

Masterarbeit

Elena Schmidt

Absicherung von ST-Controllern gegen Produktpiraterie

Elena Schmidt

Absicherung von ST-Controllern gegen Produktpiraterie

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Masterstudiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Heike Neumann

Zweitgutachter: Prof. Dr.-Ing. Hans-Dieter Schütte

Eingereicht am: 17. September 2021

Elena Schmidt

Thema der Arbeit

Absicherung von ST-Controllern gegen Produktpiraterie

Stichworte

Produktpiraterie, ST-Mikrocontroller, Security

Kurzzusammenfassung

In dieser Arbeit werden die Möglichkeiten von ST-Microcontrollern zum Schutz gegen Produktpiraterie untersucht. Dafür werden zunächst mögliche Angriffe untersucht. Basierend auf diesen Angriffen werden Schutzmöglichkeiten erläutert. Zum Schluss werden auf zwei Mikrocontroller die Schutzfunktionen umgesetzt und auf ihre Funktionalität untersucht.

Elena Schmidt

Title of Thesis

Protecion of ST microcontroller against product piracy

Keywords

product piracy, ST microcontroller, security

Abstract

In this thesis explores the different possibilities of using ST-microcontrollers to prevent product piracy. First, different ways of exploits are explored. Based on these exploits the thesis examines the different manners of protecting against them. Finally, these protection mechanisms are implemented on two microcontrollers and analysed the effect they have.

Inhaltsverzeichnis

Abbildungsverzeichnis	vii
Tabellenverzeichnis	ix
Abkürzungen	x
Listings	xii
1 Einleitung	1
2 Grundlagen	2
2.1 Security	2
2.2 Angriffsarten	3
2.3 Kryptografie	4
2.3.1 Symmetrische Verschlüsselungen	4
2.3.2 Asymmetrische Verschlüsselungen	4
2.3.3 Digitale Signaturen	5
2.3.4 Hash-Funktionen	6
2.3.5 Secret Sharing	6
3 Konzept	8
4 Angriffe auf Mikrocontroller	10
4.1 Software Angriffe	10
4.2 Nicht-Invasive Angriffe	11
4.3 Semi-Invasive Angriffe	12
4.3.1 Glitch-Angriffe / Fehlerinjektion	12
4.4 Invasive Angriffe	15

5	Schutzmaßnahmen für Mikrocontroller	17
5.1	Schutzmaßnahmen unabhängig vom Mikrocontroller	17
5.1.1	Arten von Ausleseschutzmechanismen	17
5.1.2	Schutz vor Software Angriffen	20
5.1.3	Schutz vor Fehlerinjektion und Glitch-Angriffen	23
5.1.4	Authentizitätsprüfungen	26
5.1.5	Sicherheitsmaßnahmen für Verschlüsselungen	28
5.1.6	Hardware Schutzmaßnahmen	29
5.2	Security Funktionen bei ST-Controllern	31
5.2.1	Readout Protection (RDP)	31
5.2.2	Weitere Mechanismen zum Schutz des Speichers	32
5.2.3	Umgebungsüberwachung	33
5.3	Kombination von Schutzmaßnahmen	34
5.3.1	Root und Chain of Trust	34
5.3.2	Secure Boot	34
5.3.3	Abhängigkeiten von Schutzfunktionen	35
6	Untersuchung von Schutzmaßnahmen auf ausgewählten Mikrocontrollern	38
6.1	Voraussetzungen der Untersuchung	38
6.2	Untersuchung des STM32F439ZI	39
6.2.1	Grundfunktionen des Mikrocontrollers	39
6.2.2	Funktionsumsetzung	39
6.2.3	Einfügen einer Device ID Prüfung	44
6.2.4	Untersuchung des Secure Boot	46
6.2.5	Bewertung des Secure Boot	53
6.3	Untersuchung des STM32L552ZE	56
6.3.1	Grundfunktionalität des Mikrocontrollers	56
6.3.2	Funktionsumsetzung	56
6.3.3	Untersuchung des Secure Boot	57
6.3.4	Bewertung des Secure Boot	58
6.4	Angriff auf die Readout Protection	59
6.4.1	Beschreibung der Angriffe	59
6.4.2	Versuchsdurchführung	61
6.5	Auswertung	61

7	Ausblick und Fazit	63
7.1	Ausblick	63
7.1.1	Ausblick auf weitere Schutzmöglichkeiten	63
7.1.2	Ausblick auf weitere Angriffe	64
7.2	Fazit	64
	Literaturverzeichnis	65
A	Anhang	69
A.1	Python Skript zum Einfügen der Device ID in eine Elf-Datei	69
A.2	CD Ordnerstruktur	76
	Selbstständigkeitserklärung	77

Abbildungsverzeichnis

2.1	Symmetrische Verschlüsselung [26]	4
2.2	Asymmetrische Verschlüsselung [26]	5
2.3	Digitale Signatur [26]	5
2.4	Hash-Funktionen [26]	6
4.1	Grundaufbau einer Fehlerinjektion [32]	13
4.2	Glitch-Angriff auf den Systemtakt [12]	13
4.3	Spannungsglitch für verschiedene Mikrocontroller [4]	15
5.1	TrustZone [1]	22
5.2	Secure Bool [10]	24
5.3	Redundante Abfrage [10]	25
5.4	Zufällige Verzögerung [16]	25
5.5	zufällige Ausführungsreihenfolge [16]	26
5.6	Wire Mesh [30]	30
5.7	Funktionsprinzip Secure Boot [27]	34
5.8	Abhängigkeiten der Schutzfunktionen bei einem STM32F4	36
5.9	Abhängigkeiten der Schutzfunktionen bei einem STM32L5	37
6.1	STM32F4 - vereinfachter Ablauf des Secure Boot	40
6.2	Speicheraufteilung STM32F4	40
6.3	Berechnung der Signatur [27]	41
6.4	Secure Firmware Update [27]	42
6.5	Aktivierte Schutzfunktionen [25]	43
6.6	STM32F4 - SBSFU Menü	47
6.7	STM32F4 - SBSFU korruptes Firmware Image	49
6.8	STM32F4 - SBSFU WRP Test	50
6.9	STM32F4 - SBSFU MPU Test	51
6.10	STM32F4 - SBSFU Tamper Pin Test	52

6.11 STM32F4 - SBSFU IWDG Test	53
6.12 STM32F4 - SBSFU Sichere Variablen	54
6.13 STM32L5 - Aufbau TF-M [28]	57
6.14 STM32L5 - TF-M Menü	58
6.15 Aufbau Readout Protection Angriff	61
A.1 DVD-Struktur	76

Tabellenverzeichnis

6.1	STM32F4 - Boot Zeiten	53
-----	---------------------------------	----

Abkürzungen

AES Advanced Encryption Standard.

CRC Cyclic Redundancy Check.

CSS Clock Security System.

DMA Direct Memory Access.

ECC Elliptic Curve Cryptography.

HDP Secure Hide Protection.

MPU Memory Protection Unit.

OTP One Time Programmable.

PCROP Proprietary Code Read Out Protection.

PUF Physical Unclonable Function.

PVD Programmable Voltage Detector.

RDP Readout Protection.

RSA Rivest–Shamir–Adleman.

ST STMicroelectronics.

TAMP Anti-Tamper-Pins.

WRP Flash Write Protection.

XOM Execute-Only-Memory.

Listings

6.1	STM32F4 - Angepasstes Mapping für die Firmware	42
6.2	Aufteilung der Device ID	44
6.3	Redundante Abfrage	45
6.4	Secure Bool	45
A.1	Python Skript zum Einfügen der Device ID in eine Elf-Datei	69

1 Einleitung

In immer mehr Geräten werden Mikrocontroller eingebaut. Auch durch das Internet-of-Things sind immer mehr Geräte miteinander verknüpft und der Zugriff auf diese Geräte wird immer einfacher.

Ein großer des Wertes eines Gerätes besteht daher in der Software des Gerätes. Für Produktpiraten ist es äußerst attraktiv die Hardware von Geräten (z.B. Steuerungen) nachzubauen. Die Software wird einfach aus dem Originalgerät ausgelesen und in das gefälschte Gerät eingespielt.

Die Thesis wird bei mit der Firma Reese+Thies Industrieelektronik GmbH erstellt. Die Firma entwickelt Steuerungen, in den Mikrocontroller der Firma STMicroelectronics (ST) verbaut sind. Daher ist das Ziel der Arbeit, herauszufinden, welche Maßnahmen gegen Produktpiraterie bei STM32-Mikrocontroller anwendbar sind.

Sehr viele Steuergeräte verwenden Mikrocontroller der STM32F4-Reihe. Daher soll ein besonderer Fokus auf die Möglichkeiten dieser Controllerreihe gelegt werden. Des Weiteren bietet die STM32-Reihe bereits einige eingebaute Security-Features zum Schutz der abgelegten Software. Diese Features sollen genauer untersucht werden. Die Möglichkeiten anderer STM32-Controller sollen mit denen der F4-Controller verglichen werden.

Diese Gegenmaßnahmen sollen dann auf einem Controller beispielhaft umgesetzt und auf ihre Funktionalität überprüft werden. Wenn möglich sollen auch einige einfache Angriffe durchgeführt werden.

2 Grundlagen

2.1 Security

Security bezeichnet die Fähigkeit eines Systems, sich gegen Angriffe zu verteidigen. Dabei sollen folgende drei Ziele zu erreicht werden:

- Vertraulichkeit (Confidentiality)
- Integrität (Integrity)
- Authentizität (Authenticity)

Unter **Vertraulichkeit** versteht man, dass Daten nicht unberechtigt gelesen werden können. Im Falle von Mikrocontrollern kann sich das zum einen auf geheime Daten (z.B. Schlüssel) oder auf die gesamte Firmware beziehen.

Integrität umfasst die Korrektheit von Daten. Hier soll sichergestellt werden, dass diese nicht unbemerkt in irgendeiner Form verändert werden.

Authentizität in der Kryptografie unterscheidet zwischen Teilnehmerauthentizität und Nachrichtenauthentizität. Bei der Teilnehmerauthentizität wird die Identität der Teilnehmer zweifelsfrei nachgewiesen. Bei der Nachrichtenauthentizität kann der Empfänger einer Nachricht ihren Ursprung überprüfen. Im Falle von Mikrocontrollern lässt sich die Authentizität von der Hard- und Software betrachten.

Andere Security-Ziele (z.B. Verfügbarkeit) spielen in dieser Thesis keine Rolle. Daher werden diese hier nicht näher erläutert.

2.2 Angriffsarten

Im Allgemeinen werden Angriffe in folgende Kategorien unterteilt:

- Software-Angriffe
- Nicht-invasive-Angriffe
- Invasive Angriffe
- Semi-invasive-Angriffe

Software Angriffe Diese Angriffe erfolgen auf Schwächen in der Software (z.B. Bugs, Schwächen in Protokollen, nicht vertrauenswürdiger Code). Diese Angriffe beinhalten auch Angriffe auf die Kommunikation zwischen Geräten. In Abhängigkeit von den Schwächen können die Kosten eines solchen Angriffs sehr gering bis hoch sein.

Nicht-invasive Angriffe Diese Angriffe erfolgen auf der Chip/Board-Ebene ohne diesen zu beschädigen. Angriffspunkte sind dabei Schnittstellen, wie der Debugger. Andere Angriffsmöglichkeiten sind Seitenkanalangriffe, die die Umgebung (z.B. Stromverbrauch) überwachen und analysieren.

Invasive Angriffe Diese Angriffe erfolgen auf der Siliziumebene. Hierbei wird die Hardware direkt manipuliert. Bei einem solchen Angriff wird der Chip beschädigt. Ziel dieser Angriffe ist es, mehr über die Hardware herauszufinden oder Geheimnisse zu enthüllen. Diese Angriffe sind sehr teuer und benötigen entsprechende Ausrüstung. Sie führen zur Beeinträchtigung der Funktion beziehungsweise zur Zerstörung des Gerätes. Für invasive Angriffe werden oft mehrere Geräte benötigt, um diese komplett zu analysieren.

Semi-invasive Angriffe Bei diesen Angriffen wird zwar der Chip geöffnet, jedoch wird dieser nicht funktionsunfähig beziehungsweise kaum beschädigt. Hierbei wird z.B. Licht benutzt, um Fehler in das System einzufügen. Im Vergleich zu den nicht-invasiven sind semi-invasive Angriffe teurer, da der Zugriff auf den Chip notwendig ist. Im Vergleich zu den invasiven Angriffen sind diese allerdings um einiges günstiger.

Konkrete Angriffe auf Mikrocontroller werden im Laufe der Thesis weiter erläutert.

2.3 Kryptografie

In diesem Abschnitt werden Grundbegriffe der Kryptografie erklärt. Auf die genaue Mathematik wird nicht näher eingegangen, da diese für die Thesis nicht näher interessant ist. Sollte ein genaueres Verständnis gewünscht sein, können die genauen Verfahren in der Quelle [6] nachgelesen werden.

2.3.1 Symmetrische Verschlüsselungen

Bei symmetrischen Verschlüsselungen handelt es sich um die 'klassischen' Verschlüsselungen. Bei diesem Verfahren müssen sich die beiden Parteien auf einen gemeinsamen, geheimen Schlüssel einigen. Die Ver- und Entschlüsselung wird mit dem gleichen Schlüssel durchgeführt (vgl. Abbildung 2.1). Ein häufig Vorkommendes Verfahren ist der Advanced Encryption Standard (AES)-Algorithmus.

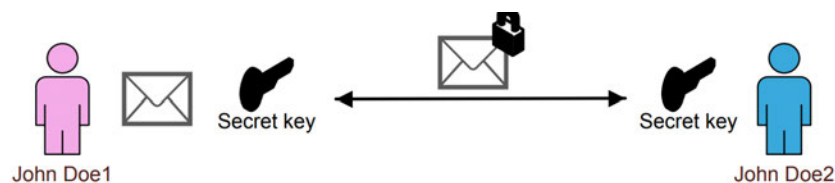
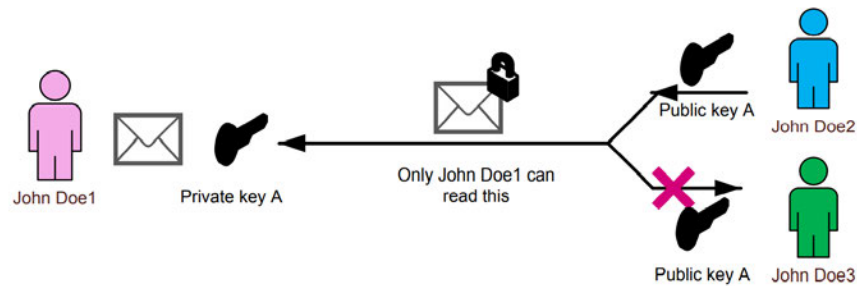


Abbildung 2.1: Symmetrische Verschlüsselung [26]

Ein großer Nachteil dieses Verfahrens ist, dass der Schlüssel zwischen den beiden Parteien ausgetauscht werden muss.

2.3.2 Asymmetrische Verschlüsselungen

Bei asymmetrischen Verschlüsselungen werden im Gegensatz zu symmetrischen Verschlüsselungen zwei Schlüssel verwendet, ein privater und ein öffentlicher Schlüssel. Dabei kann aus dem öffentlichen Schlüssel der private Schlüssel nicht mit vertretbarem Aufwand berechnet werden.



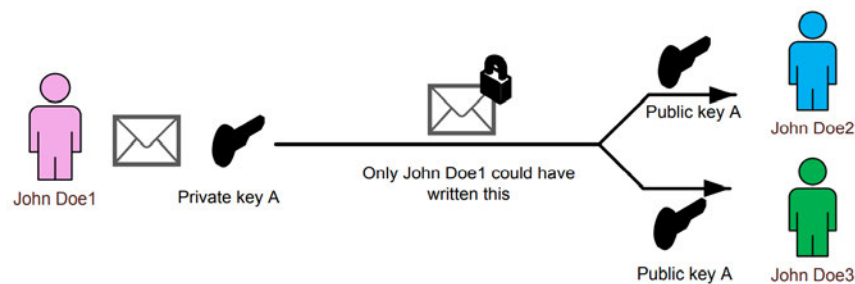
Abbildungung 2.2: Asymmetrische Verschlüsselung [26]

Eine Nachricht, die mit dem öffentlichen Schlüssel verschlüsselt wurde, kann nur mit dem privaten Schlüssel wieder entschlüsselt werden. Bekannte Verfahren sind der Rivest–Shamir–Adleman (RSA) und der Elliptic Curve Cryptography (ECC).

Der Nachteil einer asymmetrischen Verschlüsselung ist, dass die Berechnung komplexer ist als bei einem symmetrischen Verfahren.

2.3.3 Digitale Signaturen

Die Idee hinter digitalen Signaturen basiert auf der asymmetrischen Verschlüsselung. Hier wird mit dem privaten Schlüssel eine Nachricht signiert. Mit Hilfe des öffentlichen Schlüssels kann die Signatur verifiziert werden (vgl. Abbildung 2.3). Der öffentliche Schlüssel darf dabei dort, wo er hinterlegt ist, nicht veränderbar sein.



Abbildungung 2.3: Digitale Signatur [26]

2.3.4 Hash-Funktionen

Ziel der Hash-Funktionen ist es, die Integrität von Daten zu gewährleisten. Bei Hash-Funktionen handelt es sich dementsprechend um Kompressionsfunktionen. Sie generieren aus einem Bitstream beliebiger Länge einen Bitstream von fester Länge. Aus dem entstandenen Bitstream ist es nicht möglich den ursprünglichen Bitstream wiederherzustellen.

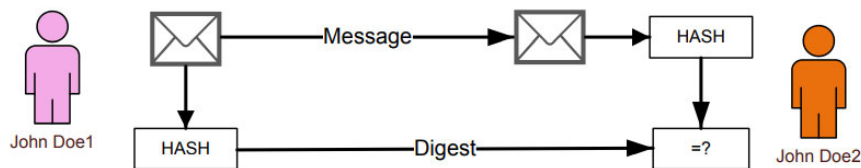


Abbildung 2.4: Hash-Funktionen [26]

Der Unterschied zu einem klassischen Cyclic Redundancy Check (CRC)-Verfahren ist, dass diese Funktionen kollisionsresistenter sind. Das bedeutet es ist praktisch unmöglich zwei Bitstreams zu finden, die den gleichen Hash ergeben.

Ein typischer Hash-Algorithmus ist der SHA-2.

2.3.5 Secret Sharing

Bei Secret Sharing geht es darum, ein Geheimnis unter verschiedenen Personen aufzuteilen. Aus einem Teil kann das Geheimnis nicht rekonstruiert werden. Es gibt im Allgemeinen zwei Varianten:

1. Alle Teile des Geheimnisses sind erforderlich: (n,n) -Verfahren
2. Nur eine Mindestanzahl der Geheimnisteile ist erforderlich:
 (k,n) -Schwellwert-Verfahren

Eine einfache Variante des Secret Sharing kann durch Summenbildung erzeugt werden. Das Geheimnis S ist dabei die Summe seiner Teile modulo m . Bei Bitstrings wird mit XOR gearbeitet werden.

Um ein Geheimnis S (mit Hilfe von XOR) auf n Personen zu verteilen, muss wie folgt vorgegangen werden:

- Es werden $n - 1$ zufällige Zahlen $S_1, \dots, S_{n-1}$ gewählt.
- Daraus ergibt sich $S_n = S \oplus (S_1 \oplus \dots \oplus S_{n-1})$
- Um S zu rekonstruieren, wird folgende Berechnung durchgeführt:
$$S = S_1 \oplus \dots \oplus S_{n-1} \oplus S_n$$

Es gibt noch weitere Secret-Sharing-Verfahren (z.B. Secret Sharing nach Shamir), die zum Beispiel erlauben Schwellwertverfahren umzusetzen. (vgl. [22]) Diese Verfahren sind für diese Thesis jedoch nicht relevant.

3 Konzept

In dieser Thesis soll untersucht werden, welche Maßnahmen ergriffen werden können, um einen Mikrocontroller von ST vor Produktpiraterie zu schützen. Diese Fragestellung durch dieses Konzept aufgegriffen werden.

Sammeln von Angriffspunkten Um herauszufinden, welche Gegenmaßnahmen gegen Produktpiraterie umzusetzen sind, werden zunächst die Angriffsmöglichkeiten auf einen Mikrocontroller untersucht. Dabei wird herausgestellt, welche Ziele die Angriffe verfolgen und wo die entsprechenden Angriffspunkte sind. Die Aufwendigkeit und die Kosten der Angriffe werden ebenfalls in die Betrachtung mit einbezogen. Darauf aufbauend wird herausgestellt, welche dieser Angriffe wahrscheinlicher durchgeführt werden.

Sammeln von Schutzmaßnahmen Auf Basis der ermittelten Angriffe werden die möglichen Schutzmaßnahmen gegen diese genauer betrachtet. Die Maßnahmen werden einem Bereich (Vertraulichkeit, Integrität, Authentizität) zugeordnet und deren jeweiligen Stärken und Schwächen herausgearbeitet. Schutzmaßnahmen, die nur oder fast nur in der Software umgesetzt werden können, sind besonders interessant. Da das Ziel der Thesis Maßnahmen für ST-Controller beinhaltet, werden die Schutzfunktionen der ST-Mikrocontroller genauer betrachtet.

Untersuchung von Mikrocontrollern Zum Schluss der Untersuchung werden einige der Schutzmaßnahmen auf zwei ausgewählten Mikrocontrollern umgesetzt und genauer untersucht. Diese Mikrocontroller sollen der STM32-Mikrocontrollerreihe angehören. Einer der Controller soll der F4-Reihe angehören, da diese von der Firma oft verwendet wird. Der zweite Controller soll mehr eingebaute Schutzmechanismen besitzen.

Vor der Untersuchung wird festgelegt, welche Angriffe erwartet werden. In Abhängigkeit des Angreifers werden Schutzmaßnahmen ausgewählt. Die Untersuchung beinhaltet, wie

die Schutzfunktionen aufeinander aufbauen. Neben den Tests der Grundfunktionalität sollen einige einfache Angriffe durchgeführt werden.

4 Angriffe auf Mikrocontroller

Dieses Kapitel beschreibt ausführlich, welche Angriffsmöglichkeiten auf einen Mikrocontroller bestehen.

4.1 Software Angriffe

Die Angriffe erfolgen auf die Schwächen in der Software (z.B. Bugs, Schwächen in Protokollen, ausführen von nicht vertrauenswürdigen Code). Diese Angriffe beinhalten auch Angriffe auf die Kommunikation zwischen Geräten. In Abhängigkeit von den Schwächen können die Kosten eines solchen Angriffs sehr gering bis hoch sein.

Software-Angriffe werden durchgeführt, indem man den Controller dazu bringt, Schadcode durchzuführen. Dieser übernimmt dann entweder die Kontrolle über das Gerät oder liest Systeminformationen aus (ID, RAM, Flash, etc.). Der Schadcode kann dabei entweder eingefügt werden oder bereits vorhanden sein (z.B. durch das Nutzen von nicht vertrauenswürdigen Bibliotheken).

Das Einfügen von Schadcode kann durch verschiedene Methoden erfolgen:

- durch offene Türen im Mikrocontroller
 - offene Ports: Debug Port (JTAG und SWD)
 - Bootloader, falls dieser erreichbar ist
- durch Updates
- durch Schwächen in den Kommunikationsschnittstellen oder Nutzung von Bugs
 - Ausführung von Daten
 - Nutzung von Buffer-Überläufen (Buffer Overflow Attack)

- Nutzung nicht vertrauenswürdiger Bibliotheken

Eine offene Debug-Schnittstelle kann nicht nur genutzt werden, um Schadcode einzufügen, sondern auch um die Firmware einfach auszulesen und auf ein gefälschtes Geräte zu kopieren.

Der Überlauf von Buffern kann ebenfalls genutzt werden, um Fehler im System auszulösen.

4.2 Nicht-Invasive Angriffe

Nicht-invasive Angriffe erfolgen auf der Chip/Board-Ebene ohne diesen zu beschädigen. Die offensichtlichen Angriffsstellen sind dabei die Schnittstellen (z.B. der Debugger), sofern diese nicht deaktiviert wurden. Ungeschützte Schnittstellen werden genutzt, um sensible Informationen (z.B. den Inhalt des Flashs) zu auszulesen.

Eine nicht-invasive Angriffsmöglichkeit sind Seitenkanalangriffe. Diese überwachen die Umgebung des Systems wie z.B. die Laufzeit, den Stromverbrauch oder die elektromagnetische Emission eines Gerätes. Durch diese Informationen können entweder Rückschlüsse gezogen oder direkt geheime Informationen ausgelesen werden. Seitenkanalangriffe werden normalerweise dazu genutzt, kryptografische Schlüssel auszulesen. Sie können aber auch verwendet werden, um Informationen für Glitch-Angriffe zu erspähen (vgl. Kapitel 4.3.1).

Timing-Angriff Bei einem Angriff auf das Timing eines Mikrocontrollers wird versucht, durch das Messen von Ausführungszeiten Informationen zu erlangen. Viele kryptografische Operationen haben ein unterschiedliches Zeitverhalten, das von dem Schlüssel abhängt. So werden oft bei einer '1' im Schlüssel mehr mathematische Operationen durchgeführt. Somit dauert die Verarbeitung von diesem Schlüsselbit länger als bei einer '0'.

Power Analyse Die Grundidee eine Power Analyse ähnelt dem des Timing-Angriffes. Auch hier wird die Abhängigkeit des Schlüssels mit der dem Stromverbrauch verglichen. Ein bekanntes Beispiel dafür ist der Angriff auf den Square-and-Multiply-Algorithmus zur Berechnung des RSA: In einer 'einfachen' Implementierung kann der Schlüssel im

Stromverlauf erkannt werden. Das liegt daran, dass eine '1' mehr Strom verbraucht als eine '0'. Eine solche Analyse kann auch bei Laden von Daten in ein Register beobachtet werden.

Elektromagnetische Abstrahlung Viele Hersteller achten bereits darauf, den Stromverbrauch bei kryptografischen Anwendungen gleich zu halten. In diesem Fall lassen sich trotzdem noch ähnliche Angriffe durchführen. Anstelle des Stromverbrauches kann die elektromagnetische Abstrahlung eines Chips gemessen werden.

Die Kosten nicht-invasiver Angriffe sind relativ gering, je nachdem welcher Seitenkanal ausgelesen wird.

4.3 Semi-Invasive Angriffe

Neben invasiven und nicht-invasiven Angriffen gibt es noch die semi-invasiven Angriffe. Bei diesen wird zwar der Chip geöffnet, aber er wird dabei kaum beschädigt. Nach dem Angriff bleibt der Chip funktionsfähig.

4.3.1 Glitch-Angriffe / Fehlerinjektion

Ziel von Glitch-Angriffen ist es, Fehler in das System einzuführen. Diese Fehler sollen dann z.B. kritische Abfragen von Security-Mechanismen überspringen. Einige Angriffe sorgen auch direkt dafür, dass Security Funktionen deaktiviert werden (vgl. Paragraph "Bekannte Sicherheitslücken" in Kapitel 5.2.1). Fehlerangriffe werden auch verwendet, um Fehler in Verschlüsselungen einzubauen.

Methoden der Fehlerinjektion Es gibt verschiedene Möglichkeiten die Umgebung eines Mikrocontrollers zu manipulieren und damit Fehler auszulösen (vgl. [2]). Abbildung 4.1 zeigt den grundlegenden Aufbau eines Glitch-Angriffes.

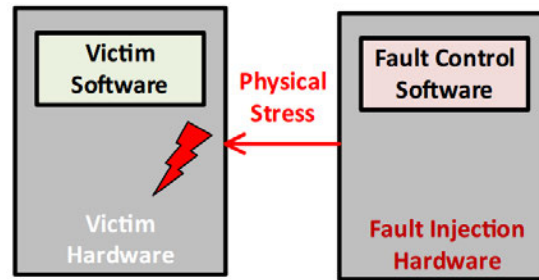


Abbildung 4.1: Grundaufbau einer Fehlerinjektion [32]

Zu den einfachsten Zugriffspunkten gehört die Versorgungsspannung. Dabei wird diese für eine kurze Zeit in einen Bereich gebracht, in dem der Mikrocontroller nicht mehr richtig funktioniert. Dabei kann die Spannung sowohl zu hoch als auch zu niedrig eingestellt werden. Die Manipulation der Spannung kann dafür sorgen, dass Instruktionen von dem Prozessor entweder falsch interpretiert oder gar übersprungen werden. Die Verringerung der Spannung führt dazu, dass die Gatterlaufzeiten größer werden. Dadurch können Instruktionen nicht schnell genug verarbeitet werden.

Ein weiterer einfacher Angriffspunkt besteht in einem externen Systemtakt. Hier werden z.B. zusätzliche Flanken eingefügt. Ein Glitch im Systemtakt kann dazu führen, dass Daten falsch ausgelesen werden oder Instruktionen übersprungen werden. (Der Prozessor fängt an die Instruktion $n+1$ durchzuführen, bevor die Instruktion n fertig ist.) Abbildung 4.2 zeigt ein Beispiel für einen Glitch im Systemtakt, bei dem eine weitere Flanke eingefügt wurde.

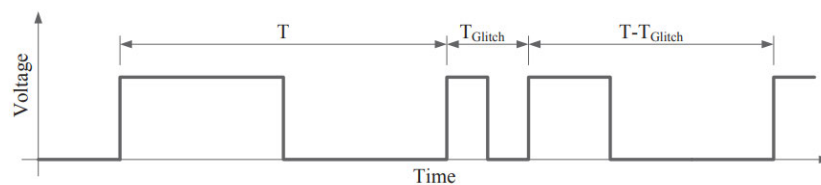


Abbildung 4.2: Glitch-Angriff auf den Systemtakt [12]

Eine weitere Möglichkeit, Fehler einzufügen, ist die Manipulation der Umgebungstemperatur. In den Datenblättern sind Temperaturbereiche angegeben, in denen die Funktionalität des Mikrocontrollers gesichert ist. Bei der Manipulation der Umgebungstemperatur

wird diese an diese Grenzen dieser Temperaturbereiche gebracht. Dabei können zwei Effekte auftreten: Zum einen kann der Inhalt des Speichers verändert werden. Zum anderen können Lese- und Schreibprozesse manipuliert werden. Laut [2] haben Lese- und Schreibprozesse unterschiedliche Temperaturen, bei denen sie noch funktionieren. Durch die Manipulation der Temperatur kann dafür gesorgt werden, dass ein Leseprozess noch durchführbar während ein Schreibprozess scheitert (oder umgekehrt).

Elektrische Schaltungen sind grundsätzlich empfindlich gegenüber Licht. So kann weißes Licht oder Laser verwendet werden, um Fehler auszulösen. Ein Angriff mit Hilfe eines Laser kann z.B. Bits im Speicher ändern und somit Security-Mechanismen deaktivieren (vgl. Kapitel 6.4). Das gleiche kann durch Röntgen- oder Ionenstrahlen erfolgen. Bei diesen Angriffen muss allerdings das Mikrocontrollergehäuse entpackt werden, um den Speicher freizulegen.

Die Effektivität von Glitch-Angriffen kann weiter erhöht werden, indem verschiedene Manipulationsarten (z.B. Takt und Spannung) kombiniert werden.

Herausforderungen von Glitch-Angriffen Glitch-Angriffe sind in der Theorie einfach auszulösen. In der Praxis ist es jedoch aufwendig, einen solchen Angriff erfolgreich durchzuführen.

Zum einen ist es wichtig den richtigen Zeitpunkt eines Glitches zu finden. Oft wird der Mikrocontroller zunächst mit Hilfe von Seitenkanalangriffen untersucht, den Zeitpunkt zu finden, bei dem kritische Abfragen durchgeführt werden. [11] findet mit Hilfe des Stromverlaufes den Zeitpunkt in dem der Ausleseschutz konfiguriert wird. Durch das Durchführen eines Spannungsglitches deaktiviert er den Ausleseschutz. Im Idealfall lässt sich mit Hilfe einer Seitenkanalanalyse ein Triggersignal finden, der einen Angriff automatisch auslöst.

Ein weiteres Problem stellt die Form des Glitches dar. In [4] wird gezeigt, dass in Abhängig vom Mikrocontroller, verschiedene Formen von Spannungsverläufen notwendig sind, um einen Fehler auszulösen (Abbildung 4.3).

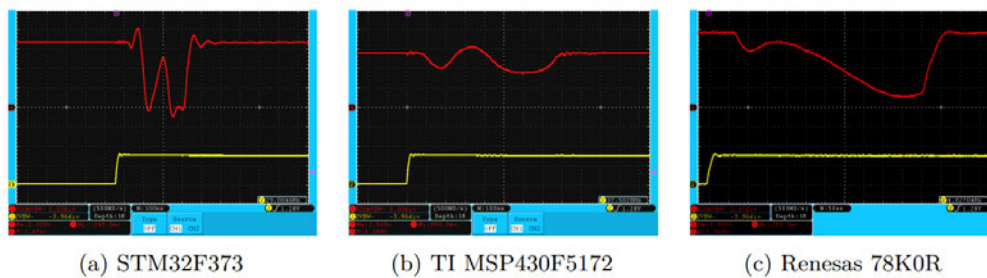


Abbildung 4.3: Spannungsglitch für verschiedene Mikrocontroller [4]

Sobald das richtige Timing und die richtige Glitchform gefunden wurde, ist ein solcher Angriff sehr einfach durchführbar.

Die Erfolgswahrscheinlichkeit für einen erfolgreichen Glitch sinkt, je mehr Fehler notwendig sind, um ein Fehlverhalten in einem System auszulösen. Insgesamt sind semi-invasive im Vergleich zu den nicht-invasiven etwas teurer, da der direkte Zugriff auf den Chip notwendig ist. Im Vergleich zu den invasiven Angriffen sind diese allerdings um einiges günstiger. Man kann davon ausgehen, dass semi-invasive Angriffe in Zukunft immer günstiger werden, da die notwendige Hardware günstiger und zugänglicher wird. Es tauchen auch immer wieder neue Geräte auf, die solche Angriffe erleichtern sollen (z.B. der ChipWhisperer).

4.4 Invasive Angriffe

Invasive Angriffe erfolgen auf Siliziumebene. Hierbei wird die Hardware direkt manipuliert. Bei einem solchen Angriff wird der Chip beschädigt. Ziel ist es, mehr über die Hardware herauszufinden oder Geheimnisse zu enthüllen.

Zum Beispiel werden bei diesen Angriffen die oberen Schichten eines Chips weggeätzt. Damit bekommt der Angreifer Zugang zu den Speicherbereichen und kann Informationen auslesen oder ändern. Bei invasiven Angriffen handelt es sich oft um Reverse Engineering. Mangels öffentlich zugänglicher Informationen wird versucht, den Chipaufbau zu ermitteln, also wo sich CPU, Speicherbereiche und kryptografische Co-Prozessoren befinden. Die Informationen werden mit einem Elektronenrastermikroskop oder durch das Proben von Bussen und Speichern (Microprobing) ausgespäht.

Beim Microprobing werden grundsätzlich folgende Schritte durchgeführt: Bei der Entkapselung wird das Gehäuse des Mikrocontrollers mit Hilfe von Säure entfernt. Dabei kann entweder die Vorder- oder Rückseite eines Chipsets freigeräumt werden. Bei dem Reverse Engineering werden versucht Design-Informationen zu extrahieren. Auf Basis dieser Informationen werden die Leitungen, die angegriffen werden sollen, lokalisiert. Im letzten Schritt werden die Leitungen ausgespäht werden (vgl. [30]).

Invasive Angriffe sind sehr teuer und benötigen entsprechende Ausrüstung. Sie führen zur Beeinträchtigung der Funktion beziehungsweise zur Zerstörung des Gerätes. Für invasive Angriffe werden oft mehrere Geräte benötigt, um es komplett zu analysieren. Auch ist viel technisches Know-how und Erfahrung notwendig, um solche Angriffe durchzuführen. Aus diesem Grund sind invasive Angriffe auch eher selten.

5 Schutzmaßnahmen für Mikrocontroller

Das vorherige Kapitel hat gezeigt, dass es zahlreiche Möglichkeiten gibt, Mikrocontroller anzugreifen und Informationen zu stehlen. Dieses Kapitel befasst sich mit den möglichen Schutzmaßnahmen für Mikrocontroller. Hierbei sind insbesondere Maßnahmen interessant, die sich in Software beziehungsweise ohne zusätzliche Hardware umsetzen lassen. Der erste Teil des Kapitels behandelt Schutzmaßnahmen die nicht von Mikrocontrollern abhängig sind. Der zweite Teil beschäftigt sich mit Schutzfunktionen von ST-Controllern. Der dritte Teil zeigt, die Schutzmaßnahmen kombiniert werden können.

5.1 Schutzmaßnahmen unabhängig vom Mikrocontroller

5.1.1 Arten von Ausleseschutzmechanismen

Einer der Hauptangriffsziele bei der Produktpiraterie ist die Software des Mikrocontrollers. Diese befindet sich in der Regel auf dem Flash-Speicher. Daher ist der Zugriff auf diesen für einen Angreifer besonders interessant. Daher ist eines der Ziele zum Schutz vor Produktpiraterie den Mikrocontroller vor dem Auslesen des Flash-Speichers durch den Debugger zu verhindern. Dadurch soll die Vertraulichkeit und Integrität von diesem gesichert werden.

Die einfachste Variante, um Flash vor einem Zugriff durch den Debugger zu schützen, ist die Umkonfigurierung der Debug-Pins beim Programmstart. Auf den ersten Blick verhindert dies einen Zugriff von außen. Aus Security-Sicht ist dies jedoch nicht ausreichend, da ein Controller einfach im Reset-Zustand gehalten werden kann. In diesem Zustand ist die Umkonfigurierung noch nicht erfolgt und daher kann das Programm aus dem Flash-Speicher ausgelesen werden.

Readout Protection (RDP) Viele Mikrocontroller besitzen Mechanismen, die den Zugriff auf den Speicher verhindern sollen. Diese werden in der Regel mit Hilfe von Option Bytes aktiviert. Es gibt verschiedene Arten von sogenannten "Readout Protections", die in unterschiedlichen Varianten den Speicher von außen unlesbar (und oft auch unschreibbar) machen.

Bei einer Variation einer solchen Debug-Sperre wird nur der Flash-Speicher unlesbar gemacht. Der Zugriff auf dem RAM und die Peripherien bleibt bestehen. Ein Beispiel dafür ist die Level 1 Readout Protection von STM32-Microcontrollern (vgl. Kapitel 5.2.1). Der Vorteil eines solchen Ausleseschutzes ist, dass Debuggen oft trotzdem möglich ist, indem die Sperre wieder deaktiviert wird. Die Entwicklung und das Durchführen von Updates ist dadurch weiterhin ohne zusätzlichen Aufwand möglich. Auch die Deaktivierung des Ausleseschutzes ist immer noch möglich, wobei hierbei der Inhalt des Flash-Speichers gelöscht wird. Dieser Vorteil bietet allerdings eine offene Tür für potentielle Angreifer: den RAM. Die hier zwischengespeicherten Daten können Angreifern erlauben, Rückschlüsse auf die Software zu ziehen. Ein Beispiel dafür zeigt der Angriff 'Cold-Boot Stepping' in [19]: In der Software wird zu Beginn der CRC über den Flash-Speicher berechnet. Die Zwischenergebnisse werden im RAM zwischengespeichert. Beim Anschluss des Debuggers wird die Software von der Readout Protection angehalten und der momentane Zustand des RAMs lässt sich auslesen. Dieser Umstand wird ausgenutzt, indem zu unterschiedlichen Zeitpunkten nach dem Systemstart der RAM ausgelesen wird. Daraus lassen sich dann die Zwischenergebnisse des CRC extrahieren. Aus diesen Zwischenergebnissen lässt sich der Inhalt des Flash-Speichers mit geringem Aufwand wieder zurückrechnen.

Mechanismen die den Flash- und RAM-Zugriff blockieren, allerdings die Prozessorregister offen lassen, bieten ähnliche offene Türen, indem sie Geheimnisse preisgeben oder z.B. den Zugriff auf den Direct Memory Access (DMA) gewähren, mit dem die Schutzmaßnahme ganz umgangen werden kann.

Den effektivsten Schutz bietet ein Ausleseschutz, der den Debugger endgültig deaktiviert. Hier werden viele der relativ einfachen Angriffsmöglichkeiten verhindert. Diese Maßnahme erhöht allerdings auch den Entwicklungsaufwand eines Gerätes. Updates müssen hier über alternative serielle Schnittstellen erfolgen. Diese Schnittstellen müssen ebenfalls geschützt werden, damit sie keine offene Türen für Angreifer bietet (vgl. Kapitel 5.1.2 und 5.1.3).

Grundsätzlich gilt für alle Varianten der Readout Protection, dass sie den Angriff über einen Debugger erschweren. Sie bieten einen geringen Entwicklungsaufwand, da keine

zusätzliche Hardware notwendig ist. Auch entsteht keine zusätzliche Auslasten der Controllern in Form von höheren Rechenzeiten, etc. Allerdings bleiben je nach Implementierungsart trotzdem Sicherheitslücken. Daher sollte man sich nicht ausschließlich auf einen solchen Ausleseschutz verlassen.

Webseiten von Firmen, die sich darauf spezialisieren Mikrocontroller von fertigen Geräten auszulesen, listen zahlreiche Mikrocontrollerreihen auf, bei denen sie in der Lage waren diese Schutzmaßnahmen zu umgehen. In [23] weist Skorobogatov ebenfalls mit Hilfe von nicht-invasiven und invasiven Angriffen zahlreiche Sicherheitslücken im Ausleseschutz verschiedener Mikrocontrollern auf. Auch Fehler in der Hardware-Umsetzung solcher Ausleseschutzmechanismen (z.B. Hardware Race Conditions) bieten offene Türen für Angreifer. Ein Beispiel dafür ist der Debug Interface Exploit von [19]. Näheres dazu lässt sich in Kapitel 6.4 nachlesen.

Execute-Only-Memory (XOM) Eine weitere Form eines Ausleseschutzes ist die Festlegung von Execute-Only-Memory-Bereichen. Die Grundidee ist, dass man Speicherbereiche definieren kann, die sowohl von innen (innerhalb der Software) als auch von außen (Debugger) weder gelesen noch beschrieben werden können. Die restlichen Speicherbereichen können normal genutzt werden. Dadurch sollen die Teile des Codes geschützt werden, die zum Beispiel Security Mechanismen umsetzen oder den eigene Code vom Code von nicht vertrauenswürdigen Drittanbietern verborgen werden, ohne dabei die Debugging-Funktion zu verlieren. Execute-Only-Memory besitzt allerdings ein Problem in seinem Konzept selbst: Genau wie bei den vorher beschriebenen Readout Protections erlaubt der Debugger auch hier eine offene Hintertür. Schink und Obermaier zeigen in [21], wie mit Hilfe von Single-Stepping oder Interrupts und der Beobachtung des Systemzustands die Instruktionen im gesicherten Bereich wiederhergestellt werden können.

Diese Angriffsmöglichkeit besteht bei Mikrocontrollern mehrerer Firmen (z.B. ST, Texas Instruments und NXP). Die genaue Angriffsmöglichkeit hängt von dem Mikrocontroller ab. Zum Beispiel die Mikrocontrollerreihen STM32F7 und STM32H7 erlauben im Gegensatz zu STM32Lx und STM32F4 zum Beispiel kein Single-Stepping im geschützten Bereich. Dort müssen für den Angriff Interrupts (z.B. mit dem SysTick) verwendet werden. Allgemein ist es bei einem Angriff durch Interrupts schwerer Rückschlüsse auf die durchgeführten Instruktionen zu schließen, da in der Zeit zwischen den Interrupts mehrere Anweisungen verarbeitet werden konnten.

Dieser Angriff lässt sich verhindern, in dem der Debugger deaktiviert wird. Das entspricht allerdings nicht der Idee hinter dem Schutzmechanismus. Eine Alternative ist die Nutzung von Trusted Execution Environment (TEE) Architekturen (z.B. TrustZone von ARM) (vgl. Kapitel 5.1.2). Execute-Only-Memory-Bereiche können alternativ zusammen mit der Deaktivierung des Debuggers als Isolationsmöglichkeit für Security-Code verwendet werden, um diese vor internen Angriffen zu schützen.

Verschlüsselung des Flash Speichers Eine auf den ersten Blick offensichtliche Methode, den Speicher vor unbefugten Auslesen abzusichern, ist das Verschlüsseln von diesem. In der Theorie kann ein Angreifer den Flash-Speicher zwar auslesen, aber diesen nicht verstehen oder weiterverwenden.

In der Praxis birgt die Verschlüsselung des Speicher allerdings auch neue Probleme: Zum einen müssen die Schlüssel gesichert werden. Dafür wird ein sicherer Speicher in irgendeiner Form benötigt, z.B. ein Secure Element. Zum anderen muss gewährleistet werden, dass die entschlüsselte Firmware nicht ausgelesen werden kann. Dadurch entsteht ein wesentlich höherer Entwicklungsaufwand.

5.1.2 Schutz vor Software Angriffen

Wie in Kapitel 4.1 erwähnt, bergen Software-Angriffe eine große Gefahr für Mikrocontroller, da diese günstig sind und relativ einfach durchzuführen. Das Ziel beim Schutz vor Software Angriffen ist daher das Einfügen von Schadcode zu erschweren. Dafür müssen zum einen die Schnittstellen abgesichert werden, als auch Geheimnisse von dem Rest des Codes isoliert werden. Zum anderen ist ein Kernelement die Isolation der verschiedenen Speicherbereiche.

Absicherung serieller Schnittstellen Wie bereits in Kapitel 5.1.1 erwähnt, müssen auch andere serielle Schnittstellen abgesichert werden. Bei der Absicherung serieller Schnittstellen gibt es zwei Dinge zu beachten:

Zum einen müssen Bufferüberläufe verhindert werden. Die Größe der empfangenen daher stets geprüft werden. Bufferüberläufe werden von Angreifern genutzt, um Schadcode einzufügen oder Fehler im System auszulösen.

Zum anderen sollte der Kommunikationsstack von Security-Funktionen isoliert werden. Falls es doch zu einem Bufferüberlauf kommt, darf es nicht möglich sein, dass empfindlichen Daten gelesen oder überschrieben werden können.

Um einen Angriff zusätzlich zu erschweren, können die Schnittstellen im Endgerät versteckt werden. Das bedeutet allerdings auch, dass diese für den Entwickler und Nutzer schwer zugänglich werden.

Je nach Anwendung kann die Kommunikation verschlüsselt werden. Hier muss zusätzlich die Sicherheit der Schlüssel gewährleistet werden.

Der Schutz serieller Schnittstellen sollte grundsätzlich eingebaut werden. Nicht um Security zu gewährleisten, sondern auch um funktionelle Sicherheit zu garantieren.

Memory Protection Unit (MPU) Die Memory Protection Unit ist ein Bestandteil von Mikrocontrollern, mit dem die Zugriffsrechte der Speicher verwaltet werden können. Wie von Computern gewohnt, können damit dem Speicher die Attribute Execute/ Write/ Read zugewiesen werden. Werden die Zugriffsrechte verletzt, wird ein Interrupt ausgelöst und es kann entsprechend reagiert werden.

Neben den Zugriffsrechten kann die Ausführung des Codes in zwei Modi erfolgen: privilegiert und unprivilegiert. Die MPU gibt dadurch die Möglichkeit den Security Code vom Rest zu isolieren. Die MPU kann somit genutzt werden, um die Integrität und Vertraulichkeit von der Software zu gewährleisten.

Ein Nachteil der MPU ist die komplexe Nutzung. Es gibt bei der Programmierung viel zu beachten. Damit steigt der Programmieraufwand. Es ist zwar auch möglich eine solche Isolierung rein in Software möglich, bietet aber im Allgemeinen geringeren Schutz.

TrustZone Bei der TrustZone handelt es sich um eine Architektur bei Single-Core-Mikrocontrollern, bei der sämtliche Teile in sichere und unsichere Bereiche aufgeteilt werden. Die Trennung beinhaltet sowohl alle Speicher (Flash, SRAM und externe Speicher) als auch die Peripherie. Jeder Bereich besitzt zudem eine eigene MPU.

Die Software wird hier in zwei Teile aufgeteilt: sichere und unsichere Software. Die sichere Software wird in dem sicheren Adressbereich ausgeführt und kann auf alle Bereiche des Mikrocontrollers zugreifen. In diesem Bereich sollen die sicherheitsrelevante Funktionen (z.B. Secure Boot, Verschlüsselungen, Updates, etc.) durchgeführt werden. Nicht-sichere

Software wird in dem nicht-sicheren Adressbereich ausgeführt und kann nur auf den nicht-sicheren Speicher und Peripherie zugreifen. Hier befindet sich die 'traditionelle' Software. Beim Systemstart ist der Mikrocontroller zuerst im sicheren Zustand (vgl. Abbildung 5.1).

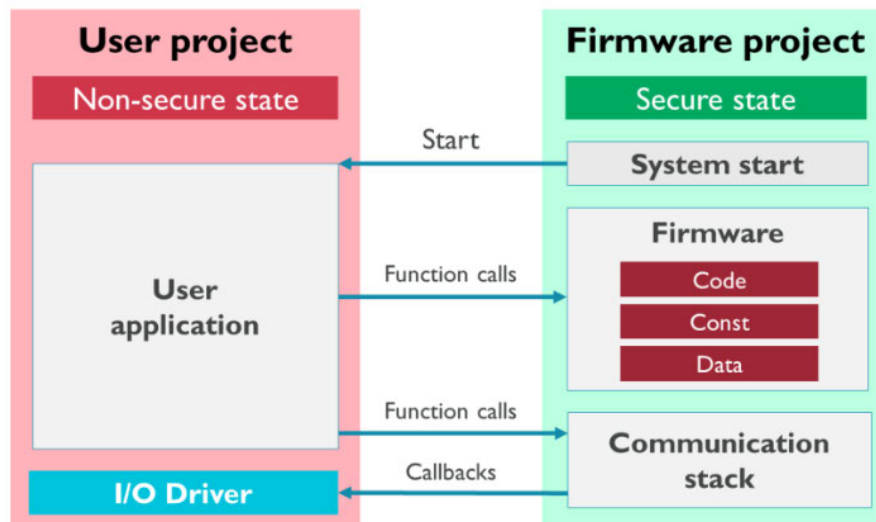


Abbildung 5.1: TrustZone [1]

Die Trennung der Adressen erfolgt durch den Prozessor mit Hilfe der Security Attribution Unit (SAU). Trotz der Trennung besteht die Möglichkeit, dass die Bereiche über Funktionsaufrufe miteinander interagieren können. Für diesen Zweck werden die Adressen in drei Bereiche eingeteilt: secure, non-secure und non-secure-callable. Letzteres ist ein Unterbereich des sicheren Bereichs. Diese können als vordefinierte Zugriffspunkte auf den sicheren Bereich für die nicht-sicheren Bereiche genutzt werden. Wird versucht unbefugt auf den sicheren Bereich zuzugreifen, wird ein Interrupt ausgelöst. Hier können dann entsprechende Reaktionen umgesetzt werden.

Sowohl der sichere als auch der unsichere Bereich besitzen beide eine eigene MPU. Diese erlaubt es, den Code weiter zu isolieren. Software Angriffe werden dadurch weiter erschwert. Die TrustZone gilt momentan bei Mikrocontrollern als State-of-the-Art-Lösung zur Isolierung der Software.

Angriffe auf die TrustZone Thomas Roth demonstrierte in [20] einen Angriff mit Hilfe von einem Spannungsglitch auf einen Microchip SAM L11. Dieser Chip verwendet

die TrustZone-Architektur. Mit dem Angriff war er in der Lage die Sicherheitsmaßnahmen der TrustZone zu umgehen.

Der Glitch wurde bei der Konfiguration der Implementation Defined Attribution Unit (IDAU) durchgeführt, die dafür zuständig ist, den Speicher in sichere und unsichere Bereiche aufzuteilen. Der Glitch sorgte dafür, dass die sicheren Speicherbereiche im Flash und RAM zugänglich waren. Der Angriff wurde auch für auf den NXP-Controller LPC55S69 durchgeführt. Hier waren zwei Glitches notwendig, um die Konfiguration zu stören. Wie angreifbar die TrustZone ist, hängt also vom konkreten Mikrocontroller ab.

Die Angriffe zeigen, dass die TrustZone zwar guten Schutz von innerhalb des Controllers bietet. Für Angriffen von außen (semi-invasive, invasive Angriffe) bleibt sie dennoch angreifbar.

5.1.3 Schutz vor Fehlerinjektion und Glitch-Angriffen

Umgebungsüberwachung Um Fehler in ein System zu injizieren, manipulieren Angreifer die Umgebung des Mikrocontroller. Dazu zählen die Spannung, Temperatur, das Magnetfeld und Licht. Diese sollten daher mittels Sensoren überwacht werden. Auch der Systemtakt sollte überwacht werden. Alternativ kann der interne Takt des Mikrocontrollers verwendet werden, da dieser schwerer zu erreichen ist.

Viele General Purpose Mikrocontroller bieten Möglichkeiten an, Spannung, Temperatur und Takt zu überwachen. Diese Überwachungsmethoden sind allerdings in der Regel nicht auf Glitch-Angriffe ausgelegt. Controller die für Security-Anwendungen ausgelegt sind, haben solche Sensoren bereits verbaut.

Fehlererkennungsverfahren im Speicher Viele Mikrocontroller bieten bereits verbaute Fehlererkennungsverfahren im Speicher (z.B. ECC) an. Dadurch werden einfachen Bitfehler automatisch korrigiert. Ein möglicher Nachteil dieser Verfahren ist, dass ein Angriff nicht erkannt wird (falls das gewünscht ist).

Fehlerresistente Programmierung Eine weitere Möglichkeit das Einfügen von Fehlern zu erschweren, ist fehlerresistente Programmierung. Dabei werden Abfragen und Variablen so gestaltet, dass Einfügen von Fehlern erschwert wird beziehungsweise erkannt wird.

Um sicher zu gehen, dass kein Code übersprungen wurde, können Variablen in jedem Programmzweig mit Primzahlen inkrementiert werden und der erwartete Wert geprüft werden. Jeder Programmzweig nutzt eine eigene Primzahl. (vgl. [31], S.15).

Eine weitere Maßnahme ist das Abhärten von Prüfvariablen. Werte wie 0 für *false* und *true* in den anderen Fällen sollten vermieden werden. (vgl. [10]) In solchen Fällen lässt sich ein *false* einfach in ein *true* ändern. Es muss dafür nur ein Bit gekippt werden. Dabei ist es egal, um welches Bit es sich handelt oder ob noch mehr Bits kippen. Besser ist es Enum-Werte zu definieren, die eine größere Hamming-Distanz haben. Je größer die Hamming Distanz, desto schwerer ist es, einen Fehler einzuführen. Es sollten allerdings nicht inverse Werte voneinander sein (z.B. 0xA5 und 0x5A), da diese einfach zu kippen sind. Ein Beispiel für ein sicheren Boolean zeigt die folgende Abbildung:

```
1 typedef enum {
2     SECURE_FALSE = 0xacefaecf,
3     SECURE_TRUE  = 0xca38c3e8
4 } secure_bool_t;
5
6 int check_flag(secure_bool_t critical_flag)
7 {
8     if (critical_flag == SECURE_TRUE) {
9         return RES_FLAG_TRUE;
10    }
11    return RES_FLAG_FALSE;
12 }
```

Abbildung 5.2: Secure Bool [10]

Um Glitch-Angriffe weiter zu erschweren, können unterschiedliche, redundante Überprüfungen in die Software eingefügt werden. Für einen Angreifer ist es schwerer, Glitch-Angriffe auf zwei Instruktionen durchzuführen, zwischen denen nur wenige Taktzyklen erfolgen. Angriffe können erschwert werden, indem Operationen zwei- oder sogar dreimal durchgeführt werden. Wenn möglich, sollte die Operation beim zweiten Mal invers durchgeführt werden und mit den Anfangswerten überprüft werden (vgl. Abbildung 5.3). Bei dem Einfügen von Redundanz muss allerdings aufgepasst werden, dass der Compiler diese nicht durch die Optimierung wieder entfernt.

```

1 #include <stdlib.h>
2 #include <inttypes.h>
3 #include <stdbool.h>
4
5 uint32_t critical_val = 42;
6
7 bool check_test(void)
8 {
9     if ((critical_val == 42) &&
10         !(critical_val != 42)) {
11         goto res_true;
12     }
13     return false;
14 res_true:
15     return true;
16 }
17
18 int main(void)
19 {
20     bool res = check_test();
21     return (int)res;
22 }

```

Abbildung 5.3: Redundante Abfrage [10]

Fehlerresistente Programmierung sollte grundsätzlich verwendet werden, da nicht vor Angriffen geschützt wird, sondern auch funktionale Sicherheit gewährleistet werden kann.

Verzögerungen Ein wichtiger Teil der Fehlerinjektion ist das Finden des richtigen Zeitpunktes. Um das Timing von solchen Angriffen weiter zu erschweren, können zufällige Verzögerungen für Security Prüfungen eingebaut werden (vgl. Abbildung 5.4).

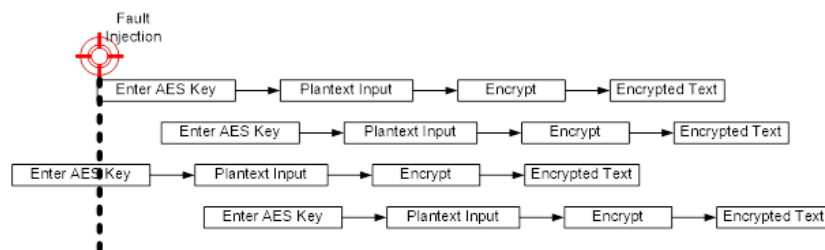


Abbildung 5.4: Zufällige Verzögerung [16]

Wenn möglich sollten die einzelnen Aufgaben ebenfalls in einer zufälligen Reihenfolge durchgeführt werden (vgl. Abbildung 5.5).

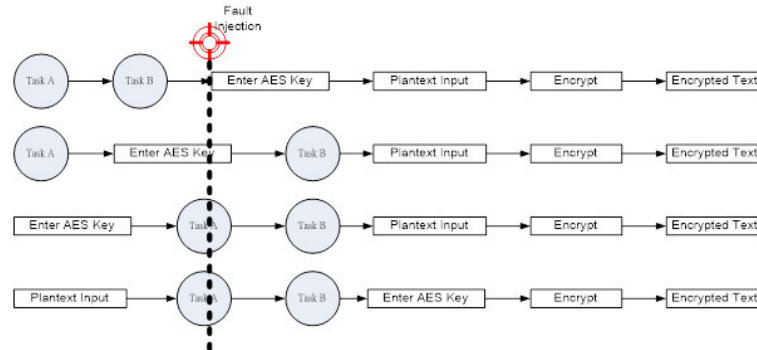


Abbildung 5.5: zufällige Ausführungsreihenfolge [16]

Je schwerer es ist das Verhalten der Software vorherzusagen, umso schwerer wird das Einfügen von Fehlern.

5.1.4 Authentizitätsprüfungen

Abfrage der Device ID Viele Mikrocontroller besitzen eine einzigartige Device ID. Die Device ID kann genutzt werden, um die Authentizität der Hardware zu überprüfen. Diese kann am Anfang der Software abgefragt werden, um herauszufinden, ob es das erwartete Gerät ist. Die Device ID mit einem Hash zu verbinden, macht keinen Sinn, wenn es keinen sicheren Speicher für den öffentlichen Schlüssel gibt beziehungsweise keine Verbindung zum Internet besteht.

Eine Möglichkeit die Abfrage sicherer zu machen, können weitere Werte aus dem One Time Programmable (OTP) Bereich genommen werden. Diese werden dann mit der Device ID verknüpft (z.B. mit einem XOR). Diesen Wert fragt die Software am Anfang ab.

Um den erwarteten Wert abzusichern, kann dieser aufgeteilt und an verschiedenen Orten gespeichert werden. Eine Empfehlung ist, einen der Werte im ROM zu speichern. Die Werte sollten auch in „zufälligen“ Bereichen liegen (nicht z.B. immer am Anfang des Speichers).

Die Aufteilung des Wertes sollte nicht nach Bytes sein (Byte 1 in Bereich 1, etc.). Der Wert sollte in Teile aufgeteilt werden, die die gleiche Länge haben wie der erwartete Wert. Die Teile müssen dann mit einem XOR miteinander verknüpft werden, damit der erwartete Wert daraus entsteht. (vgl. Kapitel 2.3.5).

Signatur der Software zusammen der Device ID Die Firma SEGGER schlägt in [7] vor, die Software mit der Device ID zusammen zu signieren. Damit sollen das Klonen von Produkten erschwert werden. Die erwartete Signatur soll dabei in einen permanenten Speicher abgespeichert werden. (Wahrscheinlich kann der OTP Bereich dafür verwendet werden.) Beim Starten des Controllers wird die Signatur verifiziert. SEGGER bietet für dieses Verfahren ihre eigene Softwarelösung für Embedded Systems an.

Physical Unclonable Function (PUF) Physical Unclonable Functions sind Hardware Strukturen, die zur eindeutigen Identifikation von Halbleitern bzw. zum Sichern/Erzeugen von kryptografischen Schlüsseln dienen. Es gibt viele verschiedene Umsetzungsmöglichkeiten für PUFs. Eine Umsetzungsmöglichkeit sind SRAM PUFs. Dabei wird die Tatsache ausgenutzt, dass beim Einschalten die Bits zufällig belegt sind. Für den PUF ist entscheiden, dass diese Belegung der Bits beim Einschalten immer (oder überwiegend) gleich ist und von System zu System unterschiedlich. Dieses Verfahren wird häufig bei Mikrocontroller (z.B. bei NXP LPC55S6x) verwendet.

Generell wird zwischen starken und schwachen PUFs unterschieden. Jede PUF kann als Challenge-Response-System interpretiert werden. Die PUF bekommt als Input eine Challenge c und antwortet mit einer Response $r = f(c)$. $f()$ beschreibt dabei das Verhältnis zwischen Input und Output. Bei einem SRAM-PUF ist zum Beispiel die Challenge der Anschluss der Spannung an den SRAM. Die Response sind die Anfangswerte im Speicher. Der Hauptunterschied zwischen starken und schwachen PUFs ist die Anzahl an möglichen $f()$ und die Anzahl von einzigartigen Challenges c , die ein PUF verarbeiten kann.

Schwache PUFs (z.B. SRAM-PUF, Ringoszillator-PUF) werden in erster Linie für die Generierung und Speicherung kryptografischer Schlüssel verwendet, ohne auf sicheren Speicher angewiesen zu sein. Starke PUFs (z.B. optische PUFs) werden für die Authentifizierung verwendet.

Nur wenige Mikrocontroller geben an, dass bei ihnen PUF nutzbar sind. STM32 bietet keine solchen Mikrocontroller, NXP und Maxim Integrated hingegen schon. [17] zeigt, dass man auch den SRAM eines STM32 als einen schwachen PUF nutzen kann, wenn auch nicht komplett. Es ist allerdings nicht geklärt, wie eindeutig (und nicht klonbar) dieser PUF ist. Des Weiteren waren von den 512 Bits ungefähr 268 nutzbar. Die genaue Anzahl hängt vom Board ab.

Physical Unclonable Functions können abhängig von der Umsetzung Schwächen (z.B. Schwach gegen Seitenkanalangriffe, Funktion doch kopierbar etc.) aufweisen oder können umgangen werden.

5.1.5 Sicherheitsmaßnahmen für Verschlüsselungen

Grundsätzlich gilt, dass Verschlüsselungen nur verwendet werden sollten, wenn ein sicherer Speicher verfügbar. Gilt das nicht, können Schlüssel einfach aus dem Speicher ausgelesen werden.

Secure Elements Ein Secure Element ist ein Mikroprozessor, der genutzt wird, um sensitive Daten (z.B. kryptografische Schlüssel) zu speichern und sichere Programme auszuführen. Diese sind stärker gegen Manipulationen von außen geschützt. Wird ein Manipulierungsversuch erkannt, werden die relevanten Informationen zerstört.

Ein Secure Element bedeutet nicht automatisch, dass ein System nicht angreifbar ist. Die Kommunikation zwischen dem Mikrocontroller und dem Secure Element bleibt angreifbar und muss daher ebenfalls abgesichert werden. Wenn das Secure Element zum Verschlüsseln der Software (also des Flash-Speichers) genutzt wurde, dann muss auch der Ort, in dem die entschlüsselte Software liegt, (z.B. der RAM) geschützt werden.

Vergleich der TrustZone mit einem Secure Element Die TrustZone kann genutzt werden, um z.B. ein Secure Element in Software zu emulieren. Daher liegt ein Vergleich der beiden Security Technologien nahe:

Ein Secure Element bietet im Vergleich zur TrustZone Implementierung einen höheren Schutz gegen lokale Angriffe (also invasive/ semi-invasive Angriffe) (vgl. [13]). Gegen Angriffe aus der Entfernung (z.B. Einfügen von Schadcode über die seriellen Schnittstellen), bieten beide Technologien einen guten Schutz.

Die TrustZone bietet im Vergleich zu einem Secure Element mehr Flexibilität. (vgl. [13]) Die Softwareentwicklung für ein Secure Element ist komplexer als für einen Mikrocontroller mit einer TrustZone. Bei einem Secure Element muss ein IC zum System hinzugefügt werden. Das beinhaltet das Hinzufügen von Kommunikationstreibern. Des Weiteren haben Secure Elements oft ihre eigenen Entwicklungsprozesse und Toolchains. Die Entwicklung für Secure Elements kann teure Werkzeuge, Geheimhaltungsvereinbarungen, etc. beinhalten. Bei der TrustZone hingegen können Standard-Toolchains verwendet werden.

Schutzmaßnahmen in der Software Eine Gefahr für Verschlüsselungen sind Seitenkanalangriffe. Daher müssen Verschlüsselungen so programmiert werden, dass diese zum einen nicht von der Zeit abhängen und zum anderen stets den gleichen Stromverbrauch haben. Eine solche Implementierung wird oft von Kryptografie-Bibliotheken bereits unterstützt.

Zum anderen empfiehlt es sich parallel zur Verschlüsselung andere Ressourcen im System zu verwenden, um einen Seitenkanalangriff über das Magnetfeld zu erschweren.

Bei der Programmierung von Verschlüsselungen gelten grundsätzlich auch die Regel zur fehlerresistenten Programmierung (vgl. Kapitel 5.1.3).

5.1.6 Hardware Schutzmaßnahmen

Eine große Gefahr geht von invasiven Angriffen aus. Hier ist es nicht möglich rein in der Software Schutzvorkehrungen zu treffen. Da sich diese Thesis primär mit Schutzmaßnahmen in Software beschäftigt, werden die Hardware Schutzmaßnahmen nur angeschnitten.

Schalter am Gehäuse des Gerätes Eine einfache Methode Schutz vor invasiven Angriffen einzubauen, ist die Verwendung von Schaltern am Gehäuse. Wird das Gehäuse geöffnet oder entfernt, werden die Schalter ausgelöst. Der Mikrocontroller kann dann darauf reagieren, indem zum Beispiel geheime Informationen gelöscht werden. Ein solcher Schutz lässt sich allerdings auch leicht umgehen.

Zugänglichkeit der Schnittstellen Eine weitere einfache Möglichkeit, Angriffe zu erschweren, ist es die seriellen Schnittstellen (wenn möglich) in den Layern einer Platine zu verstecken. Auch diese Methode durch einen erfahrenen Angreifer mit ausreichend Budget umgangen werden. Das Verstecken der Schnittstellen birgt auch Nachteile für die Entwickler, da auch sie nur noch schwer an die Schnittstellen rankommen.

Schutzfunktionen in Secure Mikrocontrollern Secure Elements und Secure Mikrocontroller besitzen in der Hardware eingebaute Maßnahmen. Zum einen befindet sich in dem Gehäuse ein sogenanntes Wire Mesh. Bei diesem handelt es sich um ein Leitungsgitter. Wird das Gehäuse für einen invasiven Angriff weggeätzt, erkennt der Mikrocontroller das und kann Maßnahmen treffen (siehe Abbildung 5.6).

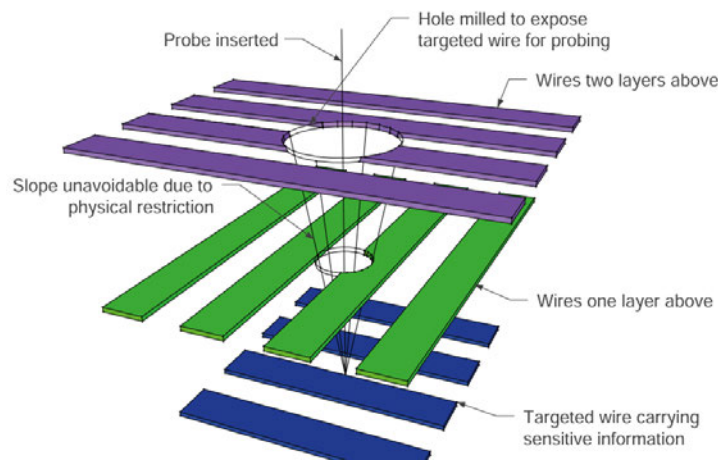


Abbildung 5.6: Wire Mesh [30]

Um Microprobing zu verhindern, werden außerdem die Kapazitäten der Leitungen des Prozessors überwacht. Ändert sich die Kapazität einer Busleitung durch einen Draht, kann der Controller darauf reagieren.

5.2 Security Funktionen bei ST-Controllern

5.2.1 Readout Protection (RDP)

Funktionsweise STM32-Controller bieten einen Leseschutz für den Flash-Speicher. Dieser Schutz besitzt drei Levels:

Level 0: Das ist die Standardeinstellung. In diesem Zustand gibt es keine Schutzfunktionen. Das Auslesen der Software ist komplett möglich. Dieser Modus ist für die Entwicklung gedacht.

Level 1: In diesem Zustand gibt es keinen Zugriff auf den Flash-Speicher (lesen, löschen, programmieren). Das Heruntersetzen von Level 1 auf Level 0 führt dazu, dass der Flash-Speicher und die Backup-Register gelöscht werden. Somit können Firmware-Updates erfolgen. Mit Hilfe des Debuggers kann weiterhin auf alle Register und auf den RAM zugegriffen werden. Beim Anschließen des Debuggers können sowohl der Debugger als auch der Mikrocontroller nicht mehr auf den Flash zugreifen. Der Controller hält daher an, da keine Instruktionen aus dem Flash mehr geholt werden können. Wird versucht auf den Flash zuzugreifen, wird ein Bus Error und ein Hard Fault ausgelöst.

Level 2: In diesem Zustand besteht kein Zugriff auf den Flash- und RAM-Speicher. Die Debug-Schnittstellen sind deaktiviert. Ein Heruntersetzen von Level 2 ist nicht möglich.

Bekannte Sicherheitslücken Es gibt einige bekannte Angriffe auf die Readout Protection von ST. Die erste Abhandlung 'Shedding too much Light on a Microcontroller's Firmware Protection' [19] wird näher im Kapitel 6.4 beschrieben.

In [18] zeigen Obermeier und Schink, dass es auch bei der STM32F1-Reihe möglich ist, die Readout Protection zu umgehen. Der Unterschied dieser Reihe zu den anderen STM32-Reihen ist, dass hier nur die Level 0 und 1 zur Verfügung stehen. Bei dem Angriff wird ausgenutzt, dass die Prozesse für die Ausnahmenbehandlungen immer noch Zugriff auf den Flash-Speicher über den ICode-Bus haben. Durch das Einfügen von Fehler (und dadurch das Auslösen von Ausnahmen) kann der Flash ausgelesen werden. Die Autoren waren damit in der Lage bis zu 94,5% des Speichers (abhängig vom Mikrocontroller) auszulesen.

Nedospasov, Datko, und Roth zeigen in [15], dass mit Hilfe eines Spannungsgiltches bei einem STM32F205-Mikrocontroller konnte der RDP Level von 2 auf 1 runtergesetzt werden und ein privater Schlüssel aus dem RAM gelesen werden. Dabei wurde für sechs Mikrosekunden die Spannung heruntergesetzt während die Optionsbytes gelesen wurden.

Die drei Berichte zeigen, dass der Schutz durch die Readout Protection bei den STM32F0-, STM32F1- und STM32F2-Reihen zu umgehen ist. Es ist anzunehmen, dass auch bei den anderen STM32-Reihen das Umgehen der Readout Protection mit diesen oder ähnlichen Methoden möglich ist. Auch bei anderen Herstellern mit einem ähnlichen Ausleseschutz sind Angriffslücken bekannt.

5.2.2 Weitere Mechanismen zum Schutz des Speichers

Flash Write Protection (WRP) Die WRP erlaubt, dass Teile des Flash-Speichers nicht beschrieben oder gelöscht werden können. Das kann genutzt werden, um zu verhindern, dass beispielsweise Schadcode eingefügt wird.

Proprietary Code Read Out Protection (PCROP) Bei der PCROP handelt es sich um die Möglichkeit einen Execute-Only-Memory-Bereich festzulegen (vgl. Kapitel 5.1.1). Dieser Schutzmechanismus sorgt also dafür, dass Teile des Flash-Speicher nur ausgeführt werden können, aber nicht gelesen oder beschrieben.

Secure Hide Protection (HDP) Die Secure Hide Protection ist ein Bereich des Flash-Speichers, der nur einmal nach dem Reset des Gerätes zugänglich ist. Die Idee ist, dass hier kritische Daten und Applikationen gespeichert werden. Diese werden nur beim Start des Gerätes genutzt und sind dann die restliche Laufzeit nicht mehr zugänglich. Der Unterschied zu dem PCROP ist, dass das Debuggen des HDP Bereichs bei seiner Durchführung immernoch möglich ist. Die HDP-Funktion dient in erster Linie dafür, dass Security-Code vom Rest isoliert werden kann.

Des Weiteren besitzen ST-Controller eine Memory Protection Unit und die Möglichkeit Bit-Fehler im Speicher mit dem ECC und der Paritätsüberprüfung zu korrigieren. Die STM32L5-Controller besitzen außerdem die TrustZone-Architektur.

5.2.3 Umgebungsüberwachung

Clock Security System (CSS) Das CSS dient dazu eine externen Taktquelle zu überwachen. In erster Linie soll erkannt werden, wenn ein externes Taktsignal entfällt. Beim Verlust des Signales wird schaltet das System auf einen internen Takt um. Das CSS eignet sich allerdings nicht dafür sehr kurze Taktglitches zu erkennen. Die Funktion sollte zwar auf jeden Fall bei der Nutzung eines externen Taktes verwendet werden. Es sollten allerdings weitere Sensoren zur Überwachung des Taktes verwendet werden, wenn Glitches erkannt werden sollen.

Programmable Voltage Detector (PVD) Der PVD ermöglicht die Spannungsversorgung zu überwachen. Damit kann eine Minimalspannung eingestellt werden. Fällt die Spannung unter diese Schwelle, wird ein Interrupt ausgelöst und es können entsprechende Maßnahmen vorgenommen werden. Die Funktion dient in erster Linie dazu, Brownouts zu erkennen. Hochfrequente Spannungsglitches oder eine zu hohe Betriebsspannung werden nicht erkannt. Auch hier sollte also auf weitere externe Sensoren zurückgegriffen werden.

Anti-Tamper-Pins (TAMP) Die Anti-Tamper-Pins bieten die Möglichkeit, dass wenn sich der Pegel an diesen Pins ändert, ein Interrupt ausgelöst wird, bei dem z.B. die Backup-Register gelöscht werden. Dieser Interrupt wird auch bei Low-Power-Zuständen ausgelöst. Dieser Pin kann daher genutzt werden, um z.B. zu erkennen, wenn das Gehäuse eines Gerätes geöffnet wurde. Es können auch externe Sensoren zur Umgebungsüberwachung damit verbunden werden. Einige der ST-Controller-Reihen besitzen interne Tamper-Pins, die die Spannung, Temperatur und den Takt überwachen.

Device ID Microcontroller von ST besitzen alle eine 96 Bit lange, einzigartige Device ID, die nicht verändert werden kann. Zusammengesetzt wird diese aus den X- und Y-Koordinaten des Wafers sowie der Nummer des Wafers und des Lot (vgl. [24]). Daraus ergibt sich, dass nicht der gesamte Wertebereich des 96 Bit ausgenutzt wird. [8] zeigt auf, dass die Device IDs starke Ähnlichkeit miteinander haben. Aus diesem Grund sollte die Device ID in kryptografischen Anwendungen (z.B. als Schlüssel) nicht für sich allein genutzt werden.

5.3 Kombination von Schutzmaßnahmen

Die vorherigen Kapitel zeigen, dass Schutzmaßnahmen für sich alleine oft weiterhin Sicherheitslücken aufweisen. Daher ist es empfehlenswert verschiedene Schutzfunktionen zu kombinieren, um die Schwächen auszugleichen.

5.3.1 Root und Chain of Trust

Bei der Chain of Trust handelt es sich um den Aufbau verschiedener Security-Mechanismen, die aufeinander aufbauen, wobei eine Komponente die Sicherheit der anderen Komponente garantieren soll. Die Root of Trust ist die Komponente am Anfang der Kette, die die Sicherheit aller anderen Komponenten gewährleistet [26].

5.3.2 Secure Boot

Der Secure Boot ist ein unveränderbarer Programmabschnitt, der beim Starten des Mikrocontrollers aufgerufen wird. Die Aufgabe des Secure Boot ist es, die Integrität und die Authentizität der Firmware zu überprüfen (vgl. Abbildung 5.7).

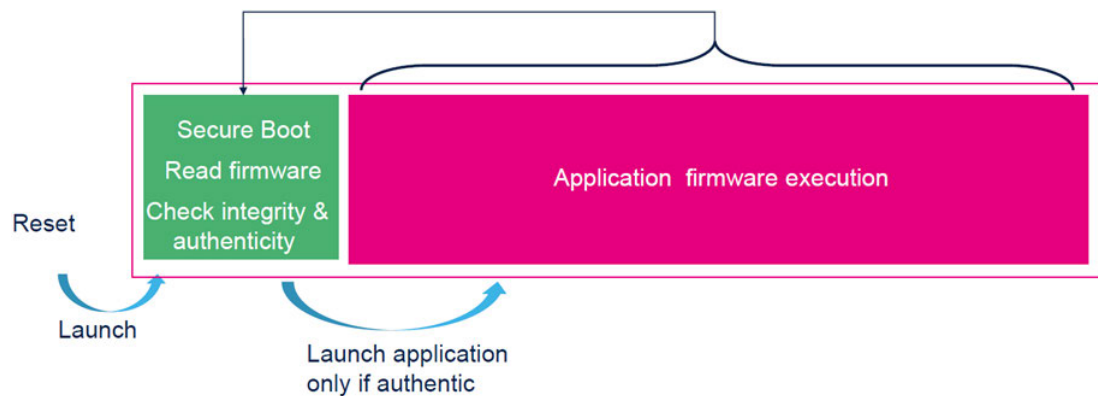


Abbildung 5.7: Funktionsprinzip Secure Boot [27]

Neben der Integrität und Authentizität der Firmware prüft der Secure Boot auch die eingestellten Security Mechanismen (z.B. Option Bytes) beziehungsweise setzt die dynamischen Schutzfunktionen (z.B. MPU).

Bei der Integritätsprüfung handelt es sich um die Berechnung eines Hashes. Der berechnete Hash wird mit einem abgespeicherten Wert verglichen. Bei der Authentizitätsprüfung handelt es sich um die Prüfung einer Signatur. Die Signatur entsteht, indem entweder die Firmware oder nur der Hash, mit einem privaten Schlüssel verschlüsselt werden. Der Secure Boot prüft die Signatur mit Hilfe des öffentlichen Schlüssels.

Bei dem Secure Boot muss sichergestellt werden, dass von außen oder von Firmware kein Zugriff auf die abgespeicherten Schlüssel besteht.

5.3.3 Abhängigkeiten von Schutzfunktionen

Funktionen, wie der Secure Boot, bauen auf verschiedenen Schutzmechanismen auf, um ihre Sicherheit zu gewährleisten. Im folgenden werden am Beispiel von einem STM32F4- und STM32L5-Controller gezeigt, wie die Schutzfunktionen aufeinander aufbauen.

STM32F4 Ziel ist es die drei Security-Ziele (Vertraulichkeit, Integrität und Authentizität) zu gewährleisten:

Um die Vertraulichkeit vor Zugriff von außen zu gewährleisten, kann zum einen der Tamper-Pin verwendet werden. Mit ihm lässt sich feststellen, ob das Gehäuse eines Gerätes geöffnet oder die Umgebung manipuliert wurde. Die zweite Maßnahme zum Schutz der Vertraulichkeit ist die Readout Protection. Sie blockiert Lesezugriffe über den Debugger. Von innen wird die Vertraulichkeit mit Hilfe der MPU geschützt.

Die Integrität der MPU-Konfiguration wird zum einen durch die Readout Protection (vor Angriffen von außen) und zum anderen durch den Schreibschutz (WRP) (vor Angriffen von innen) gesichert.

Die Integrität der Authentizitäts- und Integritätsprüfungen im Secure Boot werden zum einen durch fehlerresistente Programmierung (z.B. Redundanz) sichergestellt. Zum anderen wird mit Hilfe der Umgebungsüberwachung die Wahrscheinlichkeit von Glitch-Angriffen verringert.

Die Integritäts- und Authentizitätsprüfung selbst überprüfen die Firmware.

Die Abbildung 5.8 zeigt den Aufbau der Schutzfunktionen. Wenn eine der unteren Ebenen fehlt, werden die oberen Ebenen angreifbarer.

Legende:
 Vertraulichkeit: grün
 Integrität: blau
 Authentizität: rot
 Heller: Optionale Funktion

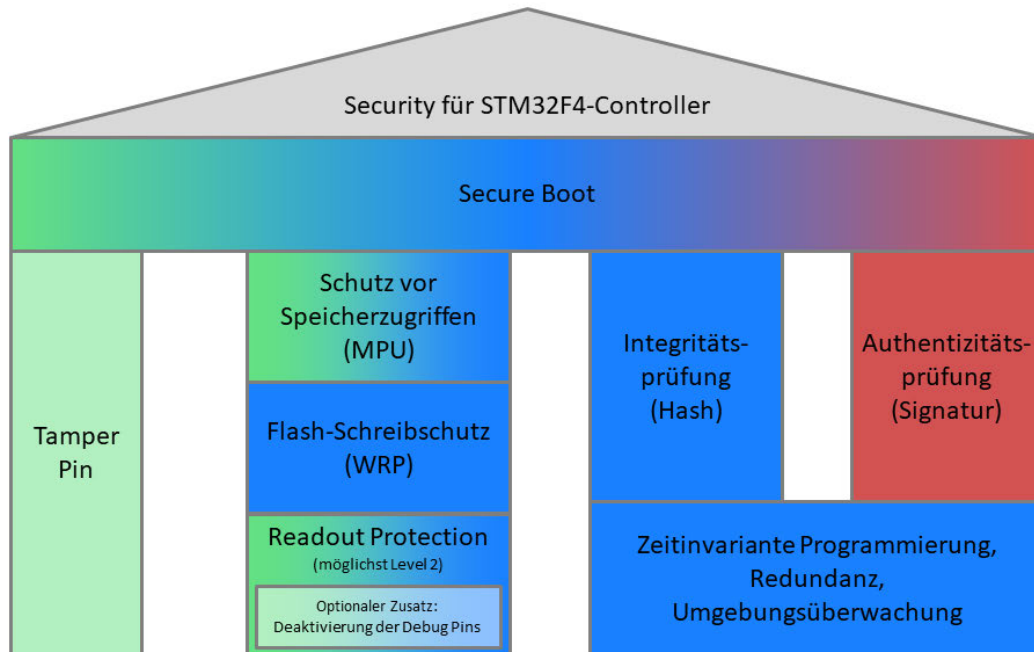


Abbildung 5.8: Abhängigkeiten der Schutzfunktionen bei einem STM32F4

STM32L5 Der Aufbau der Security-Funktion ist ähnlich wie beim F4-Controller. Einer der Unterschiede ist, dass hier mehr Schutzmechanismen zur Isolation des Secure Boot genutzt werden (vgl. Abbildung 5.9).

Eine der zusätzlichen Isolationsfunktionen ist die TrustZone. Der Secure Boot wird damit stärker von den Rest des Speichers isoliert. Die zweite Isolationsfunktion ist das Verbergen des Start-Up-Codes. Nachdem der Secure Boot durchgelaufen ist, ist der entsprechende Speicherbereich sowohl von innen als auch von außen nicht mehr sichtbar. Diese Isolationsfunktion ist optional.

Der zweite Unterschied ist der Boot Lock. Dieser schützt die Integrität des Secure Boot. Damit wird sichergestellt, dass beim Start des Controllers stets der Secure Boot ausgeführt wird. Ein Angreifer kann also nicht die Boot-Adresse manipulieren, um den Secure Boot zu überspringen.

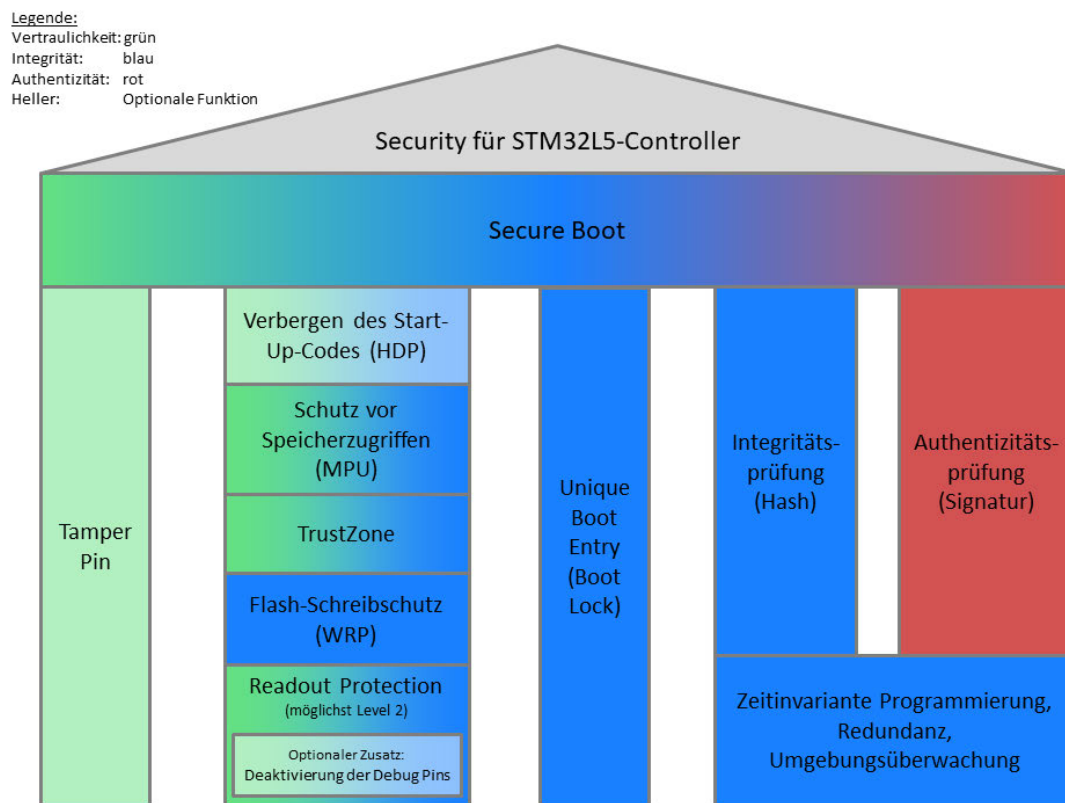


Abbildung 5.9: Abhängigkeiten der Schutzfunktionen bei einem STM32L5

6 Untersuchung von Schutzmaßnahmen auf ausgewählten Mikrocontrollern

6.1 Voraussetzungen der Untersuchung

Angreifer Bevor Schutzmaßnahmen umgesetzt werden können, muss zunächst festgelegt werden, vor welchem Angreifer der Mikrocontroller geschützt werden soll. In der folgenden Untersuchung wird ein Angreifer vorausgesetzt, dem nur ein begrenztes Budget zur Verfügung steht. Das bedeutet, dass dieser in erster Linie Software-Angriffe durchführen kann. Außerdem ist er in der Lage einfache nicht-invasive und semi-invasive Angriffe (z.B. Seitenkanalangriffe oder Glitch-Angriffe auf Spannung oder den Takt) durchzuführen.

Ausgehend von diesem Angreifer müssen folgende Bereiche geschützt werden:

- Serielle Schnittstellen (Debugger-Zugriff)
- Speicherbereiche, in denen der Security-Code umgesetzt wird
- Schutz vor dem Überspringen von sicherheitskritischem Code

Sämtlicher Code der beiden Projekte lässt sich im Anhang finden.

Auswahl der Mikrocontroller Die Firma Reese+Thies arbeitet viel mit General Purpose Mikrocontrollern von STMicroelectronics. Die Umsetzung der Schutzmaßnahmen soll dementsprechend auf diesen Controllern erfolgen. Es sollen die Möglichkeiten des Controllers für sich alleine untersucht werden. Daher soll keine zusätzliche Hardware, wie zum Beispiel Secure Elements, verwendet werden.

Basierend auf diesen Kriterien wurden zwei Mikrocontroller ausgewählt: ein STM32F4- und STM32L5-Controller. STM32F4-Controller werden bereits oft in der Firma eingesetzt. Daher soll untersucht werden, welcher Schutzlevel mit diesen Controllern erreicht werden kann. Die STM32L5-Controller sind eine neue Mikrocontrollerreihe, die TrustZone Architektur verwendet. Bei beiden Controllern wurden Nucleo Boards zur Untersuchung verwendet.

6.2 Untersuchung des STM32F439ZI

6.2.1 Grundfunktionen des Mikrocontrollers

STM32F4-Controller besitzen im Vergleich zu anderen STM32-Reihen eher wenig eingebaute Schutzfunktionen (vgl. [26]). Folgende Schutzmechanismen sind in dem Controller vorhanden: MPU, RDP und WRP wie alle STM32 Controller. Ein Pin lässt sich als Tamper-Pin konfigurieren. Zusätzlich lassen sich Hardwarebeschleunigung für Kryptografie aktivieren.

6.2.2 Funktionsumsetzung

Ziel ist eine Software umzusetzen, die sämtliche möglichen Schutzmaßnahmen des Controllers ausnutzt. Es bietet sich daher an, eine Secure Boot Funktion zu implementieren (vgl. Kapitel 5.3.2). Da eine einen Secure Boot zu programmieren den Rahmen dieser Thesis sprengen würde, wird hier auf eine fertige Lösung zurückgegriffen.

ST bietet eine bereits fertige Beispielumsetzung einer Secure Boot und Secure Firmware Update (SBSFU) Funktion für ihre STM32-Controllerreihen. Diese nutzt alle möglichen Schutzmechanismen der jeweiligen Controller, um diese Funktionen abzusichern.

Ablauf des Secure Boot Der Secure Boot verläuft grundlegend wie folgt: Zu Beginn wird geprüft, ob die Option Bytes richtig gesetzt werden. In den Option Bytes ist zum einen die Readout Protection gesetzt und zum anderen die Write Protection für den Secure Boot Bereich im Speicher. Da der Controller noch in dem Entwicklungszustand belassen werden soll, wird nur die Level 1 Readout Protection verwendet. Sollten die Option Bytes nicht wie erwartet konfiguriert sein, werden diese korrigiert und der

Controller neu gestartet. Mit der MPU werden die privilegierten Bereiche dem Secure Boot zugewiesen. Im nächsten Schritt wird die Signatur der Nutzersoftware geprüft. Bei einer korrekten Signatur wird aus der Nutzersoftware ein Hash berechnet und mit dem abgespeicherten Wert verglichen. Sollte eine die Signatur- oder Hashprüfung scheitern, wird die Nutzersoftware gelöscht und der Secure Boot bleibt in einer Endlosschleife. (vgl. Abbildung 6.1).

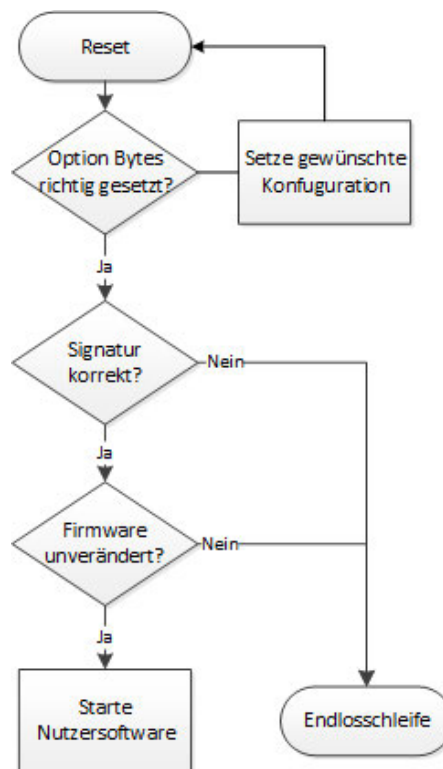


Abbildung 6.1: STM32F4 - vereinfachter Ablauf des Secure Boot

Aufteilung des Speichers Der Flash-Speicher des Controllers wird in drei Teile aufgeteilt (vgl. Abbildung 6.2). In dem ersten Teil befindet sich der Code des Secure Boot, in dem zweiten Teil die Meta-Daten und in dem dritten die Nutzerapplikation.



Abbildung 6.2: Speicheraufteilung STM32F4

Meta-Daten Bei den Meta-Daten handelt es sich um die Signatur der Nutzersoftware. Diese entsteht, in dem zunächst aus der Nutzersoftware ein Hash (SHA-256) berechnet wird. Der Hash soll die Integrität der Software gewährleisten. Wurde die Software von einem Angreifer vor dem Programmstart manipuliert, so wird beim Secure Boot ein anderer Wert Hash-Wert errechnet, der nicht mit dem in den Meta-Daten übereinstimmt. Der Hash-Wert wird mit Hilfe des privaten Schlüssels durch das ECDSA-Verfahren signiert (vgl. Abbildung 6.3). Der Secure Boot prüft die Signatur, indem mit Hilfe des öffentlichen Schlüssels die Signatur entschlüsselt wird und den erwarteten Hash ergibt.

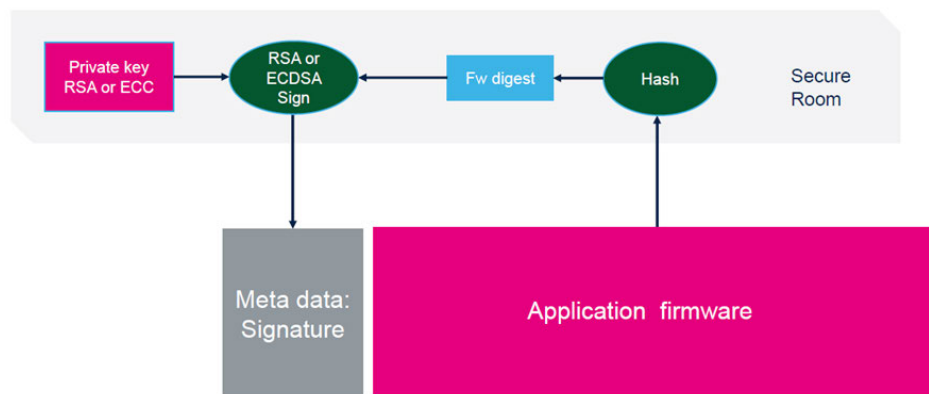


Abbildung 6.3: Berechnung der Signatur [27]

Weitere Möglichkeiten der SBSFU Der SBSFU bietet theoretisch noch die Möglichkeit den Speicherbereich der Nutzerapplikation in zwei Bereiche aufzuteilen: einen für die aktive Applikation und einen Bereich, in denen man temporär Updates laden kann. Dadurch ist es möglich ein Firmware-Update in der Nutzerapplikation einzuleiten. Bei einem Neustart prüft der Secure Boot dann die neue Applikation auf ihre Integrität und Authentizität. Bei einer erfolgreichen Prüfung wird diese dann in Bereich der aktiven Applikation kopiert und ausgeführt. Hier wurde allerdings auf diese Umsetzung verzichtet, um den Speicherbereich der Nutzerapplikation zu maximieren.

Die Code des Secure Boot und die Meta-Daten dürfen hierbei nicht von der Nutzerapplikation lesbar oder schreibbar sein. Dies wird mit Hilfe der Write Protection und der MPU gewährleistet.

Funktionsweise des Secure Firmware Updates Die Secure Firmware Update Funktion gibt die Möglichkeit, die Nutzerapplikation zu aktualisieren. Der Upload einer neuen Applikation erfolgt über UART. Die Update-Datei besteht aus den Meta-Daten und der verschlüsselten Applikation. Die Verschlüsselung erfolgt mit dem AES-Algorithmus (vgl. Abbildung 6.4).

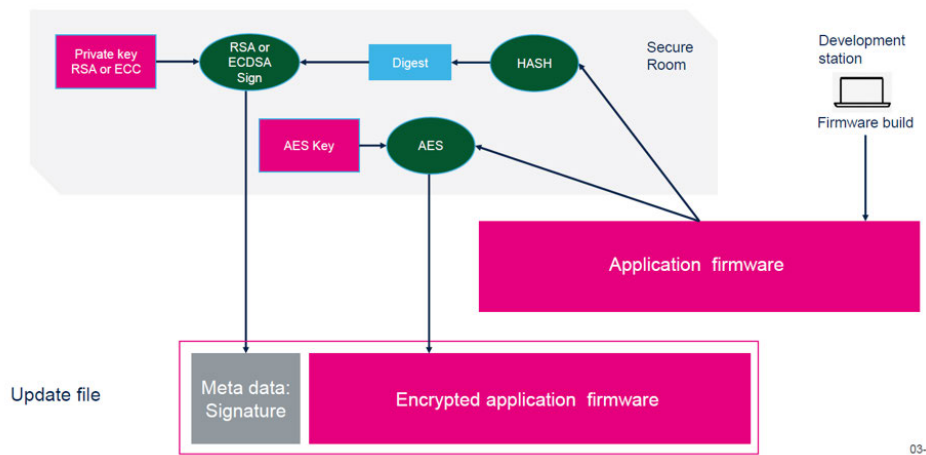


Abbildung 6.4: Secure Firmware Update [27]

Die Beispiel-Nutzerapplikation überprüft die eingebauten Schutzmechanismen.

Portierung des SBSFU Die von STM bereitgestellte Umsetzung des SBSFU beinhaltet nur eine Umsetzung für einen F4-Controller: den STM32F413H-Discovery. Dieser muss daher zunächst auf den STM32F439ZI portiert werden. Die Portierung wurde mit Hilfe der Dokumente [29] und [25] umgesetzt. Hierfür müssen zunächst die Pin-Konfigurationen der LEDs und der Tamper Pin angepasst werden. In der Konfigurationsdatei für das Speicherlayout sowie in den Linker Skripts werden die neuen Speicherbereiche zugewiesen.

Das Listing 6.1 zeigt beispielhaft die neue Zuweisung der Adressen für das Firmware Image. Um den Speicher für die Nutzerapplikation zu maximieren, wurden die Slots für weitere Applikationen entfernt.

```

/* Slots must be aligned on sector size */

/* Active slot #1 (1472 kbytes) */

```

```

__ICFEDIT_SLOT_Active_1_start__ = 0x08010000;
__ICFEDIT_SLOT_Active_1_end__   = 0x081FFFFFF;
__ICFEDIT_SLOT_Active_1_header__ = __ICFEDIT_SLOT_Active_1_start__;

/* Slots not configured */
__ICFEDIT_SLOT_Active_2_header__ = 0x00000000;
__ICFEDIT_SLOT_Active_2_start__  = 0x00000000;
__ICFEDIT_SLOT_Active_2_end__    = 0x00000000;
__ICFEDIT_SLOT_Active_3_header__ = 0x00000000;
__ICFEDIT_SLOT_Active_3_start__  = 0x00000000;
__ICFEDIT_SLOT_Active_3_end__    = 0x00000000;

```

Listing 6.1: STM32F4 - Angepasstes Mapping für die Firmware

Aktivierte Schutzfunktionen Der Secure Boot wird durch mehrere der Security-Features geschützt. Mit diesen soll eine Chain of Trust aufgebaut werden (vgl. Abbildung 6.5). Die Readout Protection schützt die Option Bytes und damit die Konfiguration der Write Protection. Die Write Protection schützt die Konfiguration der MPU. Die MPU schützt wiederum die SBSFU-Software vor unberechtigten Zugriffen aus der Nutzerapplikation. Die Kryptografie-Funktionen verifizieren die Nutzerapplikation.

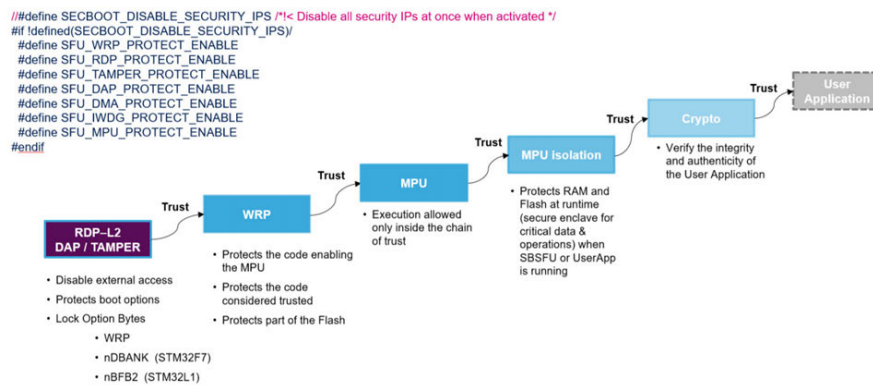


Abbildung 6.5: Aktivierte Schutzfunktionen [25]

Anpassung der Nutzerapplikation In der Nutzerapplikation wurden zur bessern Übersicht die Funktionsprüfungen entfernt, die zu den anderen Mikrocontrollerreihen

gehören (z.B. Prüfung der Firewall, Firmware Image Download in der Nutzerapplikation).

6.2.3 Einfügen einer Device ID Prüfung

Der Secure Boot von ST prüft mit der Signatur nur die Authentizität der Software. Um den Schutz vor dem Kopieren der Software zu erhöhen, bietet es sich an, auch die Authentizität der Hardware zu prüfen. Daher wurde eine Device ID Prüfung zu Beginn der Nutzer-Software hinzugefügt.

Aufteilung der Device ID Um die Sicherheit weiter zu erhöhen, wurde die Device ID in drei Teile basierend auf dem Secret Sharing Verfahren (vgl. 2.3.5) aufgeteilt. Die ersten beiden Teile wurden mit einem zufallsgenerator generiert. Aus diesen beiden Teilen wurde der dritte Teil berechnet (vgl. 6.2). Die drei Teile werden an verschiedenen Stellen des Flash-Speicher abgelegt. Die jeweiligen Regionen werden im Linker-Skript definiert und im Code werden die Werte diesen mit Hilfe von 'section' zugewiesen.

```
/// \brief Expected Device ID {0x4E003E, 0x31375119, 0x39373236}
///
/// the Device ID is split in three parts.
/// Part 1 and 2 were generated by a random number generator
/// Part 3 was calculated: Part3 = DeviceId XOR Part1 XOR Part2
static const uint32_t uid_u32DeviceIDPart1[UID_DEVICE_ID_LENGTH]
    ↪ __attribute__((section(".myDeviceIDPart1"), used)) = {0xAF15AFD8,
    ↪ 0x1B56904B, 0x3DF36453};
static const uint32_t uid_u32DeviceIDPart2[UID_DEVICE_ID_LENGTH]
    ↪ __attribute__((section(".myDeviceIDPart2"), used)) = {0x06BC6A80,
    ↪ 0x0F212DC7, 0xBAFFFD49};
static const uint32_t uid_u32DeviceIDPart3[UID_DEVICE_ID_LENGTH]
    ↪ __attribute__((section(".myDeviceIDPart3"), used)) = {0xA9E7C566,
    ↪ 0x2540EC95, 0xBE3BAB2C};
```

Listing 6.2: Aufteilung der Device ID

Software Schutzmaßnahmen In der Programmierung wurden Software - Schutzmaßnahmen aus dem Kapitel 5.1.3 eingebaut, mit denen die Fehlerinjektion erschwert wird. Zum einen werden die Abfragen doppelt durchgeführt (vgl. Listing 6.3).

```
if((u32DeviceID[u8IDPos] != u32ExpectedID[u8IDPos]) &&
    !(u32DeviceID[u8IDPos] == u32ExpectedID[u8IDPos]))
{
    boUIDcorrect = SECURE_FALSE;
}
```

Listing 6.3: Redundante Abfrage

Zum anderen wurde eine 'Secure Bool' definiert, der nicht 0 und 1 für *true* und *false* verwendet (vgl. Listing 6.4).

```
/// Secure Bool
typedef enum
{
    SECURE_FALSE = 0xACEFAECF,           ///< FALSE-Definition
    SECURE_TRUE = 0xCA38C3E8             ///< TRUE-Definition
} SEC_enSecureBool_t;
```

Listing 6.4: Secure Bool

Funktionsprüfung Bei der praktischen Umsetzung der Device ID Prüfung auf STM32F4 trat ein unerwartetes Problem auf: Das Auslesen der Device ID aus der Nutzerapplikation führt dazu, dass die MPU einen Reset auslöst. Anscheinend wird der Speicherbereich, in dem sich die Device ID befindet, als ein privilegierter Bereich betrachtet, obwohl der Bereich undefiniert belassen wurde. Die Region zu den unprivilegierten Regionen hinzuzufügen ist nicht möglich, da die Secure Boot Anwendung bereits alle definierbaren Regionen ausgenutzt werden. Die Untersuchung, warum die Device ID zum privilegiertem Bereich gehört wurde abgebrochen, da die genaue Funktionsweise der MPU nicht Kernbestandteil der Thesis ist. Daher wird die Device ID Prüfung deaktiviert, wenn die MPU aktiviert ist.

Nachteil einer Device ID Prüfung Die Device ID Prüfung bietet eine einfache Methode, um den Schutz vor dem Kopieren der Software auf ein anderes Gerät zu erhöhen. Die Prüfung hat allerdings auch einen Nachteil: Die Fertigung wird erschwert.

Durch die Device ID Prüfung muss in der Fertigung bei jedem Controller erst die Device ID ausgelesen werden. Diese Device ID muss dann in dem Code eingefügt und neu gebaut

werden. Um eine solche Situation zu vermeiden, wurde ein Python Skript geschrieben, mit dem die zu erwartende Device ID direkt in die Elf-Datei eingefügt wird. Das Skript befindet sich im Anhang.

Ein solches Skript erleichtert die Fertigung etwas, es ist jedoch trotzdem aufwendig die Device ID jedes Mal auszulesen.

6.2.4 Untersuchung des Secure Boot

Dieses Kapitel beschäftigt sich mit der Funktionsprüfung des Secure Boot. Neben der Untersuchung der grundsätzlichen Funktionalität sollen auch einige einfache Angriffe durchgeführt werden.

Hauptmenü Die Kommunikation mit dem Mikrocontroller erfolgt über UART. Abbildung 6.6 zeigt das Hauptmenü der Secure Boot und Secure Firmware Applikation. Die Ausgabe des Secure Boot (der obere Teil der Ausgabe) ist nur im Debug Modus sichtbar. Hier sind die Ergebnisse der einzelnen Prüfungen sichtbar. Der untere Teil der Ausgabe gehört zur Nutzerapplikation. Hier können die eingebauten Tests ausgewählt und durchgeführt werden.

```

= [SB00T] System Security Check successfully passed. Starting...
=====
=                                     =
=      (C) COPYRIGHT 2017 STMicroelectronics      =
=                                     =
=      Secure Boot and Secure Firmware Update      =
=====

= [SB00T] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SB00T] STATE: CHECK STATUS ON RESET
=                                     INFO: A Reboot has been triggered by a Hardware reset!
=                                     INFO: Last execution detected error was: No error. Success.
= [SB00T] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SB00T] STATE: CHECK USER FW STATUS
=                                     A FW is detected in the slot SLOT_ACTIVE_1
= [SB00T] STATE: VERIFY USER FW SIGNATURE
= [SB00T] STATE: EXECUTE USER FIRMWARE

=====
=                                     =
=      (C) COPYRIGHT 2017 STMicroelectronics      =
=                                     =
=      - User App -      =
=====

===== Main Menu =====
Download a new Fw Image ----- 1
Test Protections ----- 2
Test SE User Code ----- 3
Print Device ID ----- 4
Selection :

```

Abbildung 6.6: STM32F4 - SBSFU Menü

Test der Readout Protection Zunächst wird die Grundfunktionalität der Readout Protection überprüft. Da es möglich sein soll, den Mikrocontroller zu debuggen beziehungsweise das Programm wieder zu entfernen, wird nur die Level 1 Readout Protection verwendet. Zum Testen der Readout Protection wurde das Programm STM32 ST-LINK Utility verwendet.

Bei dem ersten Test wird bei aktivierter Readout Protection versucht, den Flash auszu-
lesen. Der Zugriff auf den Falsh wird verhindert und der Controller hält an. Das Auslesen
des RAM und der Peripherie-Register ist möglich. Damit entspricht das Verhalten dem
Sollverhalten.

Beim zweiten Test wird versucht den Flash zu beschreiben. Der Zugriff auf den Flash wird blockiert und das Schreiben wird verhindert. Der Controller hält an. Das Verhalten entspricht dem Sollverhalten.

Bei dem dritten Test wird die Readout Protection deaktiviert (also auf Level 0 zurückgesetzt). Es wird geprüft, ob im Flash Daten zu finden sind. Wie vom Datenblatt vorgeschrieben, wurden der gesamte Speicherinhalt gelöscht. Das Verhalten entspricht dem Sollverhalten.

Es wurde außerdem der Angriff 'Firmware Extraction' aus 'Shedding too much Light on a Microcontroller's Firmware Protection' nachgestellt. Die genaue Beschreibung befindet sich im Kapitel 6.4.

Test der Update Funktion Als nächstes wird die Grundfunktionalität der Update-Funktion überprüft. Zunächst wird überprüft, ob ein Update mit einer korrekten Nutzerapplikation überhaupt funktioniert. Die Update-Datei wird erfolgreich installiert. Beim Reboot wird die neue Nutzer App durchgeführt. Das Verhalten entspricht dem Sollverhalten.

Als zweiten Test wird die eingebaute Funktion zur Korruption des Firmware Images verwendet (vgl. Abbildung 6.7). Dafür wird in der Nutzerapplikation die MPU deaktiviert und einige Bytes aus der Firmware gelöscht. Bei dem Neustart stellt der Secure Boot wie erwartet fest, dass die Signatur nicht mehr übereinstimmt. Die Nutzerapplikation wird gelöscht. Das Verhalten entspricht damit dem Sollverhalten.

```

===== Test: CORRUPT ACTIVE IMAGE =====

Corrupt image from SLOT_ACTIVE_1 ----- 1
Corrupt image from SLOT_ACTIVE_2 ----- 2
Corrupt image from SLOT_ACTIVE_3 ----- 3
Previous Menu ----- x

Selection :

-- Disable MPU protection to be able to erase
-- Erasing 0x20 bytes at address: 0x8010200
-- At next boot Signature Verification will fail. Download a new FW to re
store FW image !!

= [SBOOT] System Security Check successfully passed. Starting...

=====
= (C) COPYRIGHT 2017 STMicroelectronics =
= Secure Boot and Secure Firmware Update =
=====

= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
INFO: A Reboot has been triggered by a Software reset!
INFO: Last execution detected error was: No error. Success.
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: CHECK USER FW STATUS
Slot SLOT_ACTIVE_1 not empty : erasing ...

```

Abbildung 6.7: STM32F4 - SBSFU korruptes Firmware Image

Als dritten Test wird eine unverschlüsselte Datei gesendet. Die Authentifizierung des Headers (der Meta-Daten) schlägt fehl. Die Installation der Nutzerapplikation wird nicht durchgeführt und die Daten werden wieder aus dem Flash gelöscht. Das Verhalten entspricht dem Sollverhalten.

Beim vierten Test wird ein Update mit einem falschen Hash-Wert durchgeführt. Die Authentifizierung des Headers schlägt fehl. Die Nutzerapplikation wird aus dem Flash gelöscht. Das Verhalten entspricht dem Sollverhalten.

Bei fünften Test wird während des Updates ein Reset durchgeführt. Die Installation schlägt wie erwartet fehl. Beim Reboot werden die Dateireste im Slot der Nutzerapplikation gelöscht. Das Verhalten entspricht dem Sollverhalten.

Bei dem sechsten Test werden an die zu ladende Datei eine zusätzliche Bytes angehängt. Die Installation schlägt fehl, da die Hash-Überprüfung einen Fehler erkennt. Die Nutzerapplikation wird aus dem Flash gelöscht. Das Verhalten entspricht dem Sollverhalten.

Test der Write Protection Für den Test des Schreibschutzes wird zunächst der eingebaute Test verwendet (vgl. Abbildung 6.8). Hier wird versucht den geschützten Bereich zu überschreiben. Der Versuch wird blockiert und der Controller wird neu gestartet. Das Verhalten entspricht den Sollverhalten.

```
Test Protection: WRP ----- 4
Test Protection: IWDG ----- 5
Test Protection: TAMPER ----- 6
Previous Menu ----- x
Selection :

===== Test Protection: WRP =====
-- Reading 0x800 bytes at address: 0x8008900 (for backup)
-- Erasing 0x800 bytes at address: 0x8008900
- !! HAL_FLASH_ERROR_WRP: FLASH Write protected error flag !!
```

Abbildung 6.8: STM32F4 - SBSFU WRP Test

Als zweiten Test wird mit Hilfe der ST-Link Utility der Schreibschutz zurückgesetzt. Die Optionbytes können gesetzt werden. Bei Reboot erkennt der Secure Boot die falsche Konfiguration. Die Option Bytes werden wieder gesetzt und es wird ein Reboot durchgeführt. Das Verhalten entspricht dem Sollverhalten.

Test der MPU Beim Test der MPU werden die eingebauten Tests verwendet. Hierbei werden Software Angriffe simuliert, bei denen versucht wird die geschützten Bereiche auszulesen.

Beim ersten Test wird während der Durchführung der Nutzerapp versucht, den geheimen Schlüssel auszulesen (vgl. Abbildung 6.9). Die MPU löst wie erwartet einen Reboot aus. Der Schlüssel kann daher nicht gelesen werden. Das Verhalten entspricht somit dem Sollverhalten.

```

===== Test Menu =====
Test : CORRUPT ACTIVE IMAGE ----- 1
Test SE isolation - CODE ----- 2
Test SE isolation - VDATA ----- 3
Test Protection: WRP ----- 4
Test Protection: IWDG ----- 5
Test Protection: TAMPER ----- 6
Previous Menu ----- x
Selection :

===== Test Protection: MPU privileged - CODE =====
-- Reading Key

= [SB00T] System Security Check successfully passed. Starting...

=====
= (C) COPYRIGHT 2017 STMicroelectronics =
= Secure Boot and Secure Firmware Update =
=====

= [SB00T] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SB00T] STATE: CHECK STATUS ON RESET
INFO: A Reboot has been triggered by a Software reset!
INFO: Last execution detected error was: No error. Success.
= [SB00T] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SB00T] STATE: CHECK USER FW STATUS

```

Abbildung 6.9: STM32F4 - SBSFU MPU Test

Beim zweiten Test wird während der Durchführung der Nutzerapplikation versucht, den geschützten RAM-Bereich auszulesen. Die MPU löst einen Reboot aus. Der Speicherinhalt kann nicht gelesen werden. Das Verhalten entspricht dem Sollverhalten.

Test der Device ID Für den Test der Device ID Prüfung wird ein Secure Boot verwendet, bei dem die MPU deaktiviert ist. Bei einer richtigen ID wird die Nutzerapplikation normal durchgeführt. Um das Verhalten bei einer falschen ID zu überprüfen, wird mit Hilfe des Python Skriptes eine falsche ID in die Elf-Datei eingetragen. Die Software wird hält an. Das Verhalten entspricht dem Sollverhalten. Nach den Tests wurde die MPU wieder aktiviert.

Weitere Tests Ein weiterer Test untersucht das Verhalten des Tamper-Pins. Dieser ist mit dem Nutzer-Button des Nucleo-Boards verbunden. Wird der Button während des Secure Boot betätigt, wird ein Reset durchgeführt (vgl. Abbildung 6.10). Während der Nutzerapplikation erfolgt nur eine Ausgabe auf der Konsole. Es fällt dabei auf, dass der Tamper-Pin sehr empfindlich reagiert. Es reicht, wenn man den Tamper-Pin am Nucleo-Board mit dem Finger berührt, damit ein Manipulationsversuch erkannt wird.

```
===== Test Protection: TAMPER =====
-- Pull PC13 (CN8.8) to GND
-- -- Note: sometimes it may be enough to put your finger close to PC13 (CN8
)
-- Should reset if TAMPER is enabled.
Waiting for 10 seconds...

Tamper detected

-- TAMPER Event detected!!
-- System reset requested!!!

= [SBOOT] System Security Check successfully passed. Starting...

=====
= (C) COPYRIGHT 2017 STMicroelectronics =
= Secure Boot and Secure Firmware Update =
=====

= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
INFO: A Reboot has been triggered by a Software reset!
INFO: Last execution detected error was: No error. Success.
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: CHECK USER FW STATUS
A FW is detected in the slot SLOT_ACTIVE_1
= [SBOOT] STATE: VERIFY USER FW SIGNATURE
= [SBOOT] STATE: EXECUTE USER FIRMWARE
```

Abbildung 6.10: STM32F4 - SBSFU Tamper Pin Test

Eine weitere eingebaute Schutzmaßnahme ist der Watchdog-Counter. Wenn dieser nicht regelmäßig aktualisiert wird, wird ein Reset ausgelöst. Damit sollen Software-Angriffe weiter erschwert werden. Zum Testen wurde die eingebaute Funktion verwendet. (vgl. Abbildung 6.11). Das Verhalten entspricht dem Sollverhalten: Es wird ein Reset nach spätestens 10 Sekunden ausgelöst. Der Secure Boot meldet einen Fehler im Watchdog.

```

===== Test Protection: IWDG =====
-- Waiting 10000 (ms). Should reset if IWDG is enabled.

= [SBOOT] System Security Check successfully passed. Starting...

=====
=                                     =
=      (C) COPYRIGHT 2017 STMicroelectronics      =
=                                     =
=      Secure Boot and Secure Firmware Update      =
=====

= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
=          WARNING: A Reboot has been triggered by a Watchdog reset!
=          INFO: Last execution detected error was: Watchdog error.
= [EXCPT] WATCHDOG RESET FAULT!
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: CHECK USER FW STATUS
=          A FW is detected in the slot SLOT_ACTIVE_1
= [SBOOT] STATE: VERIFY USER FW SIGNATURE
= [SBOOT] STATE: EXECUTE USER FIRMWARE

```

Abbildung 6.11: STM32F4 - SBSFU IWDG Test

Untersuchung der Boot Dauer Die Zeitmessung des Secure Boot ergab im Durchschnitt folgende Zeiten:

Metadaten Signatur Prüfung	138,6 ms
FW Signatur Prüfung	42,74 ms
Secure Boot Dauer (Normalfall)	447,6 ms
Secure Boot ohne Debug Nachrichten	362,3 ms

Tabelle 6.1: STM32F4 - Boot Zeiten

6.2.5 Bewertung des Secure Boot

Ziel der Untersuchung ist es herauszufinden, ob der Secure Boot von ST wirklich 'secure' ist.

Schutz der Integrität und Authentizität Der Secure Boot prüft erfolgreich die Integrität und Authentizität der Nutzerapplikation. Bei sämtlichen Tests, in denen die Firmware manipuliert wird, schlägt die Integritätsprüfung fehl und die korrupte Nutzerapplikation wird von dem Controller entfernt.

Die Prüfung der Authentizität ist solange verlässlich, solange der öffentliche Schlüssel im Speicher nicht verändert werden. Dieser ist mit Hilfe von dem Schreibschutz (WRP) und der MPU vor Angriffen von innen geschützt. Selbst wenn ein Angreifer es schafft, den öffentlichen Schlüssel zu lesen, kann er daraus nicht den privaten Schlüssel berechnen.

Die Integritäts- und Authentizitätsprüfungen verhindern, dass Schadcode eingefügt wird, der die Firmware auslesen könnte.

Schutz der Update Datei Ein weiteres Angriffsziel ist die Update-Datei. Ein Angreifer könnte versuchen, diese Datei abzufangen und in seine gefälschte Hardware zu kopieren. Durch die Verschlüsselung mit dem AES wird die Vertraulichkeit sichergestellt, solange der Angreifer keinen Zugriff auf den Schlüssel hat. Von innen wird der Schlüssel mit Hilfe der MPU und des WRP geschützt. Die Readout Protection schützt diesen vor Angriffen von außen.

Schutz vor Glitches Um die Integritäts- und Authentizitätsprüfungen zu umgehen, kann Angreifer versuchen, Fehler in das System einzuführen und versuchen einen Glitch auszulösen. ST hat, um so etwas zu verhindern, die Software-Maßnahmen, die in Kapitel 5.1.3 beschrieben sind, verwendet. Dazu gehört das Einfügen von Redundanz bei kritischen Abfragen sowie das doppelte Durchführen von der Hash-Prüfung. Prüfvariablen haben eine größere Hamming-Distanz (vgl. Abbildung 6.12). Zum Flow Control wurden Variablen eingebaut, die beim Durchführen einer kritischen Funktion hochgezählt werden und am Ende überprüft werden. Das Durchführen von Glitch-Angriffen ist durch diese Maßnahmen stark erschwert worden.

```
typedef enum
{
    SE_ERROR = 0x00018799U,
    SE_SUCCESS = 0x0012310FU
} SE_ErrorStatus;
```

Abbildung 6.12: STM32F4 - SBSFU Sichere Variablen

Weitere Schutzmechanismen Der Watchdog ist in erster Linie dazu gedacht Deadlocks zu verhindern. Hier wird er dazu genutzt, Software-Angriffe weiter zu erschweren.

Ein Angreifer müsste in seinem Schadcode ein ebenfalls die Aktualisierung des Counters einbauen.

Der Tamper-Pin löst momentan nur einen Reset aus. In einem echten Gerät kann hier z.B. das Löschen von Schlüsseln eingebaut werden, um die Sicherheit weiter zu erhöhen.

Device ID Prüfung Die Device ID Prüfung funktioniert im Moment nur, wenn die MPU deaktiviert ist. Dadurch geht allerdings ein beträchtliches Maß an Sicherheit verloren. Des Weiteren fiel auf, dass in der Elf-Datei erkennbar ist, wo die Device ID Teile liegen. Durch eigenständige Festlegen der Speicheradressen liegen die Teile an Orten im Speicher, wo sonst keine Daten vorhanden sind. Es wäre besser die Teile direkt in dem Firmware Bereich zu verstecken oder den Speicher mit Dummy-Werten zu füllen. Die Device ID Prüfung hat in ihrer jetzigen Form mehr Nach- als Vorteile.

Fazit Die komplette Sicherheit des Secure Boot baut allerdings darauf auf, dass die Readout Protection nicht umgangen werden kann. Wie im Kapitel 5.1.1 beschrieben, gibt es allerdings durchaus Möglichkeiten diesen zu umgehen.

Alles in allem ist der Secure Boot gut in der Lage vor relativ einfachen Angriffen die Firmware zu schützen. Diese Angriffe beinhalten Software-Angriffe als auch einfache Glitch-Angriffe. Ein Angreifer mit hohem Budget, der in der Lage ist invasive Angriffe durchzuführen, wird allerdings nicht aufgehalten. Sollte Schutz vor invasiven Angriffen erwartet werden Mikrocontroller verwendet werden, die auf Security spezialisiert sind.

Weitere Möglichkeiten Es gibt Möglichkeiten den Schutz vor dem Kopieren weiter zu erhöhen. Die Device ID Prüfung könnte in den Secure Boot selbst verlagert werden. Dadurch können die Probleme mit der MPU vermieden werden. Die Device ID Prüfung kann weiter verbessert werden, indem Daten aus dem One-Time-Programmable-Bereich mit in den erwarteten Wert eingebaut werden. Des Weiteren kann in die Nutzerapplikation eine Art Handshake eingebaut werden, in dem geprüft wird, ob der Secure Boot durchlaufen wird. Ein offensichtliche Schutzmaßnahme ist es, einen Controller zu verwenden, der mehr Möglichkeiten zur Isolation des Security-Funktionen bietet.

Eine Möglichkeit die Readout Protection zu verstärken ist, dass man zusätzlich in der Software die Debug Pins umkonfiguriert. Dadurch kann bei der Level 1 Readout Protection ein Angriff auf den RAM erschwert werden.

6.3 Untersuchung des STM32L552ZE

6.3.1 Grundfunktionalität des Mikrocontrollers

Im Gegensatz zu dem F4-Controller bietet der STM32L552ZE mehr Möglichkeiten die Software zu isolieren. Neben den gleichen Security-Mechanismen des F4-Controllers besitzt der L5-Controller die TrustZone-Architektur. Mit dieser ist es möglich, den Speicher in einen Secure und einen unsecure Bereich aufzuteilen.

6.3.2 Funktionsumsetzung

Ähnlich wie bei dem F4-Controller wird hier bereits eine fertige Lösung für einen Secure Boot verwendet. Der L5-Controller besitzt seine eigene Implementierung: die Trusted Firmware-M (TF-M), die auf einem Standard von ARM basiert.

Das Programm ist in drei Teile aufgeteilt (vgl. Abbildung 6.13): Ein 'un-trusted'-Bereich für die Nutzerapplikation. Der Secure ('trusted') Bereich ist in einen privilegierten (Secure Boot) und unprivilegierten Bereich (im Moment ungenutzt) aufgeteilt. Es ist theoretisch möglich, den Bereich der Nutzerapplikation ebenfalls in einen privilegierten und unprivilegierten Bereich aufzuteilen, da der Secure und Unsecure-Bereich jeweils eine eigene MPU besitzen.

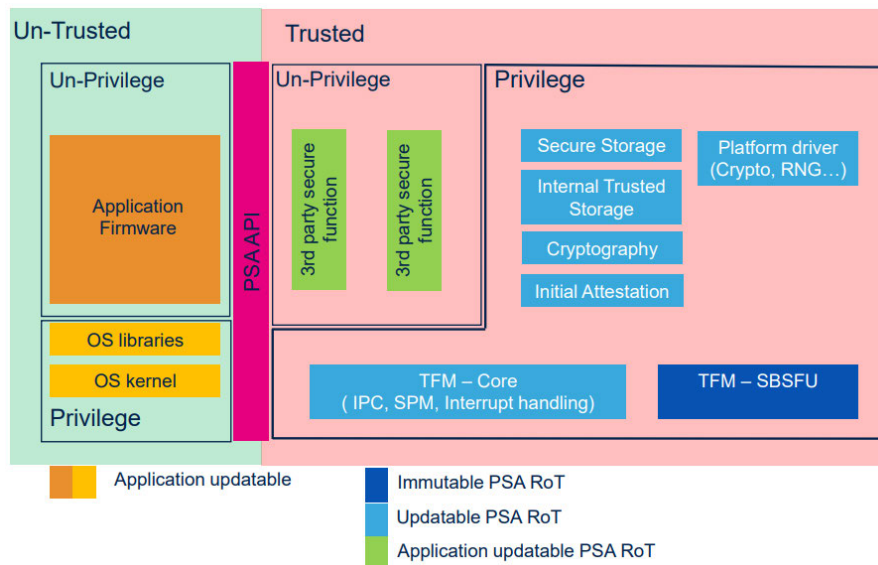


Abbildung 6.13: STM32L5 - Aufbau TF-M [28]

Zur Lagerung von Security-Daten werden zwei verschiedene sicheren Speicher angeboten: der Secure Storage (SST) Service und der Internal Trusted Storage (IST) Service. Beim Secure Storage Service werden Daten verschlüsselt in Speicher abgelegt. Beim Internal Trusted Storage Service werden Daten in einem Bereich des Flash-Speichers abgelegt, auf den unsichere oder unprivilegierte Applikationen keinen Zugriff haben.

6.3.3 Untersuchung des Secure Boot

Die Untersuchungen von dem F4-Controller wurden hier wiederholt. Die Schutzfunktionen lösten auch hier wie erwartet aus.

```

[INF] Starting bootloader
[INF] Initializing BL2 NU area : Power down/reset not supported...
[INF] Init BL2 NU Header area: Done
[INF] Initializing BL2 NU Counters
[INF] Init BL2 NU counters to 0 : Done
[INF] BL2 NU Area Initialized : Power Down/reset supported
[INF] Checking BL2 NU area
[INF] Checking BL2 NU area header
[INF] Checking BL2 NU Counter consistency
[INF] Consistent BL2 NU Counter 3 = 0x0
[INF] Consistent BL2 NU Counter 4 = 0x0
[INF] Swap type: none
[INF] Swap type: none
[INF] verify counter 0 1000000 0
[INF] counter 0 : ok
[INF] verify sig key id 0
[INF] signature OK
[INF] Counter 3 set to 0x1000000
[INF] verify counter 1 1000000 0
[INF] counter 1 : ok
[INF] verify sig key id 1
[INF] signature OK
[INF] Counter 4 set to 0x1000000
[INF] Bootloader chainload address offset: 0x14000
[INF] Jumping to the first image slot
[INF] BL2 HUK_STM32L652XX_HUK_CUSTOMIZATION_
set to BL2 SHARED DATA
[INF] Code c001900 c00ea7a
[INF] hash TFM_SBSFU_Boot 8b114fc3 .. 87de87c
[Sec Thread] Secure image initializing!

=====
=> (C) COPYRIGHT 2019 STMicroelectronics
=>
=> User App #A
=====

===== Main Menu =====

Test Protections ----- 1
Test TFM ----- 2
Download a new Fw Image ----- 3
Selection :

```

Abbildung 6.14: STM32L5 - TF-M Menü

Zusätzlich wurde untersucht, ob es möglich ist von der Nutzerapplikation aus den sicheren Bereich auszulesen. Das Auslesen führt zu einem Reboot. Das Sollverhalten wird damit erfüllt.

6.3.4 Bewertung des Secure Boot

Der Secure Boot mit der TrustZone bietet einen höheren Isolationslevel als der Secure Boot des F4-Controllern. Allerdings steigt damit auch der Entwicklungsaufwand.

Ein weiterer Vorteil des L5-Controllern ist, dass es möglich ist einen Boot-Lock zu setzen. Das bedeutet, man legt eine Adresse fest, in der der Controller immer startet. Wird der

Boot-Lock einmal gesetzt, kann er nicht mehr entfernt werden. Durch diese Option kann garantiert werden, dass der Secure Boot stets ausgeführt wird.

Einen weiteren Vorteil der TrustZone ist, dass sie nur deaktiviert werden kann, wenn auch gleichzeitig die Readout Protection deaktiviert wird. Dadurch wird garantiert, dass die Geheimnisse in dem sicheren Bereich nicht gelesen werden können. Im Gegensatz dazu kann bei dem F4-Controller die MPU deaktiviert werden, ohne dass die Geheimnisse gelöscht werden.

Die Nutzung der TrustZone ist um einiges komplexer als 'nur' die Nutzung der MPU. Es gibt bei der Entwicklung viel zu beachten. Insbesondere bei den Schnittstellen zwischen dem sicheren und unsicheren Bereich ist es einfach offene Türen für potentielle Angreifer einzubauen. Es gibt Funktionen, über die unsichere Funktionen auf Funktionen in der sicheren Welt zugreifen können. Wenn diese den Aufruf nicht ausreichend prüfen, können hier Sicherheitslücken entstehen.

Genau wie der F4-Controller bauen die Security-Funktionen auf der Readout Protection auf. Insgesamt bietet der L5-Controller einen noch stärkeren Schutz vor Software Angriffen. Ein invasiver Angriff wird auch hier nicht verhindert.

6.4 Angriff auf die Readout Protection

Im Rahmen der Recherche fiel eine Angriffsreihe auf die Readout Protection von ST besonders auf. Dabei handelt es sich um die Abhandlung von Obermeier und Tatschner, die drei verschiedene Angriffe auf die Readout Protection der STM32F0-Controller beschreibt (vgl. [19]).

6.4.1 Beschreibung der Angriffe

Cold-Boot Stepping Die erste Angriffsmöglichkeit bezieht sich auf den RDP Level 1. Im RDP Level 1 wird der Zugriff auf den Flash (sowohl für den Debugger als auch für den Mikrocontroller) gesperrt, nicht aber auf das RAM oder die Peripherie. Um die Angriffsfläche zu reduzieren, befinden sich sicherheitskritische Daten nur für kurze Zeit im RAM. Auch ist die konkrete Anordnung des Speichers (Memory Map) unbekannt. Der Angriff (Cold-Boot Stepping) wird durchgeführt, indem das System zu verschiedenen Zeiten (alle paar Taktzyklen) angehalten wird und der Inhalt der RAMs gespeichert

wird. Dadurch ist es möglich Nutzungsbereiche zu identifizieren (der grundsätzliche Programmfluss, Nutzung spezifischer Adressen und eventuell Schlüssel).

Ein Beispielangriff mit dieser Methode könnte wie folgt erfolgen: In der Praxis wird oft ein Bootloader verwendet, der beim Start des Controllers die Firmware mit Hilfe einer CRC-Checksumme überprüft wird. Das Ergebnis der CRC Berechnung wird im RAM zwischengespeichert. Ein Angreifer kann aus diesen Zwischenergebnissen, die er aus dem RAM mit Hilfe des Angriff ausgelesen hat, die Firmware rekonstruieren.

Security Downgrade Die zweite beschriebene Angriffsmöglichkeit bezieht sich auf den RDP Level 2. Das Problem liegt in der Umsetzung der RDP Level. Die Level 0 und 2 haben feste Werte (0x55AA und 0x33CC). Die restlichen möglichen Werte bilden den RDP Level 1. Um also den Level von 2 auf 1 runterzusetzen, muss nur ein Bit gekippt werden. Des Weiteren ist es egal, welches Bit gekippt wird. [19] hat gezeigt, dass dies durch den Einsatz von UV Licht möglich ist. Der Nachteil des Angriffs ist, dass auch Bits von der Firmware kippen können. Durch Analyse des Flash-Layouts können Masken erstellt werden, die diese Bereiche schützen. Die Durchführbarkeit des Angriffs hängt also davon ab, die der Flash physikalisch umgesetzt wurde. Das kann sich von Controller zu Controller unterscheiden. Physisch ist es schwer, den Angriff zu verhindern. In Software können regelmäßige Abfragen der Register (sowohl im FLASH_OBR Register als auch die Option Bytes selbst).

Debug Interface Exploit Die dritte Angriffsmöglichkeit betrifft ebenfalls den RDP Level 1. Hierbei wird die SWD-Schnittstelle genutzt. Für diesen Angriff kann nicht ST-Link genutzt werden. Obermeier und Tatschner haben dafür eine eigene SWD - Implementierung auf einem STM32F0-Controller umgesetzt. Die Autoren haben untersucht, wann der Schutz (also der Debug-Modus) aktiviert wird. Es stellte sich heraus, dass der Controller erst gestoppt wird, wenn SWD auf den System-Bus zugreift. Versucht man über SWD auf den Flash vor der Schutzaktivierung auszulesen, kann es passieren, dass ab und zu im Read-Buffer Daten aus dem geschützten Bereich stehen.

Grund dafür ist, dass der Ausleseschutz zwei Taktzyklen braucht, um aktiviert zu werden. Diese Lücke kann genutzt werden, um den ganzen Flash auszulesen, indem bei jedem Controllerstart je 32 Bit ausgelesen werden. Die Autoren erzielten eine durchschnittliche Auslesegeschwindigkeit von 45 Byte pro Sekunde. Ein Flash-Speicher mit 256 kByte kann mit dieser Methode innerhalb von zwei Stunden ausgelesen werden

Der erste und der dritte Angriff lassen sich verhindern, indem die Level 2 Readout Protection aktiviert wird. Der zweite Angriff hingegen lässt sich nicht verhindern.

6.4.2 Versuchsdurchführung

Die Kapitel 5.2.1 und 6.4.1 zeigen, dass es durchaus möglich ist die Readout Protection von Level auf Level 1 herunterzusetzen. Aus diesem Grund ist von den drei vorgestellten Angriffen ist insbesondere der dritte interessant. Dieser erlaubt trotz aktivierter Readout Protection das direkte Auslesen des Flash-Speichers. Daher soll überprüft werden, ob die Schwäche auch bei den F4- und L5-Controller vorhanden ist.

Zunächst wird der Angriff auf einem F0-Controller wiederholt, um zu überprüfen, ob der Angriff überhaupt noch durchführbar ist. Dazu wurden zwei bereits in der Firma vorhandene F0-Boards verwendet. Abbildung 6.15 zeigt den Versuchsaufbau.

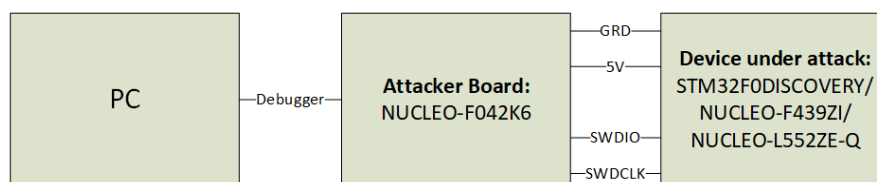


Abbildung 6.15: Aufbau Readout Protection Angriff

Bei dem F0-Controller kann der Angriff mit geringem Aufwand erfolgreich nachgestellt werden. Es ist möglich den gesamten Speicher trotz aktivierten Readout Protection auszulesen. Bei den F4- und L5-Controller hingegen scheitert der Angriff. Hier wird die Readout Protection ordnungsgemäß.

Die anderen beiden Angriffe hingegen können bei allen Mikrocontrollern von ST durchgeführt werden.

6.5 Auswertung

Beide Untersuchungen zeigen, dass man mit General Purpose Mikrocontrollern in der Lage ist, einfache Angriffe, wie Software Angriffe und einfache Glitch-Angriffe, abweh-

ren kann. Allerdings setzen beide Controller voraus, dass die Readout Protection nicht umgangen werden kann.

Sollte ein Angreifer in der Lage sein, die Readout Protection von Level 2 auf Level 1 herunterzusetzen, so könnte es ihm möglich sein, aus dem RAM geheime Informationen herauszufinden. Ein solcher Angriff kann jedoch erschwert werden. Zum einen kann zusätzlich zur Readout Protection eine Umkonfigurierung der Debug-Pins zu Beginn der Software durchgeführt werden. Zum anderen kann in der Nutzerapplikation eine regelmäßige Prüfung der Option Bytes durchgeführt werden.

Alles in allem bietet der Secure Boot bei beiden Controllern eine gute Möglichkeit die Firmware vor dem Kopieren zu schützen.

7 Ausblick und Fazit

7.1 Ausblick

7.1.1 Ausblick auf weitere Schutzmöglichkeiten

Mikrocontroller anderer Firmen Neben ST-Controller bieten auch andere Firmen bereits eingebaute Security-Features bei ihren General Purpose Mikrocontrollern. Viele Security-Features kommen bei allen Mikrocontrollern vor. Dazu gehört ein Ausleseschutz für den Debugger, einen Schreibschutz, eine MPU und ein Debugger.

Firmen wie NXP und Maxim Integrated im Gegensatz zu ST auch Mikrocontroller mit weiteren Schutzfunktionen an. Insbesondere Mikrocontroller von NXP bieten mehr Controller an, die mehr Möglichkeiten für Kryptografie bieten. So werden zum Beispiel mehr Hardware-Beschleunigungen für verschiedene Verschlüsselungsverfahren angeboten. Des Weiteren besitzen einige Controller einen eingebauten PUF, um Schlüssel sicher zu speichern.

Grundsätzlich fällt auf, dass immer Mikrocontroller angeboten werden, die für IoT ausgerichtet sind. Daher besitzen immer mehr Controller die TrustZone Architektur. In Zukunft werden durch den Trend zu IoT immer mehr General Purpose Controller auftauchen, die mehr Security Funktionen anbieten.

Weitere Schutzmöglichkeiten Viele recherchierten Schutzmöglichkeiten gegen Produktpiraterie basieren darauf in die Software Hardware Abhängigkeiten (wie z.B. die Device ID oder PUFs) einzubauen. Eine solche Software kann nicht mehr ohne weiteres auf ein neues Gerät kopiert werden. Die Hardware-Abhängigkeiten werden nicht nur einfach abgefragt, sondern die Software wird mit deren Hilfe in der einen oder anderen Form verschleiert. Damit wird dafür gesorgt, dass die Software auf einer gefälschten Hardware nicht mehr ausgeführt werden kann. Beispiele für solche Methoden zeigen [14] und [9].

7.1.2 Ausblick auf weitere Angriffe

Bei der Recherche der Angriffe fiel auf, dass immer mehr Hardware entwickelt wird, die Glitch-Angriffe vereinfachen soll. Man kann davon ausgehen, dass in Zukunft das Einfügen von Fehlern in einem System immer einfacher werden wird.

So wurden auch immer mehr Forschungsarbeiten entdeckt, die versuchen Seitenkanalangriffe mit Hilfe von maschinellem Lernen zu automatisieren. Diese sollen z.B. automatisch aus einem Stromverlauf den Schlüssel auslesen (vgl. [5]). Es kann also davon ausgegangen werden, dass in Zukunft auch Seitenkanalangriffe weiter vereinfacht werden.

7.2 Fazit

In dieser Thesis wurden zahlreiche Methoden gezeigt, mit denen ein General Purpose Mikrocontroller geschützt werden. Es wurden dabei primär Möglichkeiten untersucht, die keine zusätzliche Hardware benötigen.

So ist der Hauptbestandteil, um einen Mikrocontroller zu schützen, die Isolation der Softwareteile voneinander ist. Insbesondere Kommunikationstack und Security-Funktionen sollten voneinander getrennt sein. Diese Isolation kann bereits ausreichend mit General Purpose Controllern erreicht werden. Es ist mit diesen Controllern möglich sich gut gegen Software- und einfache Glitch-Angriffe zu schützen.

Die einzelnen Schutzmechanismen für sich allein sind in der Regel leicht angreifbar. Wenn diese jedoch miteinander kombiniert werden, können höhere Sicherheitslevel erreicht werden. Das bedeutet allerdings auch, dass wenn ein Security-Mechanismus in der Chain of Trust umgangen wird, so werden auch die Mechanismen, die darauf aufbauen umgangen.

Als Root of Trust wird oft die Readout Protection verwendet. Angreifer mit einem geringem Budget können mit dieser gut aufgehalten werden. Arbeiten wie [19] zeigen, dass es jedoch bei einigen Controller konzeptionelle Fehler gibt, die einen solchen Angriff stark vereinfachen und erlauben die Firmware ohne Probleme auszulesen. Diese Controller sollten entsprechend nicht verwendet werden.

Literaturverzeichnis

- [1] ARM: *Using TrustZone on Armv8-M*. 2019. – URL https://www.keil.com/appnotes/files/apnt_291.pdf. – Zugriffsdatum: 2021-09-13
- [2] BAR-EL, H. ; CHOUKRI, H. ; NACCACHE, D. ; TUNSTALL, M. ; WHELAN, C.: The Sorcerer's Apprentice Guide to Fault Attacks. In: *Proceedings of the IEEE* 94 (2006), Nr. 2, S. 370–382
- [3] BARENGHI, Alessandro ; BREVEGLIERI, L. ; REGAZZONI, F.: *Low-Cost Software Countermeasures Against Fault Attacks : Implementation and Performances Trade Offs*, 2010
- [4] BOZZATO, C. ; FOCARDI, R. ; PALMARINI, Francesco: Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019 (2019), S. 199–224
- [5] BRISFORS, Martin ; FORSMARK, Sebastian: *Deep-Learning Side-Channel Attacks on AES*. (2019)
- [6] BUCHMANN, Johannes: *Einführung in die Kryptographie*. 5. The address : Springer-Verlag Berlin Heidelberg, 2010. – ISBN 978-3-642-11186-0
- [7] CURTIS, Paul ; BERKENKOPF, Katrin: *Fighting product clones through digital signatures*. 2017. – URL <https://www.segger.com/downloads/emsecure/WP12001>. – Zugriffsdatum: 2020-10-23
- [8] FALSE ; INFORMATION misleading: *STM32 Device Electronic Signature – Unique Device ID Register*. 06 2012. – URL <http://false.ekta.is/2012/06/stm32-device-electronic-signature-unique-device-id-register/>. – Zugriffsdatum: 2020-09-22
- [9] GUIN, Ujjwal ; BHUNIA, Swarup ; FORTE, Domenic ; TEHRANIPOOR, Mark M.: *SMA: A System-Level Mutual Authentication for Protecting Electronic Hardware*

- and Firmware. In: *IEEE Transactions on Dependable and Secure Computing* 14 (2017), Nr. 3, S. 265–278
- [10] H2LAB: *Harden embedded systems part 1: Fault-resilient C programming*. 2020. – URL https://h2lab.org/blogposts/dev_c_fia/. – Zugriffsdatum: 2021-09-12
- [11] ITOOKTHEREDPILL: *Low-cost attacks on STM8 readout protection*. 06 2020. – URL <https://itooktheredpill.irgendwo.org/2020/stm8-readout-protection/>. – Zugriffsdatum: 2020-10-15
- [12] KORAK, Thomas ; HOEFLER, Michael: On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms. In: *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2014, S. 8–17
- [13] LESJAK, Christian ; HEIN, Daniel ; WINTER, Johannes: Hardware-security technologies for industrial IoT: TrustZone and security controller. In: *IECON 2015 - 41st Annual Conference of the IEEE Industrial Electronics Society*, 2015, S. 002589–002595
- [14] MAHMOD, Md Jubayer a.: Towards Unclonable System Design for Resource-Constrained Application. (2019), 07
- [15] NEDOSPASOV, Dmitry ; DATKO, Josh ; ROTH, Thomas: *Wallet.fail*. 2018. – URL <https://wallet.fail/>. – Zugriffsdatum: 2020-09-18
- [16] NUVOTON: *Software Countermeasures for Fault Injection Attacks*. – URL <https://www.nuvoton.com/support/technical-support/technical-articles/TSNuvotonTechBlog-000154/>. – Zugriffsdatum: 2021-09-12
- [17] OBERLI, Nicolas: *Physically Unclonable Functions in Practice*. 04 2019. – URL <https://research.kudelskisecurity.com/2019/04/11/physically-unclonable-functions-in-practice/>. – Zugriffsdatum: 2020-10-13
- [18] OBERMAIER, Johannes ; SCHINK, Marc: *Exception(al) Failure - Breaking the STM32F1 Read-Out Protection*. 03 2020. – URL <https://blog.zapb.de/stm32f1-exceptional-failure/>. – Zugriffsdatum: 2020-09-03
- [19] OBERMAIER, Johannes ; TATSCHNER, Stefan: Shedding too much Light on a Microcontroller’s Firmware Protection. In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC : USENIX Association,

2017. – URL <https://www.usenix.org/conference/woot17/workshop-program/presentation/obermaier>. – Zugriffsdatum: 2020-04-30
- [20] ROTH, Thomas: *TrustZone-M(eh): Breaking ARMv8-M's security*. 12 2019. – URL https://media.ccc.de/v/36c3-10859-trustzone-m_eh_breaking_armv8-m_s_security#t=61. – Zugriffsdatum: 2020-09-21
- [21] SCHINK, Marc ; OBERMAIER, Johannes: Taking a Look into Execute-Only Memory. In: *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA : USENIX Association, August 2019. – URL <https://www.usenix.org/conference/woot19/presentation/schink>
- [22] SCHUSTER, Tino: *Secure Secret Sharing*. 2013/14. – URL <https://pdfs.semanticscholar.org/76a7/1c89c5d2476960a46ee6fd052e259fdc4b83.pdf>. – Zugriffsdatum: 2020-10-01
- [23] SKOROBOGATOV, Sergei P.: *Copy Protection in Modern Microcontrollers*. 2000. – URL https://www.cl.cam.ac.uk/~sps32/mcu_lock.html. – Zugriffsdatum: 2021-09-11
- [24] STMICROELECTRONICS: *STM32L4 - Device Electronic Signature*. – URL https://www.st.com/content/ccc/resource/training/technical/product_training/b7/03/bd/99/54/55/4d/47/STM32L4_System_eSign.pdf/files/STM32L4_System_eSign.pdf/jcr:content/translations/en.STM32L4_System_eSign.pdf. – Zugriffsdatum: 2020-09-22
- [25] STMICROELECTRONICS: AN5056 Integration guide for the X-CUBE-SBSFU STM32Cube Expansion Package, 09 2020
- [26] STMICROELECTRONICS: AN5156 Introduction to STM32 microcontrollers security, URL https://www.st.com/resource/en/application_note/dm00493651-introduction-to-stm32-microcontrollers-security-stmicroelectronics.pdf. – Zugriffsdatum: 2020-09-02, 02 2020
- [27] STMICROELECTRONICS: *STM32 Security Workshop - 03 SBSFU presentation*. 2020
- [28] STMICROELECTRONICS: *STM32 Security Workshop - 08 PSA-TFM presentation*. 2020

- [29] STMICROELECTRONICS: UM2262 Getting started with the X-CUBE-SBSFU STM32Cube Expansion Package, 09 2020
- [30] WANG, Huanyu ; FORTE, Domenic ; TEHRANIPOOR, Mark M. ; SHI, Qihang: Probing Attacks on Integrated Circuits: Challenges and Research Opportunities. In: *IEEE Design Test* 34 (2017), Nr. 5, S. 63–71
- [31] WITTEMAN, Marc: Secure application programming in the presence of side channel attacks. (2017). – URL https://www.riscure.com/uploads/2018/11/201708_Riscure_Whitepaper_Side_Channel_Patterns.pdf. – Zugriffsdatum: 2021-09-12
- [32] YUCE, Bilgiday ; SCHAU MONT, Patrick ; WITTEMAN, Marc: Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. In: *Journal of Hardware and Systems Security* 2 (2018), May, Nr. 2, S. 111–130. – URL <http://dx.doi.org/10.1007/s41635-018-0038-1>. – ISSN 2509-3436
- [33] ZIMMER, Vincent ; KRAU, Michael: ESTABLISHING THE ROOT OF TRUST, 08 2016

A Anhang

A.1 Python Skript zum Einfügen der Device ID in eine Elf-Datei

```
#
→ #####
→
# @file
#
# @brief      Change Device ID in ELF-File
#
# @details    This script changes the rxpected Device ID in the
→ ELF-file.
#           The Device ID is split into three parts. Part 1
→ and 2 are
#           generated randomly. Part 3 is calculated.
#
#           **Script invocation:**
#
#           @verbatim
#
#           Cmd      Argument 1          2
#           python ChangeDeviceID.py  f
#
#           with
#           python = Command for Python execution
#           ChangeDeviceID.py = Name of this script
#           f = ELF file
```

```
# @endverbatim
#
# **Restrictions:**
#
# - No sanity checks on the arguments are performed
# - Implemented and tested with Python 3.7.6
# - Mandatory ELF format:
# @verbatim
# Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00
# ↪ 00 00
# Class: ELF32
# Data: 2's complement
# ↪ , little endian
# Version: 1 (current)
# OS/ABI: UNIX - System
# ↪ V
# ABI Version: 0
# Type: EXEC (
# ↪ Executable file)
# Machine: ARM
# Version: 0x1
# @endverbatim
#
# This script basically does 3 things:
# 1. Read ELF file , determine the file image offset
# ↪ of the first
# program header table entry and the number of
# ↪ entries from the
# ELF header and extract the relevant data of each
# ↪ segment
# (physical address , file image offset and file
# ↪ memory size)
# 3. Add device ID.
#
```

```
#
# ↪ #####
# ↪
# <b>HISTORY:</b><br>
#
# @version      1.00
# @date         2020-12-18
# @authors      E. Schmidt, Reese+Thies Industrieelektronik GmbH
# @brief        @li Initial Release
#
# ↪ #####
# ↪

%%% Imports
import sys
import array
import struct
import random

%%% Constants

## New Device ID
NEW_DEVICE_ID = [0x4E003E, 0x31375119, 0x39373236]

## First argument: ELF file
ARGUMENT_POSITION_ELFFILE = 1

## Address first part of the device ID – see map-File for
# ↪ details
ADDRESS_UID_PART_1 = 0x080209F4

## Address second part of the device ID – see map-File for
# ↪ details
ADDRESS_UID_PART_2 = 0x08022310
```

```

## Address third part of the device ID – see map-File for
    ↪ details
ADDRESS_UID_PART_3 = 0x0802BA04

## ELF file image offset of the pointer to the start of the
    ↪ program header
# table
ELF_E_PHOFF_OFFSET = 0x34

## Size of a program header table entry
ELF_E_PHENTSIZE = 0x20

## Offset within a program header entry of the offset of the
    ↪ segment in
# the file image
ELF_P_OFFSET_OFFSET = 0x04

## Offset within a program header entry of the segment's
    ↪ physical start
# address
ELF_P_PADDR_OFFSET = 0x0C

## Offset within a program header entry of the segment's size in
    ↪ the
# file image
ELF_P_FILESZ_OFFSET = 0x10

### Functions

# ELF functions

## Get the number of program header table entries
# @param[in] f File image
# @return Number of program header table entries
def elf_get_number_of_program_header_table_entries(f):
    f.seek(0x2C)

```

```
return int.from_bytes(f.read(2), 'little')

## Extract relevant data of each program header table entry
# @param[in] f File image
# @param[in] e_phnum Number of program header table entries
# @return Program header table entries with relevant data
def elf_extract_program_header_table_entries(f, e_phnum):
    program_headers = [dict() for i in range(e_phnum)]

    for i in range(e_phnum):
        ph_entry_offset = ELF_E_PHOFF_OFFSET + i *
            ↪ ELF_E_PHEXTSIZE
        f.seek(ph_entry_offset + ELF_P_OFFSET_OFFSET)
        program_headers[i]['p_offset'] = int.from_bytes(f.read
            ↪ (4), 'little')

        f.seek(ph_entry_offset + ELF_P_PADDR_OFFSET)
        program_headers[i]['p_paddr'] = int.from_bytes(f.read(4)
            ↪ , 'little')

        f.seek(ph_entry_offset + ELF_P_FILESZ_OFFSET)
        program_headers[i]['p_filesz'] = int.from_bytes(f.read
            ↪ (4), 'little')

    return program_headers

## Get file image offset and segment index of an address
# @param[in] program_headers Extracted program headers
# @param[in] address Address
# @return File image address offset
def elf_get_address_data(program_headers, address):
    for i in range(len(program_headers)):
        start_address = program_headers[i]['p_paddr']
        end_address = \
            program_headers[i]['p_paddr'] + program_headers[i]['
            ↪ p_filesz'] - 1
```

```
        if address >= start_address and address <= end_address:
            relative_address_offset = address - start_address
            return \
                [program_headers[i]['p_offset'] +
                 ↪ relative_address_offset, i]

    return None

## Write Device ID to file
# @param[in,out] f File image
# @param[in] crc_offset CRC offset
# @param[in] crc CRC
def write_UID(f, id_offset, deviceID):
    f.seek(id_offset)
    f.write(bytes(struct.unpack("4B", struct.pack("L", deviceID
        ↪ [0]))))
    f.write(bytes(struct.unpack("4B", struct.pack("L", deviceID
        ↪ [1]))))
    f.write(bytes(struct.unpack("4B", struct.pack("L", deviceID
        ↪ [2]))))

##### Main application
↪ #####

print("\nChange_Device_ID_in_ELF-File ... \n")
deviceID_part1 = [0xAF15AFD8, 0x1B56904B, 0x3DF36453]
deviceID_part2 = [0x06BC6A80, 0x0F212DC7, 0xBAFFFD49]
deviceID_part3 = [0xA9E7C566, 0x2540EC95, 0xBE3BAB2C]

# Open ELF file
f = open(sys.argv[ARGUMENT_POSITION_ELFFILE], 'r+b')

# Get file image position of the program header table and
↪ extract
```

```
# program header data
e_phnum = elf_get_number_of_program_header_table_entries(f)
program_headers = elf_extract_program_header_table_entries(f,
    ↪ e_phnum)

# Calculate Device ID parts
#for i in range(3):
#    deviceID_part1[i] = random.getrandbits(32)
#    deviceID_part2[i] = random.getrandbits(32)
#    deviceID_part3[i] = NEW_DEVICE_ID[0] ^ deviceID_part1[i] ^
    ↪ deviceID_part2[i]

# Write Parts into ELF file
write_UID(f,
    elf_get_address_data(program_headers,
        ↪ ADDRESS_UID_PART_1)[0] , \
    deviceID_part1)
write_UID(f,
    elf_get_address_data(program_headers,
        ↪ ADDRESS_UID_PART_2)[0] , \
    deviceID_part2)
write_UID(f,
    elf_get_address_data(program_headers,
        ↪ ADDRESS_UID_PART_3)[0] , \
    deviceID_part3)

f.close()

print("..._done\n")
```

Listing A.1: Python Skript zum Einfügen der Device ID in eine Elf-Datei

A.2 CD Ordnerstruktur

Abbildung A.1 zeigt die Ordnerstruktur der beigelegten CD.

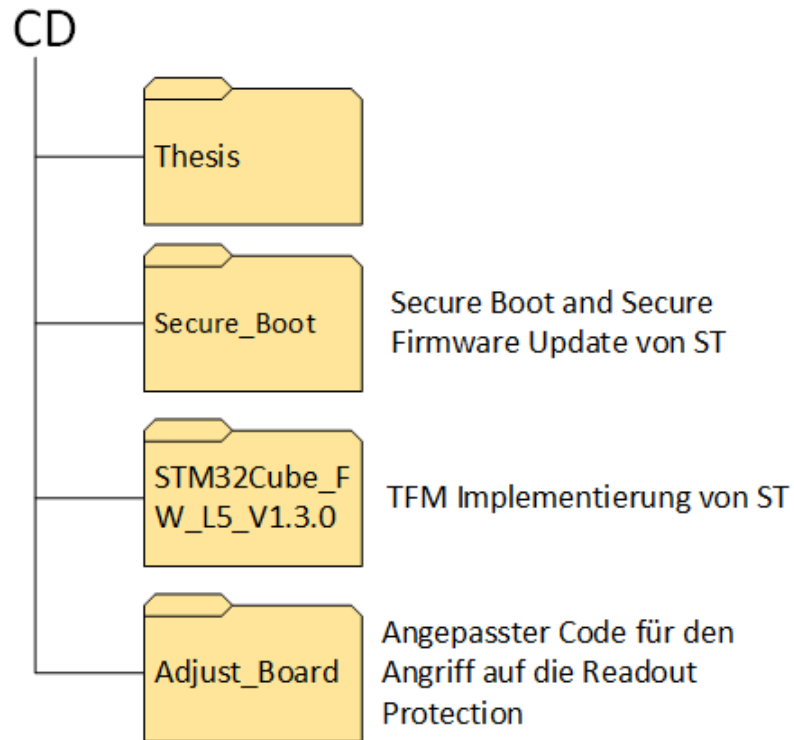
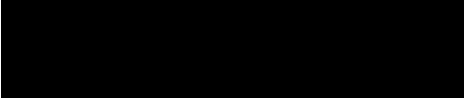


Abbildung A.1: DVD-Struktur

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

		
Ort	Datum	Unterschrift im Original