 40 50  if ok
> 5 0 0 0 0 0       if not ok
```

We connected the flash chip in the following way:


Raspberry Pi 3 GPIO Header
Top View
/CS
DO (IO$_1$)
/WP (IO$_2$)
GND
VCC
/HOLD (IO$_3$)
CLK
DI (IO$_0$)
Pin# NAME NAME Pin#
01 3.3v DC Power | DC Power 5v 02
03 GPIO02 (SDA1 , I²C) | DC Power 5v 04
05 GPIO03 (SCL1 , I²C) | Ground 06
07 GPIO04 (GPIO_GCLK) | (TXD0) GPIO14 08
09 Ground | (RXD0) GPIO15 10
11 GPIO17 (GPIO_GEN0) | (GPIO_GEN1) GPIO18 12
13 GPIO27 (GPIO_GEN2) | Ground 14
15 GPIO22 (GPIO_GEN3) | (GPIO_GEN4) GPIO23 16
17 3.3v DC Power | (GPIO_GEN5) GPIO24 18
19 GPIO10 (SPI_MOSI) | Ground 20
21 GPIO09 (SPI_MISO) | (GPIO_GEN6) GPIO25 22
23 GPIO11 (SPI_CLK) | (SPI_CE0_N) GPIO08 24
25 Ground | (SPI_CE1_N) GPIO07 26
27 ID_SD (I²C ID EEPROM) | (I²C ID EEPROM) ID_SC 28
29 GPIO05 | Ground 30
31 GPIO06 | GPIO12 32
33 GPIO13 | Ground 34
35 GPIO19 | GPIO16 36
37 GPIO26 | GPIO20 38
39 Ground | GPIO21 40
Rev. 2 29/02/2016
www.element14.com/RaspberryPi

When trying to run the flashrom utility with the following command:

```
> flashrom -p linux_spi:dev=/dev/spidev0.0 -r chip1.rom
```

We didn't get any response. Flashrom stated there was no flash/EEPROM chip detected on the SPI bus. We tried to verify SPI again, use the second slave select, rebooted and even tried the other RPi, but nothing seemed to work. After verification from one of the assistants, if became apparent that it should work this way, although it didn't.

After un-wiring and rewiring the chip, flashrom suddenly did detect an "unknown SPI flash device". Suspecting the power supply might be unstable, we connected a capacitor between the 3V3 and ground pins next to the chip. This did nothing however.

After looking through some examples, when flashrom says it does not recognise a chip, a speed must be given. Doing this with the following command gave the expected result, and flashrom started dumping the 8Mb memory from the chip to a file, after selecting the proper type of chip that is.

For chip 1:
```
> flashrom -p linux_spi:dev=/dev/spidev0.0,spispeed=1000 -r chip1.rom -c
"MX25L6406E/MX25L6408E"
```

For chip 2:
```
> flashrom -p linux_spi:dev=/dev/spidev0.0,spispeed=1000 -r chip2.rom -c W25Q64.V
```

Step 2 is to try and extract readable information from the firmware image. To do this, we ask you to first copy the image files over to your own laptop and not to install/run the entire toolchain on the raspberry pi. Try and find tools that can help you in this job (a hex dump tool is a first step, but there are more advanced tools out there). Try to find out what devices these chips belong to. If you can find one in the image, what is the job of the bootloader (if not, describe the working in general)?

To "reverse engineer" the extracted rom file, we used a combination of the unix `strings` program to extract readable data from the image and `binwalk` to view and extract the filesystems. e.g.:
```
> strings chip1.rom > chip1.strings.txt
> binwalk chip1.rom > chip1.fs.txt
> binwalk -eM chip1.rom
```

(to extract the squashfs image from chip2, squashfs-tools was used. A patched version is included in the tools directory.)

Firmware from chip 1:

The device is from Gigaset Communications GmbH (Dusseldorf, North Rhine-Westphalia), and appears to be a network connected device called the "REEF basestation".
When flashing firmware, it tries to write a filesystem to `/dev/mtdblock5` and run scripts such as `/mnt/data/post_update.sh`. The rom contains various zlib and jffs2 images in addition to a linux image: vmlinux.bin.

From the image:
```
    ,;:.
  ;;;;;;;                                   :,
 ;;:    .                                   ;;
 ;;        ,   `;;;.  `:;;,   .;;:   `;;:  :;;:,
,;`       `;  :;;:;;, :;,:;: ,;:.;  .;;,;; ;;;;
:;        `;  ;:   ;,     :; ;;     ;:   ;. ;;
:;    ;;;.`; ,;    ;,  .:;;; ,;;.  `;....;: ;;
,;`   ..;.`; :;    ;, ;;,.:;  .;;; `;;;;;;: ;;
 ;;     ;.`; ,;`   ;,.;   :;    `;; ;`      ;;
 :;;   `;.`;  ;;  ,;,`;`  : ;   ;; ;;   .  ;;
  ,;;;;;;``;  `;;;;;, ;;;;:; ;;;;;   ;;;;;  ,;;;
    `..            :,  ``      .`     `.`     .
                  `:.
              ::.,::
              `::::
Runing on SC14452 SoC
SiTelboot
```

```
 _______  _______  _______  _______ 
(  ____ )(  ____ \(  ____ \(  ____ \
| (    )|| (    \/| (    \/| (    \/
| (____)|| (__    | (__    | (__    
|     __)|  __)   |  __)   |  __)   
| (\ (   | (      | (      | (      
| ) \ \__| (____/\| (____/\| )      
|/   \__/(_______/(_______/|/       
```

Boot info

```
bootargs=noinitrd root=/dev/ram0 rw init=/linuxrc earlyprintk=serial console=ttyS0
bootcmd=bootm E8000
bootdelay=1
baudrate=115200
ethaddr=02:4e:ef:09:19:20
ipaddr=192.168.1.10
serverip=192.168.1.34
netmask=255.255.255.0
bootfile="vmlinuz"
boot_from=flash
board_rev=RevB
1st_boot_pos=E8000
2nd_boot_pos=46F000
rec_boot_pos=20000
boot_from_image_no=1
```

It checks for firmware checksum at:
https://supplies.gigaset-elements.de/firmware/bas-001.000.043/up_vmlinuz.bin.sum
And tries to connect to supplies.gigaset-elements.de (78.137.103.175:80)

Underlying linux version:
Linux kernel version "2.6.19-uc1reef-dirty (filug@gigawork) (gcc version 4.1.2) #3 Fri Jan 29 08:35:14 CET 2016"

Linux image contains:

- Certificates
    - SSL (possibly also public/private keys)
        - Key pair generated with `/usr/bin/coco --genkey --out ${KEYPAIR}`
    - Gigaset Class 3 Private Primary Certification Authority - R1
    - Jenkins test cert
    - StartCom Certification Authority
    - Thawte Premium Server CA
    - thawte Primary Root CA
    - thawte Primary Root CA - G2
    - thawte Primary Root CA - G3
    - ValiCert Class 2 Policy Validation Authority
- References to endnodes, sensors and sirens
- Sensors: with commands sent to loopback interface (127.0.0.1)
    - e.g. "listen on CloudRX while trying to delete the sensor"
    - A sensor POST command for deletion:
      ```
      {"method":"POST",
      "uri":"https://api-bs.gigaset-elements.de/api/v1/endnode///sink/ev",
      "payload": {"payload": "deleted"}, "clientId": 138}
      ```

    - Command for update: `{"cmd":"fwupdate","devId":"","file":""}`
    - These might indicate the REEF basestation has or is paired with other network nodes and can receive data from them or update their firmware (from /mnt/data/fw).
- Script to unpair a device (deleting `/mnt/data/cert/cert.crt` and `/mnt/data/cache/*` and then calling `sync`)
- Various setup scripts, mostly part of BusyBox
- Switchable environments:
    - "prod" (probably production): https://api-bs.gigaset-elements.de
    - "stg": https://stg-api-bs.gigatest.pl
    - "int": https://int-api-bs.gigatest.pl
- Mentions to the filesystem being locked or not. If unlocked, files can be run with `/mnt/data/private.sh <file>`
- basestation hostname: reefbs (for a device REEF basestation)
- The /dev/mtdblock* filesystem gets mounted to /mnt/data
- Among others, a folder /mnt/data/fw/sensors is created, indicating the availability of one or more sensors.
- On start, the following executables are started in the background:
    - /usr/bin/jbus &
    - /usr/bin/wdt &
    - silencium /usr/bin/coco
    - /usr/bin/tram &
    - /usr/bin/sens &
    - /usr/bin/pyta &
    - /usr/bin/jkeymon &
    - /usr/bin/qpa &
    - /usr/bin/uleapp -6 -dmxce -v ${ULEOPT} &
    - /usr/bin/tick &
    - /usr/bin/rebus &
- It has a powerled that is turned on after booting and at least 1 ethernet interface (eth0) that is started. A loopback interface is also configured:

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet dhcp
```

- A script to make leds (presumably the powerled or similar) blink (off, on, slow (1s delay), fast (250ms delay)).
- A telnet session is possibly started with:

```
telnet stream tcp    nowait root   /usr/sbin/telnetd /usr/sbin/telnetd
```

Firmware from chip 2:

This firmware belongs to a Broadcom Corporation ethernet device and contains or is a Broadcom BCM47XX 10/100/1000 Mbps Ethernet Controller (BCM4716a0). Going further in the rom, we can see a reference to the Linksys E2500, so this might be firmware for that router, that contains a Broadcom chip for networking.

Contains:

- References to:
    - 0:ledbh0=11 (interface 0 led, e.g. ethernet port 1)
    - 0:ledbh1=11 (interface 1 led)
    - 0:ledbh2=11 (interface 2 led)
    - 0:ledbh3=11 (interface 3 led)
    - 0:leddc=0xFFFF (other interface led, e.g. wifi)
    - 0:temps_period=5
    - 0:tempthresh=120 (120°F = 48,89°C, probably to throttle CPU)
    - 0:temps_hysteresis=5
    - 0:phycal_tempdelta=0
    - 0:tempoffset=0
    - robo_reset, robo_ss, robo_sck, robo_mosi, robo_miso
        - Maybe something on SPI?

  Indicating controllable LEDs and a temperature sensor. The leds are probably the cable connected indicators on ethernet ports.
- A couple of default IP addresses (like 192.168.1.1/.254)
    - Also references to eth0 interface, so this device also has a network interface
    - Strings indicating a changeable MAC address
    - tftp to upload a new image for firmware upgrade
- References to:
    - BCM4716a0 (Broadcom IEEE 802.11N 2.4-GHZ + 5-GHZ ROUTER SOC)
    - Atmel (Arduino-like MCU family)
    - ST compatible (STMicroelectronics)
    - Broadcom BCM47XX 10/100/1000 Mbps Ethernet Controller
- A user called: root@GavinPC.localdomain
- References to firmware files (maybe indicating OpenWRT the open-source routing firmware?):
    - mfgtest_wrt610nv2.bin
    - mfgtest_wrt610n.bin
- HTTP requests:
    - GET /Upgrade.asp
    - GET /index.asp
    - POST /upgrade.cgi
    - GET /cancel.asp
- Parameters:
    - `boardrev=0x1101`
    - `lan_hwaddr          =58:6D:8F:7B:A5:F7`
    - `http_client_mac     =64:51:06:07:FF:22`
    - `device_info_mac0    =64:51:06:07:ff:22`
    - `wan_hwaddr          =58:6D:8F:7B:A5:F8`
    - `wl0_hwaddr          =58:6D:8F:7B:A5:F9`
    - `wl0.1_hwaddr        =58:6D:8F:7B:A5:FB`
    - `wl0.2_hwaddr        =5A:6D:8F:7B:A5:FA`

  - wl0.3_hwaddr   =5A:6D:8F:7B:A5:FB
  - wl0.4_hwaddr   =5A:6D:8F:7B:A5:FC
  - wl0.5_hwaddr   =5A:6D:8F:7B:A5:FD
  - wl0.6_hwaddr   =5A:6D:8F:7B:A5:FE
  - wl0.7_hwaddr   =5A:6D:8F:7B:A5:FF
  - wl0.8_hwaddr   =5A:6D:8F:7B:A5:F0
  - wl0.9_hwaddr   =5A:6D:8F:7B:A5:F1
  - wl0.10_hwaddr   =5A:6D:8F:7B:A5:F2
  - wl0.11_hwaddr   =5A:6D:8F:7B:A5:F3
  - wl0.12_hwaddr   =5A:6D:8F:7B:A5:F4
  - wl0.13_hwaddr   =5A:6D:8F:7B:A5:F5
  - wl0.14_hwaddr   =5A:6D:8F:7B:A5:F6
  - wl0.15_hwaddr   =5A:6D:8F:7B:A5:F7
  - wl1_hwaddr   =58:6D:8F:7B:A5:FA
  - wl1.1_hwaddr   =5A:6D:8F:7B:A5:FB
  - wl1.2_hwaddr   =5A:6D:8F:7B:A5:F8
  - wl1.3_hwaddr   =5A:6D:8F:7B:A5:F9
  - hwaddr_24g   =58:6D:8F:7B:A5:F9
  - hwaddr_5g   =58:6D:8F:7B:A5:FA
  - aftr_addr   =2001:db8:ff4e:4::4
  - gn_lan_ipaddr   =192.168.33.1
  - wps_modelname=Linksys E2500
  - wps_device_pin=53211842
  - wpa_psk_24g=wij zijn secure
  - hnap_devicename=TTISD
  - gpio9=wps_button   (WPS button)
  - gpio23=wombo_reset (reset button)
  - wl0_ssid=TTISD
  - wl0_wpa_psk=wij zijn secure
  - wl0_crypto=tkip+aes
  - wl1_ssid=TTISD
  - wps_random_ssid_prefix=Cisco_WPS_
  - wl0_radioids=BCM2057
  - wl_default_ssid=Cisco82134
  - http_client_ip=192.168.0.118
  - gn_account_password=guest

  - os_version=5.60.127.2901
  - os_ram_addr=80001000 (could possibly be used for decompilation)
- It appears the w10 wlan interface (2.4GHz wifi) is configured with the SSID "TTISD" (possibly referring to the course given by prof. dr. Ramakers), with the password "wij zijn secure".

Step 3 is to find some useful information contained in the image that you could not have accessed otherwise (e.g. security related information or inner workings of software) - it is certainly not guaranteed that this is actually in the image, but give it a go.

From chip 1:

- Device: Gigaset "REEF" base station
- API commands such as `{"method":"POST", "uri":"https://api-bs.gigaset-elements.de/api/v1/endnode///sink/ev", "payload": {"payload": "deleted"}, "clientId": 138}`
  could be exploited (DNS cache poisoning to create our own server to receive callbacks or the payload above could be used to pretend to be a base station to the original server)
- URLs such as https://stg-api-bs.gigatest.pl indicate the developments of this device was probably done in Poland.
- The device has CA certificates and can be paired to other devices to update their firmware.
  Now that we have the certificates, we could spoof an existing device and thereby trick the existing network in believing our device is part of the network.

From chip 2:

- Device: `Linksys E2500`, with Broadcom BCM4716a0 chip and running linux.
- WLAN interface (2.4GHz wifi) is configured with:
  - SSID: "TTISD"
  - Password: "wij zijn secure"
  - Using WPA2 and `tkip+aes`
- The default WPS pin is `53211842`
- The WPS-pin is on gpio9:
  We could alter the firmware so that on a specific network request, the WPS function gets activated. For example by adding additional code in a boot script. Now that we know the pin code, we could easily get access to the network, even if the credentials are changed afterwards.

The mystery device

This assignment should be considered an 'extra'. Finish the others parts first; if you have time left you can try your hands on this one.

We have uploaded some code to a pycom device. It is your job to figure out what this device is actually doing (both on micro and macro level). Do not download the code from the pycom, that's not fun. You are provided with the following information:

P8 is an output pin that is to be connected to a bus which is active low (so use external pull-ups to keep it high during periods of non-activity).
P22 is an input pin that is active low (but in this case, the pull-up is internal, so no need to add extra components).

Create your own setup to run tests on the mystery device. Make sure you keep in mind the above-mentioned details.

Attach the logic analyzer to the setup and try to figure out what is going on - remember to connect both devices to the same GND as a reference voltage. To do this analysis, you cannot simply use a built-in protocol decoder like I2C or SPI (hint : this is not a standard protocol that you will find in the list of supported protocols). Try to figure out the exact timings and try to decode (to binary) the information you see.

Because this is a very non-trivial task without any clues, we will get you started using the following hints : the protocol is human 'decodable' and contains 1 start and 2 stop bits for each 'protocol data unit'. Look for repetitions in the pattern and subtle differences. If you get stuck, ask the teaching team for further hints. A second hint is that you have to limit the number of channels in the decoding software but record for a long duration of time.

This is the 'micro' analysis level. Now try to figure out, using the information you obtained, what is going on on the macro level. This is actually a very simplified version of a well-known piece of hardware. Discuss how this works and why it is/was used.