NTNU – Trondheim
Norwegian University of
Science and Technology

Securing the boot process of embedded Linux systems

Nahom Aseged Belay

Submission date: June 2022
Supervisor: Danilo Gligoroski, NTNU
Co-supervisor: Sven Schwermer, Disruptive Technologies

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title:	Securing the boot process of embedded Linux systems
Student:	Nahom Aseged Belay

Problem description:

Internet of Things devices have had a growing impact on our daily lives as more smart devices are occupying crucial roles in domains such as health, agriculture and energy. Despite their success, they are facing major challenges when it comes to security and data privacy. This makes researching and resolving the security concerns of Internet of Things critical.

This thesis will focus on securing the boot time process of an embedded Linux system by guaranteeing the authenticity and integrity of the software loaded during the initial boot sequence. It will be pursued in cooperation with Disruptive Technologies, a Norwegian technology company known for developing the world's smallest sensors and for its Internet of Things infrastructure.

The aim of this thesis is securing the boot sequence of an embedded Linux system running on an i.MX 7Dual processor ARM System-on-Chips processor by studying and implementing different security protocols. Activities may include implementing the secure boot protocol, evaluating and implementing the measured boot protocol using an external Trusted Platform Module, discussing secure firmware updates and carrying out different attacks to evaluate the robustness of our implementations.

Date approved:	2022-02-15
Responsible professor:	Danilo Gligoroski, NTNU
Supervisor(s):	Sven Schwermer, Disruptive Technologies

Abstract

The number of connected devices that make up the Internet of Things has risen dramatically in recent years and is showing no signs of slowing down. Furthermore, these devices play a vital role in critical sectors which makes guaranteeing that only trusted code is running on them evermore crucial. The goal of this research is to investigate how securing the boot process of embedded devices could prevent unauthorised code from being executed. This thesis focused on a particular System-on-Chip and its application in a chosen Internet of Things infrastructure to understand the overall impact resulting from its exploitation. A study of the secure boot protocol was made to determine if it is the most adequate solution for our problem. The requirements of the protocol were evaluated through implementations and literature review. The fault injection hardware attack was used to evaluate the robustness of the secure boot protocol. Our results show that the secure boot protocol is not the ideal solution when it comes to securing the boot process of embedded devices as the security guarantees are not as solid as one might expect. Further research is needed to explore other protocols such as the measured boot to determine if running unauthorised code can be entirely prevented by securing the boot process.

Sammendrag

I løpet av de siste årene har antallet tilknyttede enheter som utgjør *Internet of Things* økt dramatisk, og lite tyder på at denne utviklingen er i ferd med å snu. Ettersom disse enhetene utgjør en vesentlig del av kritiske sektorer i næringslivet er det essensielt å kunne garantere at de kun kjører pålitelig kode. Målet med denne studien er å undersøke hvordan kjøring av ikke-autorisert kode kan forhindres gjennom sikring av oppstartsprosessen i innebygde systemer. For å forstå virkningen av utnyttelse av en tilknyttet enhet har denne avhandlingen fokusert på en spesifikk *System-on-Chip* og dens bruk i en valgt *Internet of Things*-infrastruktur. En undersøkelse av *the secure boot protocol* ble gjennomført for å vurdere om den er en passende løsning på problemet. Kravene til protokollen ble vurdert gjennom litteraturstudier og implementasjoner av protokollen. Injeksjon av feil i systemet gjennom et maskinvareangrep ble gjennomført for å evaluere hvor robust protokollen er. Resultatene våre viser at *the secure boot protocol* ikke er en ideell løsning for å sikre oppstartsprosessen til innebygde systemer. Dette er grunnet at protokollen i realiteten gir lavere sikkerhetsgarantier enn først forventet. Videre forskning på andre protokoller, som for eksempel *the measured boot*, er nødvendig for å avgjøre om kjøring av ikke-autorisert kode kan forhindres gjennom sikring av oppstartsprosessen.

Preface

This research was undertaken at the Department of Information Security and Communication Technology at the Norwegian University of Science and Technology (NTNU) as part of the final project of a Master of Science in Communication Technology.

The study conducted in collaboration with Disruptive Technologies was carried out by Nahom Aseged Belay. The work was supervised by Sven Schwermer from Disruptive Technologies. Danilo Gligoroski was the responsible professor at NTNU.

Acknowledgements

I would like to thank Disruptive Technologies for allowing me to work on an interesting research project and providing me with all the necessary resources. I am very grateful for the continuous support and guidance I received from Sven Schwermer throughout my project. Thank you Gunnar Bolme for taking the time to help with the electronics portion of my project. I would also like to thank Danilo Gligoroski.

I am very grateful to NTNU and Institut National des Sciences Appliquées (INSA) Toulouse for allowing me to take part in their joint-master's program and thank you to all professors from both universities.

I would like to express my gratitude to all of the friends I made during my studies, I am grateful to each and every one of you.

I would like to thank Endre Bruaset, my roommate and an amazing friend, for translating by abstract into Norwegian.

A special thanks to Théau André Pierre Giraud for embarking on this joint-master's program with me and making it memorable. Thank you for your insightful comments on my thesis and for being such a wonderful friend.

Lastly, I would like to thank my parents and my two sisters for their unconditional love and support throughout the years.

Contents

List of Figures

List of Tables

List of Acronyms

AES Advanced Encryption Standard.

API Application Programming Interface.

CA Certification Authority.

CAAM Cryptographic Acceleration and Assurance Module.

CIA Confidentiality, Integrity and Authenticity.

CMOS Complementary Metal–Oxide–Semiconductor.

CRL Certificate Revocation List.

CRTM Core Root of Trust for Measurement.

CSF Command Sequence File.

CSFK Command Sequence File Key.

CST Code Signing Tool.

DDoS Distributed Denial-of-Service.

DES Data Encryption Standard.

DoS Denial of Service.

DPA Differential Power Analysis.

DRAM Dynamic Random Access Memory.

ECC Elliptic Curve Cryptography.

ECDH Elliptic Curve Diffie–Hellman.

ECDSA Elliptic Curve Digital Signature Algorithm.

eFuse electronic Fuse.

FPGA Field-Programmable Gate Array.

HAB High Assurance Boot.

initramfs initial Random Access Memory filesystem.

INSA Institut National des Sciences Appliquées.

IoT Internet of Things.

IVT Image Vector Table.

MMC MultiMediaCard.

MMU Memory Management Unit.

mTLS mutual Transport Layer Authentication.

NTNU Norwegian University of Science and Technology.

OSCP Online Certificate Status Protocol.

OSI Open Systems Interconnection.

OTA Over-The-Air.

PCB Printed Circuit Board.

PCR Platform Configuration Registers.

PKCS Public Key Cryptography Standards.

PKI Public-Key Infrastructure.

POR Power-On Reset.

ROM Read-Only Memory.

rootfs root filesystem.

RSA Rivest–Shamir–Adleman.

RVT Read-Only Memory Vector Table.

SDP Serial Download Protocol.

SHA Secure Hashing Algorithm.

SoC System-on-Chip.

SPA Simple Power Analysis.

SPL Secondary Program Loader.

SRAM Static Random Access Memory.

SRK Super Root Key.

SSH Secure Shell.

SSL Secure Socket Layer.

STRIDE Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.

TCG Trusted Computing Group.

TLS Transport Layer Authentication.

TPM Trusted Platform Module.

USB Universal Serial Bus.

Chapter 1
Introduction

The Internet of Things (IoT) is made up of a diverse set of devices that communicate with one another to collect and share data. The amount of connected devices has seen a dramatic increase in the last couple of years and this trend is likely to continue. IoT Analytics predicts that by 2025, more than 27.1 billion IoT devices will be online [Sin21].

Despite their success, there is growing concern over their security due to the heterogeneity of the devices and their limits in terms of energy, communication, computation and storage capabilities [MCZ+19].

When using a connected device, it is crucial to guarantee that only authorised code is running on the device. This guarantee is as important to the user as it is to the manufacturer as it allows them to trust that the device is behaving as intended. For instance, an attacker could install malicious software on a device to retrieve sensitive user data or paralyse a device rendering it useless.

The first step in ensuring that only trustworthy code is running on a device is securing its boot process. Furthermore, taking control of the device early on in the boot process allows an attacker to operate undetected and with greater privilege, highlighting the need to secure the booting sequence.

1.1 Context

Disruptive Technologies is a Norwegian technology company best known for developing the world's smallest sensors and their IoT infrastructure [Dis]. They are also renowned for their SecureDataShot™protocol which provides a secure communication between their sensors and their cloud through end-to-end encryption. Figure 1.1 is a simplified illustration of their their IoT infrastructure. The different sensors measure data such as temperature, motion, humidity, *etc.* and which they transmit to a cloud storage through a cloud connector.

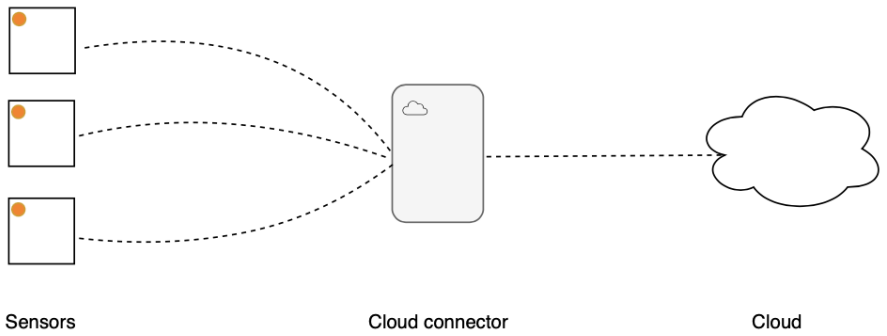


Figure 1.1: Disruptive Technologies' infrastructure (simplified)

This thesis will focus on the cloud connector which is a an embedded Linux system running on an i.MX 7Dual processor ARM System-on-Chip (SoC) processor. However, the results from this work can be extended to different NXP SoCs.

1.2 Objectives and methodology

Here are the three objectives we defined for this thesis:

Objective 1: identifying the potential threats in the infrastructure mentioned in Subsection 1.1 and understanding how implementing the secure boot protocol on the cloud connector could help strengthen the security of our system.

> **Methodology:** making a threat model of potential threats in our current infrastructure.

Objective 2: evaluating the secure boot protocol and understanding what kind of Public-Key Infrastructure (PKI) needs to be deployed to support this protocol.

> **Methodology**: conducting a literature review of the different PKIs, understanding the specific tools necessary to implement the protocol on an i.MX 7Dual SoC and implementing it on a development board.

Objective 3: evaluating the robustness of the secure boot protocol against different logical and hardware attacks.

Methodology: researching disclosed vulnerabilities of the protocol. We will focus on fault injections, a hardware attack we will attempt to replicate.

1.3 Outline of the thesis

Chapter 1 gave the context, scope as well as the objectives for our research in securing the booting sequence of embedded systems. Chapter 2 gives the reader the necessary technical and conceptual background to understand the results of this thesis. In Chapter 3, we will investigate the threats to our infrastructure which can be exploited through the cloud connector.

In Chapter 4, we will look at the implementation of the secure boot protocol using the tools and libraries provided by NXP as well as our results. Chapter 5 will focus on the vulnerabilities of the secure boot protocol with a particular focus on fault injections.

Chapter 6 will take a critical look at the secure boot protocol and discuss how it compares to other solutions. Chapter 7 summarises the work presented as well as suggestions for future research on this topic.

Chapter 2
Background

This chapter will present the relevant technical and theoretical background needed to understand this research.

2.1 Basic principles of information security

2.1.1 Information security triad

The Confidentiality, Integrity and Authenticity (CIA) triad is the main component of any security model.

Confidentiality

Confidentiality is "preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information" [PB19].

Integrity

Integrity is "the property that data or information have not been altered or destroyed in an unauthorized manner." [PB19].

Authenticity

Authenticity is "the property that data originated from its purported source" [PB19].

2.1.2 Extending the information security triad

While the CIA triad gives us a proper base to define information security models, non-repudiation, availability and authentication complete those models.

Non-repudiation

Non-repudiation is the "assurance that the sender of information is provided with proof of delivery and the recipient is provided with proof of the sender's identity, so neither can later deny having processed the information" [PB19].

Availability

Availability is "ensuring timely and reliable access to and use of information" [PB19].

Authentication

Authentication is "verifying the identity of a user, process, or device, often as a prerequisite to allowing access to resources in an information system" [PB19].

2.2 Cryptography

2.2.1 Encryption

Encryption is the " process of changing plaintext into ciphertext for the purpose of security or privacy" [PB19]. Encryption protects the confidentiality of data transmitted on a public (or insecure) channel. An encryption scheme is comprised of three algorithms [KL14]:

- **Key generation (Gen)** probabilistic algorithm that generates one or multiple keys used by the other next two algorithms
- **Encryption (Enc):** probabilistic or deterministic algorithm that outputs a ciphertext c given as input a key k and a message m
- **Decryption (Dec):** deterministic algorithm that outputs a message m given as input a key k and a ciphertext m

There are two types of encryption schemes: symmetric and asymmetric encryption.

In **symmetric encryption** (or private-key encryption), the encryption and decryption algorithms use the same key. This encryption scheme requires that both parties have already agreed upon a shared secret before they can securely communicate in a insecure channel. Commonly used symmetric encryption algorithms include the Advanced Encryption Standard (AES) and the Data Encryption Standard (DES).

In **asymmetric encryption** (or public-key encryption), the scheme uses a public-private key pair for the encryption and decryption algorithms. Suppose we have two parties Alice and Bob that have both generated a public-private key pair

and have shared each other's public keys over a public network. If Alice wants to send Bob a message, she uses his public key to encrypt a message and Bob uses his private key to decrypt it. In turn, Bob uses Alice's public key to encrypt data which only Alice can decrypt using her private key. The main advantage of using asymmetric encryption is that it absolves the need to have a pre-shared key prior to communicating on the insecure channel. However, it is slower compared to symmetric encryption. Commonly used asymmetric encryption algorithms include the Rivest–Shamir–Adleman (RSA) encryption algorithm and the Elliptic Curve Cryptography (ECC) algorithm.

The most commonly used type of encryption scheme is somewhat of a hybrid approach in which the key exchange (refer to Subsection 2.2.2) that uses asymmetric cryptography to deduce a common secret that will subsequently be used as the encryption key in a symmetric encryption algorithm. This enables us to take advantage of symmetric encryption without having the need to have a pre-shared key.

When considering a secure encryption scheme, we need to at least satisfy the following two properties [Sma15]:

Correctness: the decryption always works for a validly encrypted message

Semantic security: in simple terms, having access to the ciphertext does not leak any information about the corresponding plaintext

2.2.2 Key exchange

A key exchange protocol is a scheme in which two parties negotiate a common secret key without having a pre-established key. Both parties start by generating a private-public key pair and exchange their public keys. Then, combining the other party's public key and their personal private key, both are able to deduce a common shared secret. The Elliptic Curve Diffie–Hellman (ECDH) key exchange is the most commonly used key exchange protocol. [Cer09]

A secure key exchange protocol is built in way such that an adversary who has access to the public keys of both parties should not be able to deduce the negotiated key.

2.2.3 Cryptographic hash functions

A cryptographic hash function H is a "function that maps a bit string of arbitrary length to a fixed length bit string" [PB19]. The output of such functions are referred to as "message digest" or "hash value".

$$H\colon \{0,1\}^{*} \to \{0,1\}^{\mathrm{n}}, \text{with n} \in \mathbb{N}$$
$$m \mapsto H(m)$$

A cryptographic hash function should satisfy the following two properties [Sma15]:

One-way functions: given any y from the co-domain of H, it should be computationally infeasible[1] to find the value of x from the domain of H such that:

$$H(x) = y$$

Collision resistant functions: it should be computationally infeasible[1] to find two values m and m' such that:

$$H(m) = H(m')$$

Hash functions are deterministic and un-keyed functions and are ideal when it comes to integrity checks. Given a message, a hash algorithm and the hash value, it is relatively straightforward to verify the integrity of the message. Commonly used hash functions are slights variants of the Secure Hashing Algorithm (SHA) 2 or SHA-2 such as SHA-256 and SHA-512 which respectively produce a 256 and 512 bit-length outputs.

2.2.4 Digital signatures

Digital signatures provide origin authentication, data integrity, and signatory non-repudiation.

A digital signature is an "asymmetric key operation where the private key is used to digitally sign data and the public key is used to verify the signature" [PB19]. When signing a message m, the signer S uses their private key to generate a signature. The most important information contained in the signature is the hash output of m that is encrypted using the private key. Any party that wants to verify that m originated from S uses the tuple message, signature and public key. The verifier decrypts the encrypted hash output contained in the signature, computes the hash corresponding to m and verifies that the two outputs match (see Figure 2.1). [KL14]

[1] despite it being computable, it is practically impossible to compute as it requires too many resources

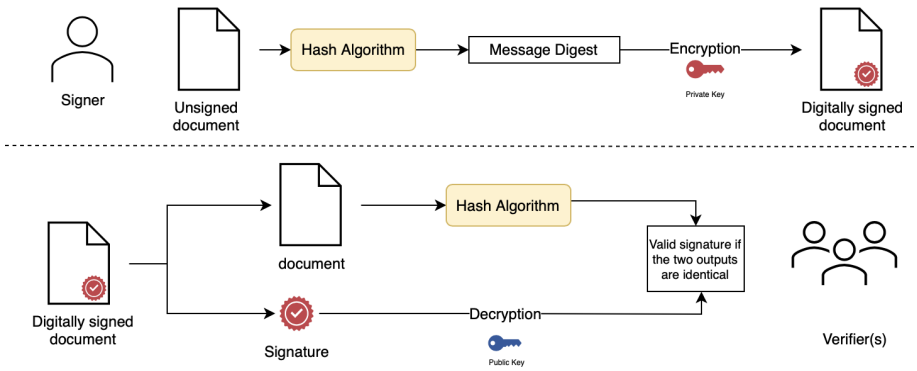


Figure 2.1: Signature creation and verification

A secure digital signature is built in a way that an attacker can not forge a message and a valid signature. If the pair message and signature (m, σ) are generated using a private key, it should be computationally infeasible[1] to forge a pair (m', σ') that can be verified using the corresponding public key.

There are numerous digital signature algorithms to choose from; however, the RSA signing algorithm and Elliptic Curve Digital Signature Algorithm (ECDSA) are the most commonly used ones.

2.2.5 Transport Layer Security

The Transport Layer Authentication (TLS) cryptographic protocol is the successor of the deprecated Secure Socket Layer (SSL) and aims at securing communication over an insecure channel. As the name indicates, its purpose is securing the transport layer in the Open Systems Interconnection (OSI) model. In this section we will have a quick overview of the TLS 1.3 protocol [Res18].

Description of the TLS 1.3 protocol

TLS 1.3 handshake

We will look at the full handshake procedure of TLS 1.3, the most recent and efficient amongst the TLS protocols. Figure 2.2 illustrates this handshake.

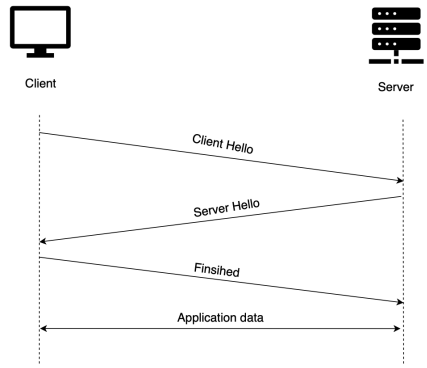


Figure 2.2: The TLS 1.3 handshake

Client Hello – this is the first message sent by the client to the server as a communication request. It is sent along with the following data:

- Supported ciphers: a list of the ciphers the client supports
- Key agreement: the protocol used for the key exchange
- Key share: pre-emptively calculates a shared key based on the different ciphers it supports

Server Hello – after receiving the *client hello*, the server responds with a *server hello* along with the following information:

- Chosen ciphers: the cipher from the client's supported ciphers
- Key share: key share calculated using the chosen cipher to later be used for encryption
- Signed[2] and encrypted certificate: at this point the server has all that is needed to encrypt the certificate
- Encrypted server finish message

Finished – after receiving the *server hello*, the client is now capable of also generating the symmetric key that will be used and decrypt the different encrypted data from the *server hello*. After authenticating the server, it sends the *finished* message marking the start of the secure communication channel.

[2]signed by a trusted Certification Authority (CA), see Section 2.5 for more details

Application Data – Once a secure communication channel is established, the two parties can securely exchange their data.

In the case where the server is not able to use any cipher suite proposed by the client it sends a *hello retry* request in order to renegotiate the key exchange. If unsuccessful, the connection is aborted.

Mutual transport layer security

With the classic TLS protocol, only the server authenticates itself to the client. Mutual authentication is achieved through the mutual Transport Layer Authentication (mTLS). The client sends an additional message to the server containing its signed[2] and encrypted certificate. The server then verifies it before granting access.

2.3 Hardware attacks on embedded devices

2.3.1 Fault injections

Computing systems are designed to withstand faults in their normal operating conditions. Fault injections (or glitching) attacks work by stressing systems and having them operate outside their comfort ranges in order to tamper with the normal flow of execution and give an attacker partial or full control of the device.

Theory behind fault injections

We will use the work of L. Zussa *et al.* [ZDC+12] to illustrate the theory behind fault injections. We will focus on a simple circuit comprised of two registers separated by an abstract block of combinatorial logic network[3] (refer to Figure 2.3). The different blocks use the same clock to synchronise their operations, with a clock period of T_{clk}.

Here is a quick run down of the terminology that will be used to illustrate timing constraints:

- **Maximum propagation delay** (T_{dMax}): the maximum time it takes a signal to travel from a source register to a destination register
- **Setup time** (T_s): the minimal amount of time the input should be stable before a rising edge
- **Hold time** (T_h): the minimal amount of time the input should be stable after a rising edge

[3]Combinational logic is a type of digital circuit comprised solely of primitive logic gates (AND, OR, *etc.*) and in which the outputs are direct consequences of the inputs. These differ from sequential logic whose outputs are direct consequences of both current and past inputs.

- **Skew** (T_{skew}): the slight difference that may exist between the clock signals at the two registers
- **Jitter** (T_{jitter}): the undesired deviation from pure periodicity of our clock signal
- **Internal propagation delay** ($T_{dInternal}$): the time that separates a rising edge and the actual update of the output signal of a register.

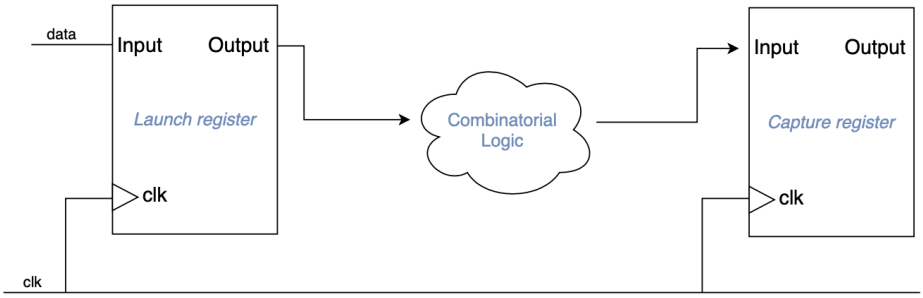


Figure 2.3: Simple diagram to illustrate timing constraints

By data path and clock path we respectively refer to the paths in which the data and the clock signals travel. The data and clock paths can be expressed with equations 2.1 and 2.2 respectively:

$$\text{Data path} = T_{dMax} + T_s + T_{dInternal} + T_{jitter} \tag{2.1}$$

$$\text{Clock path} = T_{clk} + T_{skew} - T_{jitter} \tag{2.2}$$

The launch register outputs data on a certain clock rising edge which is processed by the combinational logic network before it is used as the input to the capture register on the following rising edge. In order to have a properly functioning circuit, the data path should be smaller than the clock path. If this condition is not respected, the data is not processed in time before the following rising edge and creates what is called a setup timing violation. The setup timing constraint can be represented as follows:

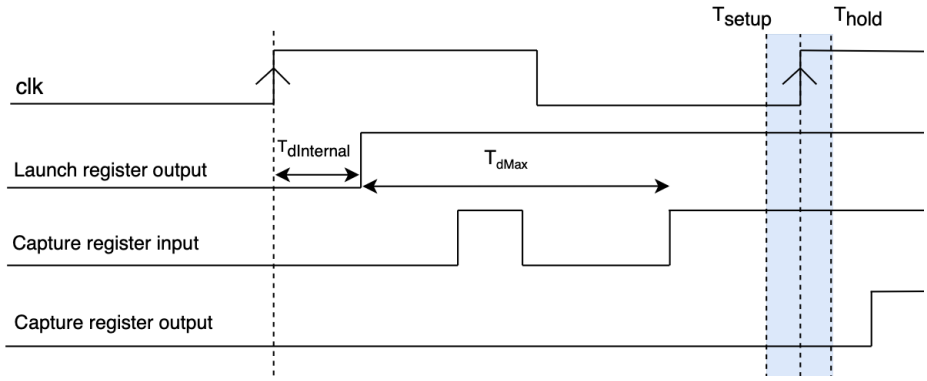
$$\text{Clock path} > \text{Data path} \tag{2.3}$$

$$\Leftrightarrow T_{clk} > T_{dMax} + T_s + T_{dInternal} + 2\,T_{jitter} - T_{skew} \tag{2.4}$$

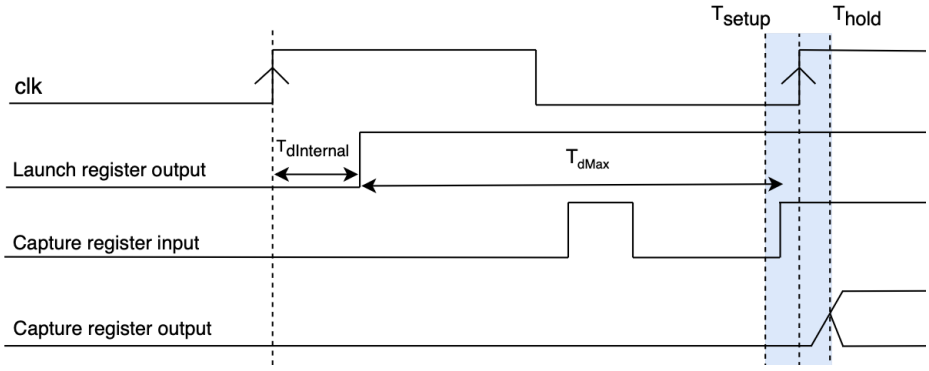
At the same rising edge, the capture register is capturing the current data whereas the launch data is launching the next data. However, we need to make sure that the next data does not arrive during the hold time and change the value of the one currently being processed by the capture register. This creates what is called a hold timing violation. The latter is not relevant to understand fault injections so we will not discuss it any further and any reference to timing constraints refer to the setup timing constraint.

The different violations of setup timing constraints are illustrated in Figure 2.4:

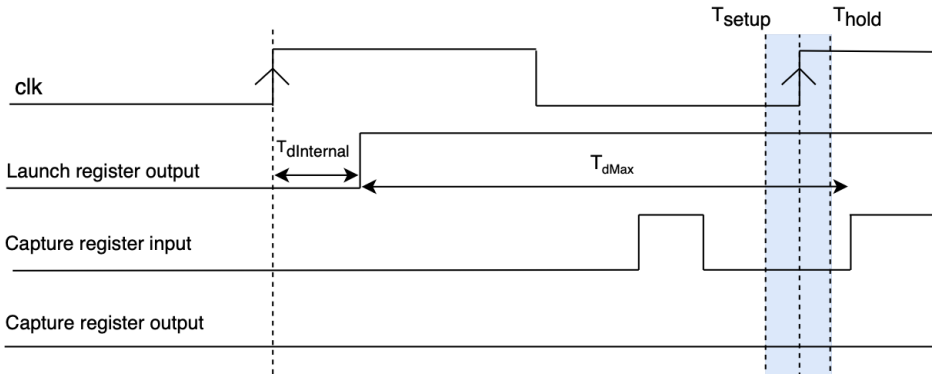
- Respected timing constraints (Subfigure 2.4a): the capture register is able to process the data in time and output the correct value.
- Violated timing constraint leading to a nondeterministic state (Subfigure 2.4b): the capture register processes the data too close to the rising edge resulting in a nondeterministic state (it can either stabilise on a high or low state). A fault may or may not occur.
- Violated timing constraint resulting in fault (Subfigure 2.4c): the capture register processes the wrong value resulting in a fault.



(a) Timing constraints respected



(b) Violation of the timing constraint leading to a nondeterministic state



(c) Violation of the timing constraint resulting in a fault

Figure 2.4: Timing constraints (simplified and adapted from [ZDC+12])

There are two ways of causing setup timing violation [ZDC+12]. Overclocking is the most straight forward one and is achieved by decreasing the clock signal's period. The other option is to increase the propagation time which can be achieved through overheating or underpowering.

The different fault injection methods

There are numerous ways to inject faults onto a system but we will only discuss three of the most common ones [WO22].

Clock fault injections work by inserting clock cycles that are either slightly narrower or wider than regular clock cycles. An attacker generates a signal mimicking the regular clock signal of the device with a slightly different duty cycle around the injection point. The main shortcoming of using the clock glitching method is that it requires the device to be able to use an external clock generator, which unfortunately most devices do not.

Voltage fault injections work by interfering with the voltage supply of a device. When the voltage is increased, the propagation time is reduced, and when the voltage is decreased, the propagation time is increased. Therefore, it is possible to create a setup timing violation by decreasing the voltage.

Electromagnetic fault injections use electromagnetic pulses to cause faults in the system. In simple terms, by inducing variations in the magnetic field around the device, it is possible to cause a voltage difference.

2.3.2 Power analysis

Power analysis is a side-channel attack that seeks to exploit sensitive information leaked by the system when doing sensitive operations [WO22]. It monitors the power consumption of the device and exploits the fact that devices have a different power consumption level based on the operations being performed. There are two forms of power analysis: Simple Power Analysis (SPA) and Differential Power Analysis (DPA). SPA involves evaluating a single power trace of an execution whereas DPA combines multiple executions with varying data.

Power analysis is commonly used to leak cryptographic keys. The power consumed by a device varies based on what instruction is being executed. When performing cryptographic operations, the different instructions executed are strongly linked to the secret key being used, which makes recovering the key possible. Furthermore, power analysis can also be used to extract information that can later be used to precisely time other attacks. In our case, we will use a simple power analysis to properly time the fault injection attack.

2.4 Booting sequence in an embedded Linux system

This section will focus on the booting sequence in an embedded Linux system and individually focus on the four stages that make up this process which, step-by-step, bring our device into an operational state (see Figure 2.5). We will focus on the i.MX 7Dual SoC as it is the chip that is used for this thesis.

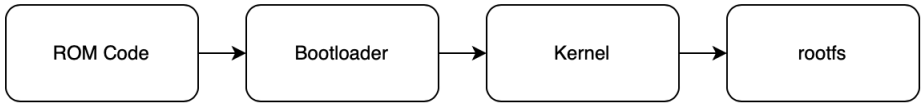


Figure 2.5: Four stages that make up the boot process

2.4.1 Read-only memory code

Read-Only Memory (ROM) code (or boot ROM code) is a piece of code that is stored on a read-only memory of the chip and is the first to be executed when a device is powered on. This code is vendor specific and is loaded onto the device during manufacturing, it can not be modified. When a device is powered on, the Power-On Reset (POR) circuit is executed and resets the system into a known stable state. This ensures that the device boots from the same condition at each reboot. Once stabilised, the circuit sends a POR signal which triggers the execution of the ROM code.

The boot ROM has three primary functions [SSW04; NXP18c]:

Initial hardware configurations: such configurations include initialising memory such as the Static Random Access Memory (SRAM) and the initialising hardware blocks (*e.g.* Cryptographic Acceleration and Assurance Module (CAAM)[4], secure non-volatile storage, Universal Serial Bus (USB), *etc.*); enabling Memory Management Unit (MMU) and caches.

Diagnostics: hardware diagnostics are carried out when the device boots to ensure that everything is functioning properly; enabling exception and interrupt handling, *etc.*

Loading the next image: in its final phase, the boot ROM loads the next image (*i.e.* the bootloader) onto the SRAM and hands over control to it. The image is either stored in a nonvolatile internal memory such as MultiMediaCard (MMC) and flash or it is downloaded through a serial communication such

[4]refer to Section 4.1.1

as USB. This can be done either by copying the whole image or just a data area containing volatile variables onto the SRAM. The loaded image can be compressed in which case it is up to the ROM code to decompress it. Handing over control is done by updating the program counter to point to the start of the next image.

The boot ROM is in charge of loading the code and uses the internal BOOT_MODE registers and electronic Fuses (eFuses) to determine which method will be used to run the subsequent programs on the device.

If the device is configured to run using verified boot or encrypted boot, the High Assurance Boot (HAB) (see Subsection 4.1.1 for more details), a component of the boot ROM is in charge of carrying out the various cryptographic operations.

2.4.2 The bootloader

The primary function of the bootloader is to do the remaining hardware configurations in order to load the kernel [SV21]. If the size of the bootloader image is large and can not fit into the SRAM, the bootloader can be broken down into two stages. Das Universal Boot Loader (or U-Boot) is the most popular bootloader used in embedded systems. Secondary Program Loader (SPL) and U-Boot proper are the names given to the first and second stage bootloaders respectively.

SPL

The primary function of the SPL is to initialise the Dynamic Random Access Memory (DRAM) which has a larger memory compared to SRAM on which it will load U-boot proper. Once initialised, SPL copies U-Boot proper onto SRAM and hands over control to it.

U-Boot proper

After initialising the remaining hardware, U-Boot proper prepares the device to load the kernel. U-Boot proper loads the kernel image into memory as well as a structure containing the different hardware configurations the kernel needs and hands over control to it. U-Boot proper also provides a command-line user interface that can be used to interact with the system. It is also in charge of providing the necessary information for the kernel to mount the root filesystem (rootfs). If an initial Random Access Memory filesystem (initramfs) is used, U-Boot proper loads it onto memory and passes its location and size to the kernel or it simply provides the location of the rootfs using the kernel's command line argument.

2.4.3 Kernel

The kernel is the core component of any operating system. Embedded Linux kernels differ from "traditional" Linux kernels (used in desktops or servers) by being highly customised to their target and use case, resulting in smaller sizes and better performance [GSK+19]. The kernel is the core interface with the device's hardware and is in charge of managing its resources (*e.g.* memory, drivers, processes, *etc*). Once the kernel is done with its initialisation it mounts rootfs.

2.4.4 Rootfs

The rootfs contains all the files, libraries and programs necessary to have an operational system [GSK+19]. After the kernel transfers control of the device, the *init* program is the first program executed. It is executed using root privileges and it launches other daemon programs and brings the device to a working state by doing the remaining configurations.

2.5 Public key infrastructure

A PKI is the backbone of any system that relies on public-key cryptography. We will look at the main components of a PKI as described by J. Buchmann *et al.* [BKW13]

2.5.1 Certificates

In a PKI context, public-key certificates bind the identity of an entity with their public keys. The CA is the trusted third-party entity that binds the public key to the entity which holds the corresponding private key and issues a public key certificate. The relying party (the entity that verifies the validity of the certificate) reduces the trust in a public key to the trust in an entity (*i.e.* the CA). By reiterating this reduction, certificate chains are formed. The relying party should first verify the public keys are from a trusted source before using them.

X.509 certificates

The most commonly used certificate standard is the X.509 standard and is defined by the International Telecommunication Union [ITU19]. There are two types of X.509 certificate:

- public key certificates: specify the information to be used in a PKI setting
- attribute certificate: specify the information used in a privilege management infrastructure to bind an attribute to a subject

We will only focus on public-key certificates.

X.509 certificates are represented as ASN.1[5] of type *SEQUENCE*. The latter simply represents the data as an ordered list. Listings 2.1 and 2.2 provide more details on the structure of a X.509 certificate.

```
Certificate  ::=  SEQUENCE  {
    tbsCertificate       TBSCertificate,
    signatureAlgorithm   AlgorithmIdentifier,
    signatureValue       BIT STRING
}
```

Listing 2.1: ASN.1 specification of an X.509 certificate

```
TBSCertificate  ::=  SEQUENCE  {
    version         [0]  EXPLICIT Version DEFAULT v1,
    serialNumber         CertificateSerialNumber,
    signature            AlgorithmIdentifier,
    issuer               Name,
    validity             Validity,
    subject              Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                         -- If present, version MUST be v2 or v3
                               subjectUniqueID [2]  IMPLICIT
    UniqueIdentifier OPTIONAL,
                         -- If present, version MUST be v2 or v3
    extensions      [3]  EXPLICIT Extensions OPTIONAL
                         -- If present, version MUST be v3
}
```

Listing 2.2: ASN.1 specification of the TBSCertificate object of an X.509 certificate

Other than the information about the public keys, the TBSCertificate (to be signed certificate) contains information about the CA and the owner of the certificate. The extensions portion of a TBSCertificate allow for the addition of extra information to satisfy the different PKI process without altering the basic ASN.1 data type.

The AlgorithmIdentifier field defines the algorithm that will be used to sign the certificate and the signatureValue contains the signature of the TBSCertificate.

[5]The Abstract Syntax Notation One (ASN.1) is used to formally represent data transmitted in telecommunication protocols [ITU08]

2.5.2 Trust and validity Models

In order to have a properly working PKI, the relying party should be able to verify the authenticity and validity of the public keys. We will take a look at the different types of trust and validity models that can be used in a PKI.

Trust models

Trust models define how we can trust the authenticity of public keys. There are three types of trust models: hierarchical trust, direct trust and web of trust.

Direct trust

The direct trust is the most basic trust model out there. In this scenario, the public keys are directly obtained from the key owner or the key owner is the one confirming the authenticity of the keys.

Hierarchical trust

A hierarchical trust model, as the name suggests follows a hierarchical structure which can be easily represented using a tree structure, as illustrated in Figure 2.6.

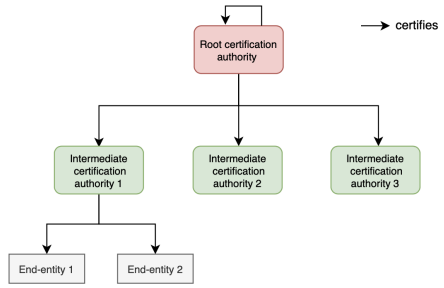


Figure 2.6: Hierarchical trust model

At the root of the hierarchy we have what is called a root certification authority, which is a self-signed CA and is considered the root of trust of the structure. It certifies the public keys of its CA successors which in-turn certify other CAs or end-entities. Generally speaking, the intermediate nodes are CAs and the leaves of the tree are the end-entities.

During the verification process, the authenticity of the public keys are verified using what is called a certification path. The first certificate in this path should be a root certification authority and the entity being verified is the last subject of this path.

For this trust model to work, direct trust should be established between the end-entities and the root CA.

Web of trust model

In a web of trust model, public keys are trusted if they have been either acquired from a trusted source or directly from the owner like in a direct trust model. This trust model aims at forming a decentralised trust model.

Validity models

A validity model in a PKI refers to the period in which a digital signature is valid. The signing and verification are not necessarily done at the same point in time which can result in an invalid certificate during the verification process. We will suppose that the relying party needs to verify the following certification path: $C_0, C_1, .., C_n, n \in \mathbb{N}$.

Shell model

For a certificate to be valid in a shell model, all certificates in the certification path should be valid at verification time (refer to Figure 2.7 for a visual representation). This model is mostly used in cases where signing and verification are performed close to each other.

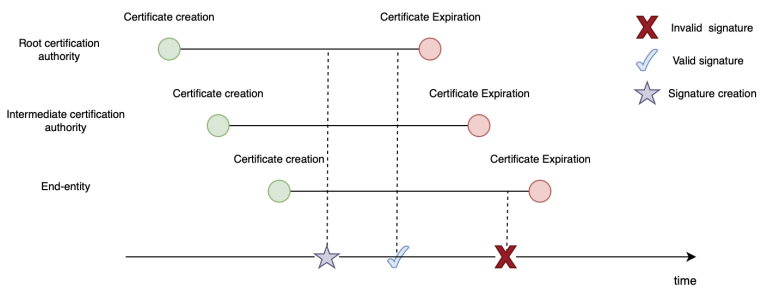


Figure 2.7: Signature validity in the shell model (adapted from [BKW13])

Chain model

For a certificate to be valid in the chain model, its predecessor in the certification path should be valid when signing it. Contrary to the shell model, the chain model only required to have a valid signature during the signing process and not during the verification (refer to Figure 2.8 for a visual representation). This may pose an issue when it comes to long term support when dealing with revoked keys.

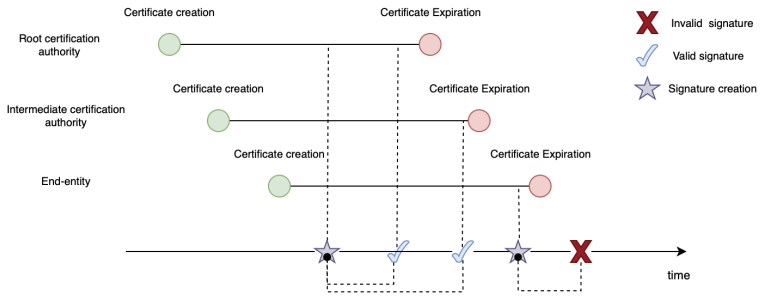


Figure 2.8: Signature validity in the chain model (adapted from [BKW13])

2.5.3 Certificate revocation

Certificate revocation refers to the process of invalidating a certificate before its expiration. Certification revocation can either be due to changes to information found on certificate (*e.g.* change of subject name or identity) or more seriously due to key compromise (leak or suspected leak of private keys) that weaken the whole infrastructure. In order to have a proper revocation scheme, the following requirements should be met:

- Provide information on the revocation: time, reason for revocation
- Readily available for anyone and the most up-to-date information
- Authenticity of the information should be verifiable by anyone

There are two ways of tracking revoked certificates, through the use of Certificate Revocation Lists (CRLs) [HPFS02] or using the Online Certificate Status Protocol (OSCP) [GAM+99].

CRLs

CRLs are lists of revoked certificates signed and published by the CA. They are made available through a public URI. They are specified using the X.509 standard. This list is regularly updated as more and more certificates are revoked by the same CA. When a user wants to verify that the certificate they have is valid, they simply download the CRLs and see if that particular certificate is in the CRL.

Delta CRLs

The main drawback of the CRL is that its length grows overtime. This becomes less efficient as CRL require to be frequently downloaded in order to have the most up-to-date lists. Delta CRL aim to resolve this issue by only publishing certificates added to the Base CRL (which is the complete CRL).

Authority Revocation Lists

Authority Revocation Lists are a particular form of CRLs which are used to list revoked CA certificates. When a CA's certificates are revoked, all the certificates which it has signed prior to and after the revocation date should not be trusted.

OSCP

When using CRLs, the user should periodically check if they have the up-to-date list on hand and download the latest version before verifying that the signature has not been revoked. When using the OSCP, the user simply queries a server with the certificate it wants to verify to obtain its status.

Chapter 3
Threat model of the cloud connector

This chapter will focus on Disruptive Technology's IoT infrastructure to understand what threats can be exploited by an attacker. The STRIDE threat modeling method was used for this analysis.

3.1 The STRIDE threat model

STRIDE is a mnemonic that stands for **S**poofing, **T**ampering, **R**epudiation, **I**nformation disclosure, **D**enial of service, and **E**levation of Privilege. Each threat violates a security property any system should have. Table 3.1 gives a quick overview of each threat.

Threat	Property Violated	Threat Definition
Spoofing	Authenticity	"The deliberate inducement of a user or resource to take incorrect action" [PB19]
Tampering	Integrity	"An intentional but unauthorized act resulting in the modification of a system, components of systems, its intended behavior, or data" [PB19]
Repudiation	Non-repudiation	Denying involvement in data exchange or processing [PB19]
Information Disclosure	Confidentiality	"Data disclosed without authorization" [PB19]
Denial of Service	Availability	"The prevention of authorized access to a system resource or the delaying of system operations and functions" [PB19]

Threat	Property Violated	Threat Definition
Elevation of Privilege	Authentication (or authorization)	"The exploitation of a bug or flaw that allows for a higher privilege level than what would normally be permitted" [PB19]

Table 3.1: Summary of the STRIDE model

3.1.1 The STRIDE model applied to the cloud connector

We will apply the Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege (STRIDE) model to evaluate the security threats that may weaken our infrastructure from exploiting the cloud connector. The same infrastructure presented in Subsection 1.1 will be used for this analysis. This modeling is based on the work of A. Shostack [Sho14].

Spoofing

Spoofing can take two forms in our case: spoofing a machine and spoofing a "file" or a process running on a system.

By spoofing the cloud connector, both the sensors and the cloud would wrongfully assume that the data they are receiving and transmitting were relayed by an authentic cloud connector which may give the attacker the possibility to tamper with the data being transmitted.

Spoofing "files" or processes on the device can dupe the genuine running processes. The latter are led to believe that they are calling or using the appropriate resources when in fact they are not.

Tampering

Another possibility is to tamper with the processes running on a genuine cloud connector. This can either be done to prevent the device from operating normally or to run an arbitrary process on the device as it is doing its intended tasks. A perfect example of the latter would be infecting a device and using it in an IoT botnet[1]. The arbitrary code running can also be a bootloader or a kernel.

Tampering with the network traffic is another way an attacker can cause damage to the infrastructure. By forging or dropping packets, an attacker can manipulate the data flow between the different components of the infrastructure to their liking.

[1] An IoT botnet is a network of IoT devices infected by malware used to launch greater attacks such as Distributed Denial-of-Service (DDoS) attacks.

Repudiation

A compromised gateway can perform malicious tasks such as tampering with the network traffic but then deny its involvement. Without a proper logging system, an attacker can easily operate without any accountability.

Information Disclosure

Leaked information such as cryptographic keys, proprietary code, etc. can have devastating effects. Since the cloud connector is not necessarily placed in a secure location, away from any possible attacker, having physical access to the device can result in firmware dumping and reversing the code on the device. However, even when not having physical access to the device, an attacker can monitor the ongoing traffic and possibly leak sensitive information.

Denial of Service

A Denial of Service attack on our cloud connector simply renders it useless. This can be achieved internally by running resource exhausting programs or externally by flooding the device with useless traffic in order to paralyse it.

Escalation of Privilege

Having complete access to the device is without a doubt the most devastating effect an attack can have on the cloud connector. Elevated privileges allow an attacker to easily install malware on the device, modify the device's configurations and establish a starting point for larger scale attacks.

3.1.2 Addressing these threats

Different security measures can be implemented, or have already been implemented, to prevent these threats. Some of these measures will be briefly discussed.

Authenticated encryption

The infrastructure uses an end-to-end encryption scheme between the sensors and the cloud. During manufacturing, each sensor is assigned an asymmetric encryption key. The sensors and the cloud exchange their public keys through a secure channel using the TLS protocol in order to deduce the encryption keys that will be used to exchange packets. Furthermore, the packets sent by the cloud and the sensors are signed using their respective keys. This means that the cloud connector is not able to decrypt these packets and makes forging packets more difficult.

Securing network traffic

In section 2.2.5, we have seen how TLS and mTLS can be used to establish secure communication in an insecure communication channel. mTLS is used to secure communications between the cloud connector and the cloud. This reduces the chances of an attacker impersonating the cloud connector and prevents unauthorised devices from connecting to the cloud. The server only accepts communications if it is able to verify the identity of the client using a certificate signed by a CA it trusts.

Furthermore, to protect the cloud connector from Denial of Service (DoS) attacks through request connection flooding, an intrusion detection system or firewall can be implemented to block and filter unwanted traffic.

Securing the physical layer

Securing the physical layer of the cloud connector is crucial as not doing so gives an attacker a larger attack surface. The cloud connector is not necessarily stored in a secure location so it is likely that a malicious attacker can get their hands on their device and exploit the hardware or the physical layer of the device. For instance, the different debugging interfaces used during the production phase should be disabled. During the design, ensuring a tamper-resistant and tamper-evident product is important.

Running integrity and authenticity checks

Regularly running integrity and authenticity checks on the device greatly reduces the risk of having unauthorised code running on the device. These verifications can either be done early on in the boot process or continuously as the device is operating. Such measures have not been implemented which exposes the cloud connector to different attacks.

3.2 Focusing on the threat of running unauthorised code

Our main focus is to eliminate the threat of having unauthorised code on the device which can lead to two types of attacks: targeted and untargeted attacks.

3.2.1 Untargeted attacks

By untargeted attacks we refer to attacks that do not solely target a company or its resources. The goal is to exploit a particular vulnerability that can be found across multiple devices. Untargeted attacks are easier to execute because the attacker looks for victims for a particular attack.

An example of such attacks is the Mirai malware [AAB+17], a worm-like malware that infects IoT devices and adds them to its DDoS botnet. Infected devices scan for open ports on the Telnet or Secure Shell (SSH) ports and attempt to login in by brute forcing the authentication. If successful, the malicious payload is delivered, the device is infected, attempts to infect other devices and executes the DDoS command sent by the attacker.

UbootKit is another instance of a worm-like malware that targets different IoT devices using U-boot as a bootloader [GWL+18]. It rewrites the bootloader and allows the execution of arbitrary code which entirely compromises the device. Its persistence through multiple reboots make this malware even more dangerous.

3.2.2 Targeted attacks

In this scenario, the attacker's objective is to harm a specific company and their infrastructure by targeting their systems. These attacks tend to have a devastating impact but are harder to implement. Contrary to untargeted attacks, the attacker is not looking for potential victims for a particular attack but rather potential exploits for a particular victim. We will be working with the assumption that an attacker can gain physical access to the device.

The STRIDE model in Subsection 3.1 revealed that an attacker can modify the intended behaviour of the cloud connector as no measures prevent unauthorised code or processes from running on the device. A possible scenario would be crafting an exploit that can leak the private keys used by the cloud connector when connecting to the cloud and using these credentials as an entry point to exploit potential vulnerabilities on the cloud.

3.2.3 The secure boot protocol

The secure boot protocol ensures that only verified code is running on the device by establishing what is referred to as a chain of trust linking the main components of the boot sequence. Ensuring the authenticity and integrity of the running code from the moment the device is reset until it is fully functional gives us the confidence to claim that no unauthorised code is running on the device.

The root of trust of this chain is the ROM Code. As mentioned in Section 2.4, this code is a vendor-specific read-only code. It uses the public keys permanently programmed into the eFuses during manufacturing to verify the authenticity of the next component (*i.e.* the bootloader) before handing control over to it. Each component does the same with its successor. In the case where the signature verification fails, the execution should be interrupted because the chain of trust is broken.

The chain of trust should at least be extended until the rootfs. In Subsection 2.4.4, we mentioned that the *init* program is executed using root privileges which makes ensuring its authenticity crucial. The rootfs should be configured as a read-only partition[2].

[2]Having a read-only partition without extending the chain of trust until the rootfs is not enough as it does not prevent an attacker from overwriting the whole partition

Chapter 4
The secure boot protocol on the cloud connector

As mentioned in Subsection 1.1, the cloud connector is running an i.MX SoC from NXP. In this chapter, we will focus on the implementation of the secure boot protocol on this particular SoC as well as the PKI it supports.

4.1 Secure boot on NXP processors

4.1.1 High assurance boot

The HAB library can be found on certain NXP processors and allows the authentication and encryption of boot images [NXP18b]. This library rests in the on-chip ROM code. Furthermore, the Read-Only Memory Vector Table (RVT) contains the HAB Application Programming Interface (API) addresses through which the bootloader verifies the kernel image. Unfortunately, the HAB library can not be used to verify the integrity and authenticity of the rootfs. To extend the chain of trust until the rootfs, the kernel should be in charge of running the authenticity and integrity check. The kernel having been itself verified by the HAB library, we can trust it to run the image verification. The module in the kernel in charge of running the verification is called *dm-verity* [ZAH+14].

The HAB library uses RSA signatures[1] for image authentication and uses the X.509 public key certificate format. SHA-256 is the cryptographic hash function used during the signature verification.

The CAAM is a hardware component that allows cryptographic acceleration of hashing, encryption and decryption algorithms, secure random number generation, long-term protection of secret data, *etc.* If the SoC contains a CAAM hardware block, it uses it to accelerate the SHA-256 digest operations.

[1] supported key sizes: 1024, 2048, 3072, and 4096 bits

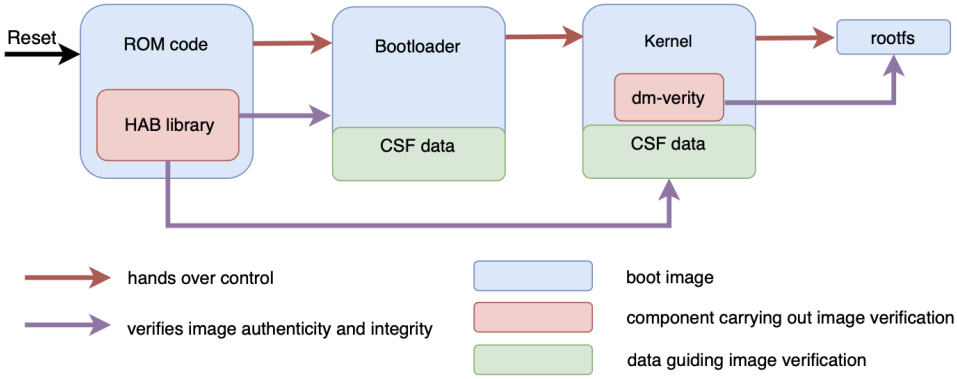


Figure 4.1: The secure boot protocol using the High Assurance Boot

4.1.2 Command sequence files

The Command Sequence File (CSF) data is processed by the HAB during the image authentication. It contains the commands for the HAB as well as the digital signatures and public key certificate. The CSF is a text file that is parsed using the Code Signing Tool (CST) (see Subsection 4.1.3) to generate the CSF binary data that can be interpreted by the HAB. This data is then appended to the image being signed. The most important commands that can be found in the CSF for image authentication are:

Header: contains header data as well as default values that will be used in the subsequent commands.

Install Super Root Key (SRK): command to install and authenticate the SRK that will be used when the HAB authenticates the CSF and the image.

Install Command Sequence File Key (CSFK): installs the public key that will be used to authenticate the CSF data of an image by the HAB. This key is authenticated using the previously installed SRK.

Authenticate CSF: command to authenticate the CSF using the previously installed CSFK.

Install Key: command to authenticate and install the public key used to authenticate the actual image being signed. This key is verified by the previously installed SRK.

Authenticate data: verifies the authenticity of the image loaded into memory. For each block that is being verified, the name of the binary file, the starting

load address in memory, offset and length should be specified. The key that will be used for authentication should also be specified.

Unlock: prevents the specified hardware features from being locked when exiting the boot ROM (*e.g.* the CAAM or the possibility blow fuses in order to revoke SRKs[2]).

An example of a CSF file can be found in Listing A.1 in Appendix A.

4.1.3 The code signing tool

The CST is a tool developed by NXP and contains all the necessary programs to implement the secure boot protocol on the SoC [NXP18a].

Image signing

Given a CSF description file and the corresponding boot image the *cst* tool generates the binary CSF data that will later be appended to the image being signed. The commands to use can be found in Listing A.2 in Appendix A

PKI tree generation

The HAB PKI structure is generated using the *hab4_pki_tree.sh* script. Here is a visual representation of that structure:

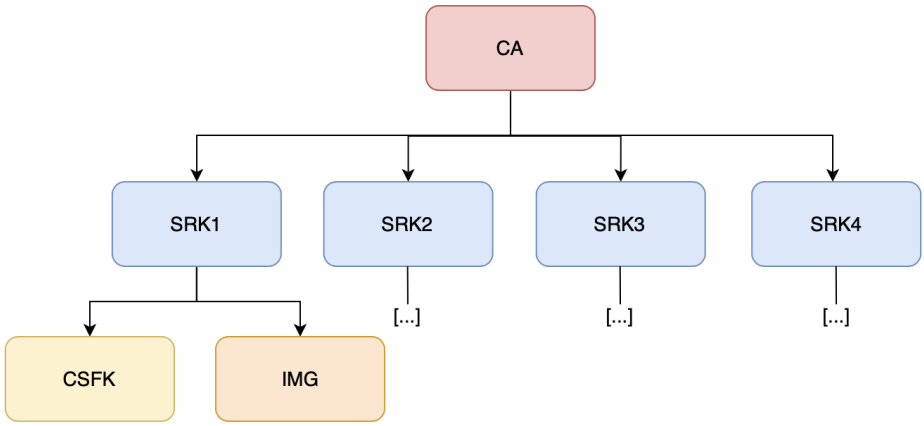


Figure 4.2: HAB PKI tree

[2]refer to Subsection 4.2.1

The CST generates all the necessary keys and their corresponding certificates:

- **CA key:** Key that will be used to sign the SRK certificates
- **SRK:** Key used to sign the CSF and the Image (IMG) certificates. Only one SRK can be used per reset cycle.
- **CSFK:** key used to verify the authenticity of the CSF
- **IMG key:** key used to verify the authenticity of the boot image

The HAB can also be configured to use fast authentication where the same key is used to verify both the CSF and image file. This reduces the amount of keys needed as well as certification verifications resulting in a faster boot. In such cases, only the SRK keys are generated.

The key generation process can be replaced by an alternative, secure generator. However, the following three conditions must be met in order to be able to use the CST to sign the images:

- Keys should be in the Public Key Cryptography Standards (PKCS)#8 format
- Certificates should be in the X.509 format
- Keys and their corresponding certificates should follow certain naming and placement conventions: a key <key_name>_key.<extension> stored in the *cst/keys/* directory should have the corresponding certificate: <key_name>_crt.<extension> stored the *cst/crts/*

SRK table and eFuse data

The CST also contains the *srktool* script that generates the SRK public table and its corresponding eFuse data. During the signing process, the CST uses the SRK table to determine which key will be used to sign the image[3]. The eFuse data contains the hash values of the SRKs in the same order as they were specified in the SRK table. The SHA-256 hash values of the SRK public keys are permanently burnt onto the SoC and will be used by the HAB when verifying the images during boot. The one-way and collision resistant properties of hash functions mentioned in Subsection 2.2.3 guarantee that using the hash of the public key stored on the eFuses to verify the public keys found in the CSF is enough to verify the authenticity of those public keys.

[3]the naming conventions for the keys and their certificates allow the CST to easily find the private keys corresponding to the certificates mentioned in the CSF to sign the image

4.2 Results

4.2.1 PKI for the secure boot protocol

Trust model

As we mentioned in Subsection 4.1.1, during boot, the HAB does the signature verification by comparing the hash of the public keys with the actual public keys found in the CSF binary data of the image. It then uses the public keys found in the certificate to verify the digital signature of the image. Therefore, the verification scheme during boot follows the direct trust model. This is partly related to the fact that during boot, the device is unable to cover the whole certificate path and can only trust the public keys stored on its eFuses.

Validity model

Verifying the validity of the certificate during boot is unfeasible. At the early stages of the boot sequence, the ROM code does not have the possibility to verify any certificate in the certification path. The HAB library does not provide the possibility to verify the validity of a certificate [Yur18]. NXP recommends adding the validity check further along the boot process (by the bootloader or the kernel).

Certificate revocation

Our gateway, and embedded systems in general, have limited computing capabilities compared to larger computing systems. Having to regularly download complete or delta CRLs would be too resource intensive and OSCP should be the favored scheme as the device is only interested in verifying the status of at most four certificates for the secure boot protocol.

The SoC we are working on can only store four SRKs. Revoking a key is done by burning its corresponding SRK_REVOKE fuse. In order to allow the bootloader or the kernel to blow an SRK_REVOKE fuse, the HAB should be instructed by an Unlock command in the CSF to allow burning these fuses. Here is the scheme we came up with for a secure revocation of a SRK:

> **Scenario 1:** the certificate being revoked is the one being used to verify the current boot process
>
> The different boot images should first be updated and signed with a new key. During the first update phase, these image should contain the unlock command that allows to burn SRK_REVOKE fuses. After the bootloader or kernel revokes the key by burning its corresponding SRK_REVOKE, the boot images

should once more be updated and this time without the unlock command that allows key revocation.

Scenario 2: the certificate being revoked is not the one being used to verify the current boot process. The steps are similar to the ones in Scenario 1, except that we do not sign the updated images with a new key.

It is possible to allow any image to blow SRK_Revoke fuses by leaving the appropriate unlock command on every image's CSF. This removes the need to update the images when the certificate being revoked is the not the one in use for the current boot sequence. However, this opens up the device to possible attacks (*e.g.* burning all the SRK_Revoke eFuses and "bricking" the device).

Furthermore, it is recommended to never revoke one key to prevent having a "bricked" device.

4.2.2 Implementation results

Using the tools provided by NXP, we attempted to implement the secure boot protocol on an i.MX 7Dual SoC development board. The goal of this implementation was to have a working prototype to test the fault injection attack on. However, the implementation was not entirely successful. The implementation followed the instructions provided by the instruction manual provided by NXP [iMX21; NXP20] using a U-Boot bootloader.

After generating a PKI, burning the corresponding SRKs with the appropriate public keys, signing the image and flashing it onto the device we received HAB events indicating that the signature verification failed (see Section A.3 in Appendix A for the detailed log output). Debugging the issue using these HAB events was unsuccessful. Debugging efforts included:

- Manually comparing the binary segments of the image that failed the verification from our workstation and from the development board's memory
- Testing different bootloader images built using Disruptive Technologies' personalised bootloader images as well as different U-Boot git repositories found online
- Implementing the protocol on a i.MX 7Dual Sabre board, a different development board

- Flashing the bootloader images using different methods: Serial Download Protocol (SDP) such as *imx_usb*[4] and *uuu*[5] or by directly exporting block devices over a USB connection

However, we did manage to interrupt the execution whenever the flashed image did not have CSF binary data appended to it, as shown in Figure 4.3. These results were enough to be exploited with a fault injection attack.

```
U-Boot SPL 2020.01-gec6d8ef2c9-dirty (Mar 07 2022 - 11:23:33 +0100)
Trying to boot from USB SDP
SDP: initialize...
SDP: handle requests...
Downloading file of size 491552 to 0x877fffc0... done
Jumping to header at 0x877fffc0
Header Tag is not an IMX image
hab fuse not enabled

Authenticate image from DDR location 0x877fffc0...
Error: CSF header command not found
spl: ERROR:  image authentication fail
### ERROR ### Please RESET the board ###
```

Figure 4.3: Boot interrupted when no CSF data was found

[4]https://github.com/boundarydevices/imx_usb_loader
[5]https://github.com/NXPmicro/mfgtools/releases/tag/uuu_1.4.193

Chapter 5
Fault injection attack to bypass secure boot

By verifying the authenticity and integrity that only authorised code is loaded during boot, the secure boot gives us the confidence to claim that only authorised code is running on the device. However, fault injections attacks are known for being able to bypass the secure boot protocol by skipping the instructions in charge of the image verification. In this chapter, we will present our work in replicating a fault injection attack using a "homebrewed" approach to understand the complexity of this attack.

5.1 Making a voltage glitcher

There are a number of commercially available voltage glitchers such as ChipWhisperer [OC14]. However, we decided to build our own voltage glitcher. We decided to implement the glitcher on a Field-Programmable Gate Array (FPGA) for its flexibility and its precision. Ideally, the duration of the glitch should be measured in the range of nanoseconds or microseconds and it is important to be able to time when to glitch as precisely as possible. A Zedboard was used for this project.

5.1.1 Description of the implementation

What is expected of the glitcher is being able to output a signal that indicates when the device should be in a high state and when in it in a low state. A high state indicates that the device is powered on and a low state indicates that is is not. Figure 5.1 illustrates the system that was designed to control and time the glitch and the source code can be found in Appendix B.

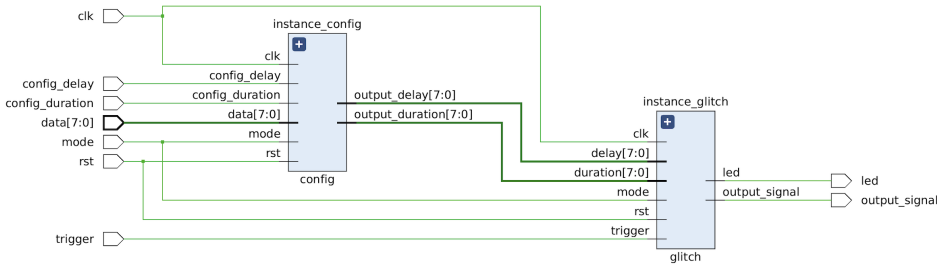


Figure 5.1: Block diagram for the glitcher

There are a total of five inputs to the system and two output. Their functions are stated in Table 5.1.

Name	Type	Description
data	Input byte array of length 8	Binary input used to configure the delay and the duration of the glitch. These values indicate the number of rising edges for both these values
mode	Input bit	If mode is 1, then we are in the configure mode If mode is 1, we are in the glitch mode
trigger	Input bit	When trigger is set to 1, causes a glitch after waiting for the configured delay
config_duration	Input bit	When config_duration is set to 1, the value read from data is used to update the value of the duration of the glitch
config_delay	Input bit	When config_delay is set to 1, the value read from data is used to update the value of the delay of the glitch
reset	Input bit	**Configure mode:** When reset is set to 1, it replaces the values of the duration and the delay to their default values.
		Glitch mode: when reset is set to 1, it goes back to an idle state and waits for the trigger. It outputs a 0 indicating a low state.
output	Output bit	If output is 1, then it indicates a high state If output is 0, then it indicates a low state

Name	Type	Description
led	Output bit	Used as a visual indicator of whether we are in a high state or a low state

Table 5.1: Summary of the inputs and output of the designed voltage glitcher

In order to have a flexible glitcher, the user can configure the duration of the glitch and when the glitch should occur directly on the board, without having to reprogram the bit image. We made use of four types of input and outputs:

- **Push buttons:** these are used to read the five bit inputs of our system. When using mechanical buttons, unpredictable bounces occur when toggled. In our case, this will not have an impact on most of the buttons except for the trigger. Instead of implementing a debouncing circuit for the trigger, an alternative would be to define a holdoff period after the first change is detected in which any subsequent changes will not be taken into account. This prevents having multiple glitches back to back. For the other inputs, the unpredictable bounces do not have an effect on their functions.
- **Slide switches:** these switches are used to read the value data. The 8 slide switches are used to read an 8 bit binary value which allows the user to define the values for the delay and the duration.
- **LED light:** indicates whether or not the glitcher is in a high state.
- **Pmods:** Using a pin on a Pmod of the FPGA, we are able to output the signal that will be used to power the device.

Figure 5.2 illustrates the different mappings on our FPGA.

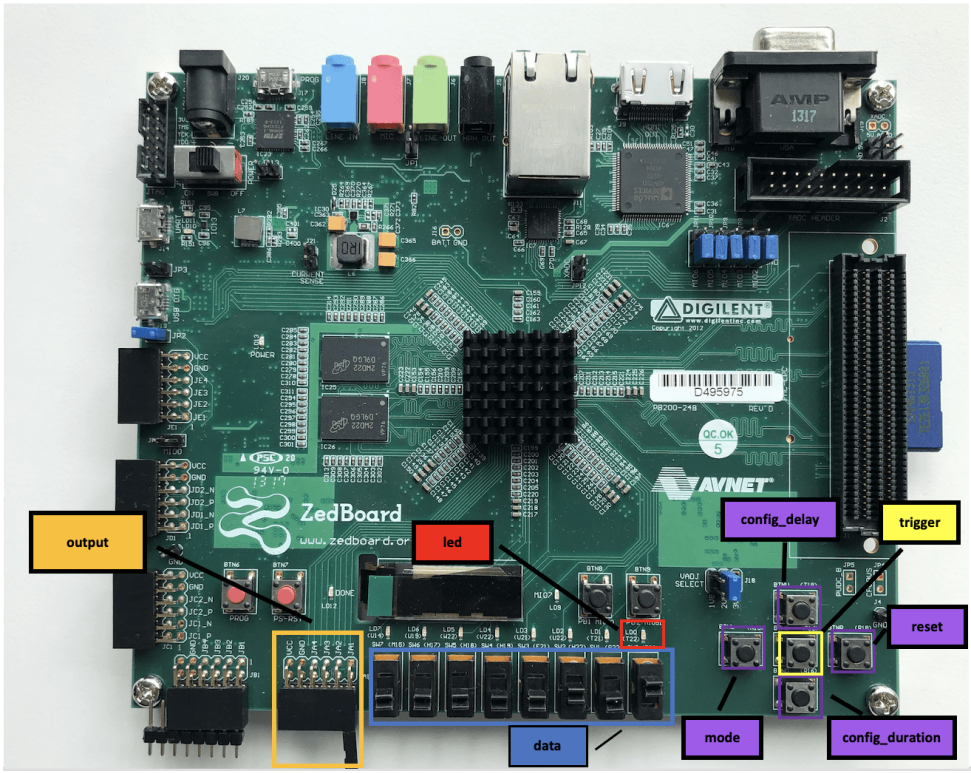


Figure 5.2: The different input and output mappings on the FPGA

5.1.2 Testing the implementation

Simulation

The first step in testing the implementation was by using logic simulator by testing each block separately. This was followed by testing the system as a whole. The test script for the whole system can be found in Listing B.4 in Appendix B configures the duration and the delay to 9 and 4 rising edges respectively and triggers the glitch. The results of the simulation can be found in Figure 5.3. We can see that once the trigger is set to 1, output_signal is activated for 9 rising edges and glitches for 4 rising edges.

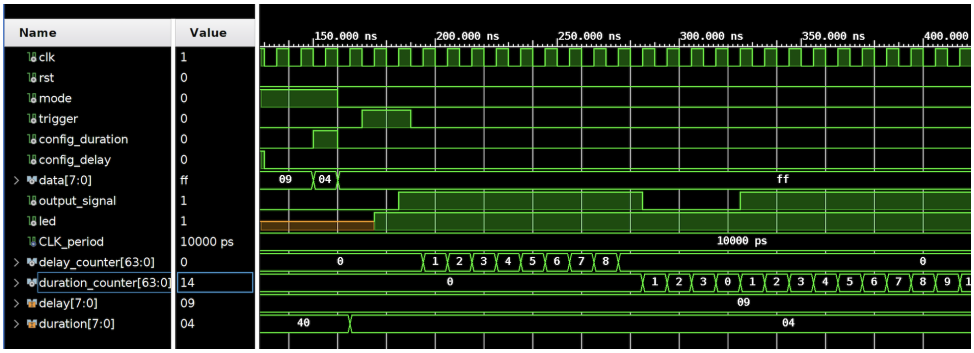
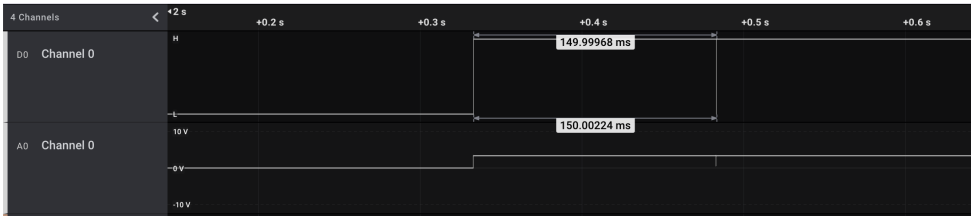


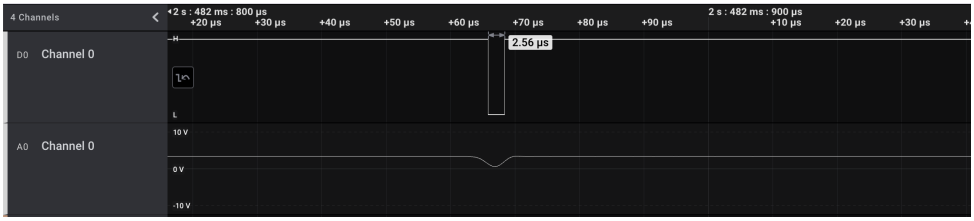
Figure 5.3: Simulation results

Using a logic analyser

Once the simulation indicated everything worked as expected, the next step is to program the FPGA and test that we have the expected results on the board as well. The output signal was analysed using the Salea logic analyser[1] and the results can be found in Figure 5.4.



(a) Focus on the delay before the glitch



(b) Focus on the duration of the glitch

Figure 5.4: Output signal from the logic analyser

[1] https://www.saleae.com

5.1.3 Logic level converter to power the target device

The output of the FPGA can not be used to directly power the target device for two reasons:

- the FPGA outputs 3V whereas the target requires a 5V input
- the FPGA can not provide the necessary current to power the target

We used a Complementary Metal–Oxide–Semiconductor (CMOS) inverter to amplify the output of the FPGA in order to power the target (see Figure 5.5). Because the circuit uses an inverted logic compared to the FPGA output, some adaptations needed to be made in Listing B.3 in Appendix B.

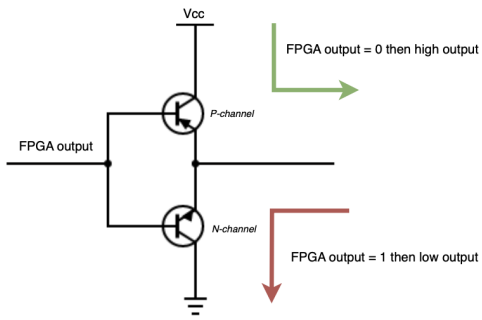


Figure 5.5: CMOS inverter circuit used to amplify the FPGA output

5.2 Attacking the secure boot protocol with a fault injection

We will be looking at two different approaches when trying to attack the secure boot protocol with a voltage fault injection. The first one uses a development board and the second one looks at a real world scenario on a commercially available device.

5.2.1 Using a development board

Development boards provide different hardware interfaces that are easy to access and allow for easy debugging, programming and monitoring. For the fault injection attack on the secure boot protocol, using this approach allows us to easily flash into memory the different boot images using serial communication, monitor the boot process using a console output and measure the power consumption for the power analysis. Two approaches were defined when injecting faults:

Approach 1: attempt to skip a single instruction
The first attempt was to simply skip an instruction when booting, without worrying too much about the precise timing. To do so, a simple infinite loop was added in the bootloader image and the goal was to break out of that loop. Listing 5.1 illustrates the modifications made to the U-boot source code. The comparison that will be executed at the *while* instruction provides a possible injection point to break out of the loop.

```
1 while (1) {
2 }
```

Listing 5.1: Infinite loop added to the U-boot image

Approach 2: attempt to skip a the verification instruction
The second attempt was to precisely time the injection using a power analysis and inject the fault at that precise time. Since the implementation was not entirely successful, the goal was not to skip the verification of the image authenticity but the verification of the presence of a CSF binary (see Listing 5.2).

```
1 /* Verify CSF*/
2 if (!csf_is_valid(ivt, start, bytes)) {
3     goto hab_authentication_exit
4 }
```

Listing 5.2: Instructions verifying the presence of CSF data

To properly time the injection point, we chose to measure multiple power consumption of valid and invalid images to see when the execution flow diverges. The power traces of one valid and one invalid image are show in Figure 5.6.

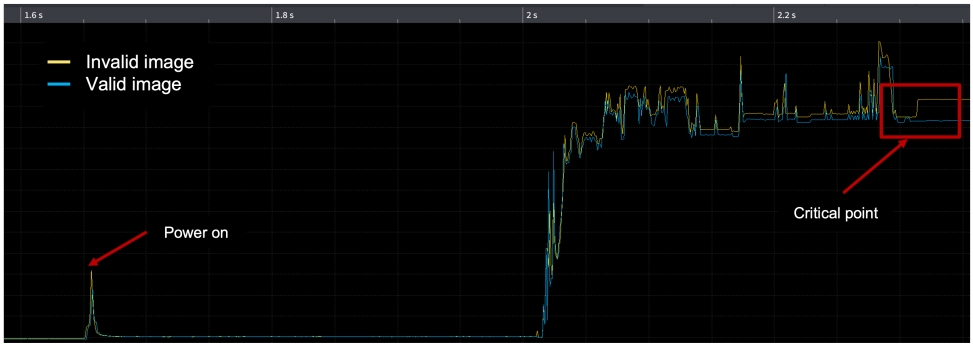


Figure 5.6: Power traces of a valid and invalid image

5.2.2 Real world scenario

Injecting faults on a commercially available embedded device is a lot more complex compared to using a development board. Finding the hardware interfaces for debugging or flashing images is not (or at least should not) be as straightforward. A successful attack requires opening up the device and identifying interesting ports to exploit. Although putting in place different precautions to protect these debugging features does not entirely prevent them from being exploited, it adds a significant amount of complexity to deter attackers.

5.3 Results

Most components on embedded devices require a steady voltage input in order to function properly. However, during their normal operations, they are confronted to a wide range of factors that could cause voltage spikes or dips that may cause malfunctions or damage. Decoupling capacitors are used to suppress such noise by "absorbing" power spikes and by "powering" the system during power dips [COS13].

Because of these voltage regulators, the short injected glitches were "absorbed" and had no visible effect. The only visible effects were recorded when inserting longer glitches that had the effect of resetting the board. Figure 5.7 shows the board being reset multiple times after inserting multiple glitches.

```
U-Boot SPL 2020.01-gec6d8ef2c9-dirty (May 05 2022 - 11:32:
U-Boot SPL 2020.01-gec6d8ef2c9-dirty (May 05 2022 - 11
U-Boot SPL 2020.01-gec6d8ef2c9-dirty (May 05 2022 -
U-Boot SPL 2020.01-gec6d8ef2c9-dirty (May 05 2022
U-Boot SPL 2020.01-gec6d8ef2c9-dirty (May 05 2022 - 11:32:21 +0200)
Trying to boot from MMC1
hab fuse not enabled

Authenticate image from DDR location 0x877fffc0...
```

Figure 5.7: Board being reset after multiple long glitches

Physically removing these capacitors from the board resolves the issue but we chose not to do so as it could damage the development board. However, in case of a targeted attacker, they would not be discouraged from carrying out such modifications to the board.

Despite our unsuccessful replication of the fault injection attack to bypass the secure boot, it should still be considered as a way to bypass the security protocol. C. O'Flynn's work provides proof that voltage fault injection can be used to alter the normal flow of execution of a code [OFl16] and the works N. Timmers *et al.* proved that it could be used to bypass the secure boot protocol [TSW16].

Chapter 6
Discussion

In this chapter, we will discuss the limits of the secure boot protocol and what can be done to enhance the security of our device.

6.1 Limits of the secure boot protocol

6.1.1 Lack of long term support

During this study, one of the main issues we found deals with the lack of long term support for the secure boot protocol.

As mentioned is Subsection 3.2.3, the secure boot protocol uses the public keys programmed onto its eFuses when verifying the images in the boot chain. In Subsection 4.2.1, we saw that due to the limited capabilities of the device during boot, verifying the authenticity of the certificate before handing control over to its successor was unfeasible. What we are essentially doing is trusting that image was signed using the appropriate keys but have no way of guaranteeing that it was signed by the right authority.

During the "early" stages of the device's life cycle, we can assume that these keys embedded onto the fuses can be trusted without having the need to validate the authenticity of the certificates because they were programmed by the manufacturer. However, overtime, with the increasing likelyhood of certificate revocation it is difficult to make the same assumptions.

Consequently, the robustness of our chain of trust is weakened. For it to work properly, handing over control of the boot sequence should only be done if we are certain that the image is authentic. In practice we are not able to make such claims. In addition, verifying the authenticity of the certificates used during the image verification later on in the boot stage goes against the principle of the chain of trust.

We mentioned in Subsection 4.2.1 that out of the four SRKs one should never be

revoked even if the corresponding key is no longer valid in our PKI. We are forced to make this compromise because revoking all four keys will render our device unusable. Continuing to verify the boot images using invalid keys essentially gives us the same trust in the authenticity of the image as if we had not implemented the secure boot protocol.

6.1.2 Vulnerabilities in the boot ROM

Vulnerabilities on certain NXP processors were discovered that allowed bypassing the secure boot protocol. They were identified as CVE-2017-7932 (or ERR010873) [NXP17b] and CVE-2017-7936 (or ERR01872) [NXP17a]. Since these vulnerabilities are in the boot ROM code, they can not be patched through software updates.

The first vulnerability (CVE-2017-7932) is due to an improper parsing of the X.509 certificates by the HAB which results in a stack-based buffer overflow. An attacker can craft a malicious certificate to bypass the signature verification. The i.MX 7Dual chip we are working on has this vulnerability. Recommendations to protect devices against this vulnerability include having a proper Over-The-Air (OTA) update policy and restricting physical access to the device.

The second vulnerability (CVE-2017-7936) is also a stack-based buffer overflow exploit that allows an attacker to write and execute code from an unprotected section of memory using SDP. Recommendations to protect devices against this vulnerability include disabling the SDP port if the chip supports it and restricting physical access to the device.

6.1.3 Frequent image updates with certificate revocation

In Subsection 4.2.1 we proposed a solution that could allow certification revocation to be done in a secure way. However, it required updating the images of our boot chain multiple times.

The complexity associated with updating the earlier stages of the boot sequence is really high and also poses a risk of rendering the device unusable [SV21]. Furthermore, since the HAB can only use one SRK per reset cycle, we are obliged to update every component of the boot cycle during revocation.

6.2 Measured boot, an alternative protocol

Measured boot is an other protocol that aims at ensuring the authenticity and integrity of the code running on a device. This protocol relies on the use of a Trusted Platform Module (TPM) to properly function.

6.2.1 Overview of Trusted Platform Modules

A TPM is a crypto-processor (or chip) designed to enhance the security of a system by providing hardware level protection that runs separately from its host system. We will briefly cover the main security specifications and features of such chips as specified in the TPM 2.0 Library specifications [Tru19] provided by the Trusted Computing Group (TCG).

Root of trust

The specification requires that TPMs provide three forms of trust:

- **Measurement:** when a device boots, the Core Root of Trust for Measurement (CRTM) which is part of the boot ROM is executed. This component does the initial self measurements which it shares with the TPM. These measurements first recorded on the TPM are representative of the initial state of the module and constitutes the initial trust.

- **Reporting:** the TPM can share the part of its contents stored in its memory with a digital signature signed with the endorsement key. The private portion of the key that is used to sign the data never leaves the module and is stored in the non-volatile memory.

- **Storage:** the TPM provides a secure way to store sensitive data and prevent any inappropriate access to memory. The module generates a Storage Root Key (SRK) and stores it in the non-volatile memory. When storing sensitive information such as other keys or certificates, the TPM encrypts them using the SRK and stores them outside the module. Upon request, it decrypts them. Sealing and unsealing respectively refer to the encryption and decryption using the SRK.

Hardware level protection

A TPM should be resistant to all forms of logical and hardware attacks. Tampering with the storage and the reporting of the stored data should not be possible.

Cryptographic features

Previous version of TPMs did not allow the main processor to use their cryptographic primitives. However this is not the case with version 2.0. Operations that can be done on this module include random number generation, hashing, symmetric and asymmetric encryption and digital signatures.

Platform Configuration Registers

Platform Configuration Registers (PCR) are part of a TPM's volatile memory. The values of the register can not be changed directly. The only way of changing a value of a register is by extending it as follows:

$$PCR[x] = Hash\_Algorithm(PCR[x], new\_value)$$

Upon request, the TPM can either simply return the values found in the PCR or can provide a quote. A quote is an attestation that contains a nonce[1],the digest of the concatenation of the values found in the registers which is signed by using the endorsement key.

Sealed storage

With sealed storage, it is possible to wrap keys but what is interesting is that the PCR should be configured in a particular order to unseal them. This provides an extra level of security as decrypting can only be done if the TPM is in a clearly defined state.

6.2.2 Description of the protocol

Similarly to secure boot, each component of the boot sequence computes the hash of its successor. However, instead of verifying the signature it extends the value of one of the PCR.

At the end of the boot sequence, once the kernel is loaded and operational, the values in the PCR indicate the current state of the device and a "summary" of what was previously loaded. By comparing these values to an expected value, a judgement is made to decide whether or not the system can be trusted.

However, the TPM can not on its own interrupt the boot sequence if the system is corrupted but there are different ways of evaluating the boot process.

[1] A nonce is "time-varying value that has at most a negligible chance of repeating, for example, a random value that is generated anew for each use, a timestamp, a sequence number, or some combination of these" [PB19]. In our case, the use of nonce would prevent the reuse of quotes.

Reporting to the device

Reporting the values to the device and letting it decide on how to proceed. This obviously is the worst solution as nothing is preventing the system from continuing to boot even if the recorded values on the PCR are wrong.

Sealing the rootfs encryption key

Encrypting the rootfs, sealing the encryption key on the TPM and linking it to the values on the PCR. Doing so ensures that a corrupted system can not complete its boot process as it will not be able to access its rootfs[2].

When using this approach, one important aspect to consider is how to properly update the values the PCR should expect in accordance to the updated images' signatures.

Remote attestation

To further enhance security, remote attestation can be implemented. The TPM can return a quote with the PCR values which will be forwarded to a remote server. The server then decides whether or not the device can be trusted and if it wishes to give it access to the rest of its resources.

While this may not be an adequate solution for standalone devices that do not need external resources, this allows cutting off the device from the cloud if it is compromised. Furthermore, combining sealed storage and remote attestation can further enhance the security of the device as it would block the user from accessing the cloud but also prevent them from exploiting it.

6.2.3 An alternative to the secure boot protocol?

The measured boot protocol is an interesting protocol to study because it may solve some of the issues the secure boot protocol raises all while adding new features.

Providing an alternative to local attestation

Through remote attestation, it is possible to have a third party decide whether or not the system can be trusted. While this solution is not adequate for all use-cases, in our case where the device in question is a gateway, remote attestation is an interesting feature.

[2]In this case, the kernel should extend the PCR with the digest of the encrypted rootfs

Issues related to the PKI

The secure boot protocol establishes the chain of trust by verifying each component of the boot chain before handing control over to the device whereas with measured boot it is established at the end of the boot sequence using the values recorded in the PCR.

The digital signatures and certificate verification carried out when using measured boot are not done early on in the boot stage. This enables proper certificate validity checks.

Provide runtime security

The aim of the secure boot protocol is to ensure the authenticity and integrity of the initial code loaded onto the device. Any malware that infects the device during runtime can not be prevented by the secure boot protocol. By using a TPM and through remote attestation, running integrity verification on running processes is possible.

6.3 Secure over-the-air updates

Regardless of whether we are referring to secure boot or measured boot, an important aspect to consider is securely updating the running images. With OTA updates are entirely managed by a remote server and require minimal human input. Amongst the many requirements an OTA update protocol should fulfil, we have security and fail-safe mechanisms [SV21].

Before installing an image, we should verify that it is coming from an appropriate source. This requires mutual authentication with the server, sending the update through a secure channel and verifying the authenticity and integrity of the updated image.

Another important factor to consider is having a fail-safe mechanism if the update were to fail. By having such measures put in place, the device will have an image to revert back to if it encounters any problem (*e.g.* errors that were not identified during testing, unexpected power outage duing the update, *etc.*).

In the realm of the IoT Mender and SWupdate are amongst the popular open-source projects providing secure and efficient OTA updates.

Chapter 7
Conclusion and future research

7.1 Conclusion

There is still a long way to go when it comes to IoT security. The threat model conducted on our infrastructure revealed certain security gaps that could be exploited by a malicious actor. Our research aimed at removing the threat of running unauthorised code by securing the boot system of embedded Linux systems using the secure boot protocol.

Our study revealed that secure boot is not the ideal solution to our problem. While it does prevent unauthorised code from being loaded during boot, there are known methods of bypassing this security measure, mainly through fault injection attacks. Our attempt to replicate this attack using a "homebrewed" approach was unsuccessful but commercially available glitchers should make this task a lot more easier. Furthermore, we revealed that this protocol does not provide a long-term solution.

While secure boot is not the ideal solution, it is still better than having no security measures put in place as it adds a layer of complexity for the attacker. While achieving "perfect security" is impossible, putting in place different security measures does contribute in getting closer to that goal.

7.2 Future research

The limitations in this research have indicated the following recommendations for future works:

- A similar study of the measured protocol for the cloud connector to evaluate the added security features either by implementing it along with secure boot or as an alternative. As the SoC used in the cloud connector does not have a built in TPM, securely integrating one should be the main focus of the study.

- An evaluation of the cloud connector's Printed Circuit Board (PCB). We mentioned the importance of securing the physical layer of IoT devices in Subsection 3.1.2. Unfortunately, most vendors do not make the effort in securing this layer which gives attackers a larger attack surface.

References

[BKW13] J. Buchmann, E. Karatsiolis, and A. Wiesmaier, *Introduction to public key infrastructures*. Springer, 2013, vol. 36. [Online]. Available: https://link.springer.com/book/10.1007/978-3-642-40657-7 (last visited: May 29, 2022).

[Cer09] Certicom Research, «Standards for efficient cryptography, sec 1: Elliptic curve cryptography», May 2009, Version 2.0. [Online]. Available: https://www.secg.org/sec1-v2.pdf (last visited: May 29, 2022).

[COS13] T. Charania, A. Opal, and M. Sachdev, «Analysis and design of on-chip decoupling capacitors», *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 4, pp. 648–658, 2013. [Online]. Available: https://ieeexplore.ieee.org/document/6239613 (last visited: May 29, 2022).

[Dis] Disruptive Technologies. «Disruptive Technologies - Tiny Wireless Sensors & IoT Infrastructure», [Online]. Available: https://www.disruptive-technologies.com (last visited: May 29, 2022).

[GAM+99] S. Galperin, D. C. Adams, *et al.*, *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*, RFC 2560, Jun. 1999. [Online]. Available: https://www.rfc-editor.org/info/rfc2560 (last visited: May 29, 2022).

[GSK+19] J. Gediya, J. Singh, *et al.*, «7 - open-source software», in *Software Engineering for Embedded Systems*, R. Oshana and M. Kraeling, Eds., Second Edition, Newnes, 2019, pp. 207–244. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128094488000072 (last visited: May 29, 2022).

[GWL+18] C. Geng, B. WANG, *et al.*, «Ubootkit: A worm attack for the bootloader of iot devices», presented at the BlackHat Asia 2018, 2018. [Online]. Available: https://i.blackhat.com/briefings/asia/2018/asia-18-Yang-UbootKit-A-Worm-Attack-for-the-Bootloader-of-IoT-Devices-wp.pdf (last visited: May 29, 2022).

[HPFS02] R. Housley, T. Polk, *et al.*, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, RFC 3280, May 2002. [Online]. Available: https://www.rfc-editor.org/info/rfc3280 (last visited: May 29, 2022).

[iMX21] i.MX U-Boot, *i.MX6, i.MX7 U-Boot HABv4 Secure Boot guide for SPL targets*, Oct. 20, 2021. [Online]. Available: https://source.codeaurora.org/external/imx/uboot-imx/tree/doc/imx/habv4/guides/mx6_mx7_spl_secure_boot.txt?h=lf-5.15.5-1.0.0 (last visited: May 29, 2022).

[ITU08] ITU-T, «X.680 : Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation», International Telecommunication Union - Telecommunication Standardization Sector, Standard, Nov. 2008. [Online]. Available: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.680-200811-S!!PDF-E&type=items (last visited: May 29, 2022).

[ITU19] ——, «X.509 : Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks», International Telecommunication Union - Telecommunication Standardization Sector, Standard, Oct. 2019. [Online]. Available: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.509-201910-I!!PDF-E&type=items (last visited: May 29, 2022).

[KL14] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*, 2nd. Chapman & Hall/CRC, 2014. [Online]. Available: https://dl.acm.org/doi/book/10.5555/2700550 (last visited: May 29, 2022).

[MCZ+19] F. Meneghello, M. Calore, *et al.*, «Iot: Internet of threats? a survey of practical security vulnerabilities in real iot devices», *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8182–8201, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8796409 (last visited: May 29, 2022).

[NXP17a] NXP Semiconductors, *ERR010872 ROM: Secure boot vulnerability when using the Serial Downloader*, Jul. 2017. [Online]. Available: https://community.nxp.com/t5/i-MX-Processors-Knowledge-Base/i-MX-Vybrid-Security-Vulnerability-Errata-ERR010872-ERR010873/ta-p/1120527?attachment-id=85276 (last visited: May 29, 2022).

[NXP17b] ——, *ERR010873ROM: Secure boot vulnerability when authenticating a certificate*, Jul. 2017. [Online]. Available: https://community.nxp.com/t5/i-MX-Processors-Knowledge-Base/i-MX-Vybrid-Security-Vulnerability-Errata-ERR010872-ERR010873/ta-p/1120527?attachment-id=85277 (last visited: May 29, 2022).

[NXP18a] ——, *Code-Signing Tool - User's Guide*, Sep. 2018.

[NXP18b] ——, *High Assurance Boot Version 4 Application Programming Interface Reference Manual*, 2018.

[NXP18c] ——, *i.MX 7Dual Applications Processor Reference Manual*, Jan. 2018.

[NXP20] ——, *i.MX Secure Boot on HABv4 Supported Devices*, Jun. 2020.

[OC14] C. O'Flynn and Z. D. Chen, «Chipwhisperer: An open-source platform for hardware embedded security research», in *International Workshop on Constructive Side-Channel Analysis and Secure Design*, Springer, 2014, pp. 243–260. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-10175-0_17 (last visited: May 29, 2022).

[OFl16] C. O'Flynn, «Fault injection using crowbars on embedded systems», *Cryptology ePrint Archive*, 2016. [Online]. Available: https://eprint.iacr.org/2016/810 (last visited: May 29, 2022).

[PB19] C. Paulsen and R. Byers, *Glossary of key information security terms*, en, Jul. 2019. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.7298r3.pdf (last visited: May 29, 2022).

[Res18] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446 (last visited: May 29, 2022).

[Sho14] A. Shostack, *Threat Modeling: Designing for Security*, 1st. Wiley Publishing, 2014. [Online]. Available: https://dl.acm.org/doi/10.5555/2829295 (last visited: May 29, 2022).

[Sin21] S. Sinha. «State of IoT 2021: Number of connected IoT devices growing 9% to 12.3 billion globally, cellular IoT now surpassing 2 billion», IoT Analytics. (Sep. 22, 2021), [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/ (last visited: May 29, 2022).

[Sma15] N. P. Smart, *Cryptography Made Simple*, 1st. Springer Publishing Company, Incorporated, 2015. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-21936-3 (last visited: May 29, 2022).

[SSW04] A. N. Sloss, D. Symes, and C. Wright, «Chapter 1 - ARM embedded systems», in *ARM System Developer's Guide*, ser. The Morgan Kaufmann Series in Computer Architecture and Design, A. N. SLOSS, D. SYMES, and C. WRIGHT, Eds., Burlington: Morgan Kaufmann, 2004, pp. 2–16. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558608740500022 (last visited: May 29, 2022).

[SV21] C. Simmonds and F. Vasquez, *Mastering Embedded Linux Programming*, 3rd ed. Packt Publishing, 2021. [Online]. Available: https://www.oreilly.com/library/view/mastering-embedded-linux/9781789530384/ (last visited: May 29, 2022).

[Tru19] Trusted Computing Group, *Trusted Platform Module Library Specification, Family "2.0"*, Nov. 2019. [Online]. Available: https://trustedcomputinggroup.org/resource/tpm-library-specification/ (last visited: May 29, 2022).

[TSW16] N. Timmers, A. Spruyt, and M. Witteman, «Controlling PC on ARM Using Fault Injection», in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016, pp. 25–35. [Online]. Available: https://ieeexplore.ieee.org/document/7774479 (last visited: May 29, 2022).

[WO22] J. v. Woudenberg and C. O'Flynn, *The Hardware Hacking Handbook Breaking Embedded Security with hardware attacks*. No Starch Press, 2022. [Online]. Available: https://nostarch.com/hardwarehacking (last visited: May 29, 2022).

[Yur18] Yuri. «[Secure Boot CST 3.0.1]: the validity period of the corresponding certificates». (Jun. 20, 2018), [Online]. Available: https://community.nxp.com/t5/i-MX-Processors/Secure-Boot-CST-3-0-1-the-validity-period-of-the-corresponding/m-p/804589 (last visited: May 29, 2022).

[ZAH+14] R. Zhou, Z. Ai, *et al.*, «Data Integrity Checking for iSCSI with Dm-verity», in *Advanced Technologies, Embedded and Multimedia for Human-centric Computing*, Y.-M. Huang, H.-C. Chao, *et al.*, Eds., Dordrecht: Springer Netherlands, 2014, pp. 691–697. [Online]. Available: https://link.springer.com/chapter/10.1007/978-94-007-7262-5_79 (last visited: May 29, 2022).

[ZDC+12] L. Zussa, J.-M. Dutertre, *et al.*, «Investigation of timing constraints violation as a fault injection means», in *27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France*, Citeseer, 2012, pp. 1–6. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.726.6326&rep=rep1&type=pdf (last visited: May 29, 2022).

[AAB+17] M. Antonakakis, T. April, *et al.*, «Understanding the mirai botnet», in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis (last visited: May 29, 2022).

Appendix A

High assurance boot and the code signing tool

A.1 Example of a command sequence file

```
[Header]
    Version = 4.2
    Hash Algorithm = sha256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
    Engine = CAAM

[Install SRK]
    # Index of the key location in the SRK table to be installed
    File = "../crts/SRK_1_2_3_4_table.bin"
    Source index = 0

[Install CSFK]
    # Key used to authenticate the CSF data
    File = "../crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem"

[Authenticate CSF]

[Install Key]
    # Key slot index used to authenticate the key to be installed
    Verification index = 0
    # Target key slot in HAB key store where key will be installed
    Target Index = 2
    # Key to install
    File= "../crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem"

[Authenticate Data]
    # Key slot index used to authenticate the image data
    Verification index = 2
    Blocks =  <start_address> <offset> <length> "path_to_image"

[Unlock]
    # Leaves SRK revocation unlocked
    Engine = OCOTP
```

```
36     Features = SRK Revoke
```

Listing A.1: Command sequence file

A.2 Using the code signing tool to sign an image

```
1     ~/cst/linux64/bin$ ./cst --output spl_csf.bin --input spl.csf
2
3     ~/cst/linux64/bin$ cat SPL spl_csf.bin > signed_spl
```

Listing A.2: Signing an SPL image using the CST

A.3 HAB status indicating a failed image verification

HAB event 1 indicates an image verification failure was recorded during boot. HAB events 2 to 4 indicate which portion of the image loaded failed the authentication:

- HAB event 1: authentication of the Image Vector Table (IVT) failed
- HAB event 2: authentication of the first byte of the boot data failed
- HAB event 3: authentication of the first 4 bytes of the SPL failed

```
1  => hab_status
2
3  Secure boot disabled
4
5  HAB Configuration: 0xf0, HAB State: 0x66
6
7  --------- HAB Event 1 -----------------
8  event data:
9          0xdb 0x00 0x08 0x42 0x33 0x22 0x0a 0x00
10
11 STS = HAB_FAILURE (0x33)
12 RSN = HAB_INV_ADDRESS (0x22)
13 CTX = HAB_CTX_AUTHENTICATE (0x0A)
14 ENG = HAB_ENG_ANY (0x00)
15
16
17 --------- HAB Event 2 -----------------
18 event data:
19         0xdb 0x00 0x14 0x42 0x33 0x0c 0xa0 0x00
20         0x00 0x00 0x00 0x00 0x00 0x91 0x14 0x00
21         0x00 0x00 0x00 0x20
22
23 STS = HAB_FAILURE (0x33)
24 RSN = HAB_INV_ASSERTION (0x0C)
25 CTX = HAB_CTX_ASSERT (0xA0)
```

```
26 ENG = HAB_ENG_ANY (0x00)
27
28
29 --------- HAB Event 3 -----------------
30 event data:
31         0xdb 0x00 0x14 0x42 0x33 0x0c 0xa0 0x00
32         0x00 0x00 0x00 0x00 0x00 0x91 0x14 0x20
33         0x00 0x00 0x00 0x01
34
35 STS = HAB_FAILURE (0x33)
36 RSN = HAB_INV_ASSERTION (0x0C)
37 CTX = HAB_CTX_ASSERT (0xA0)
38 ENG = HAB_ENG_ANY (0x00)
39
40
41 --------- HAB Event 4 -----------------
42 event data:
43         0xdb 0x00 0x14 0x42 0x33 0x0c 0xa0 0x00
44         0x00 0x00 0x00 0x00 0x00 0x91 0x20 0x00
45         0x00 0x00 0x00 0x04
46
47 STS = HAB_FAILURE (0x33)
48 RSN = HAB_INV_ASSERTION (0x0C)
49 CTX = HAB_CTX_ASSERT (0xA0)
50 ENG = HAB_ENG_ANY (0x00)
```

Listing A.3: HAB status for a failed image verification

Appendix B

Developing a voltage glitcher on an FPGA

The glitcher was programmed using the VDHL[1] language and the Xilinx Vivado Integrated Development Environment[2].

B.1 Code written for the voltage glitcher

B.1.1 Source code

```
1  -- Author: Nahom Belay
2  -- Module Name: config - Behavioral
3
4  library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;
6
7  entity config is
8    Port (
9    clk : in STD_LOGIC;
10   rst : in STD_LOGIC;
11   config_duration: in std_logic;
12   config_delay: in std_logic;
13   mode: in std_logic;
14   data: in std_logic_vector (7 downto 0);
15   output_duration: out std_logic_vector (7 downto 0);
16   output_delay: out std_logic_vector (7 downto 0));
17 end config;
18
19 architecture Behavioral of config is
20
21 signal duration : std_logic_vector (7 downto 0) := "01000000";
22 signal delay : std_logic_vector (7 downto 0) := "01000000";
23
24 begin
25     process
26         begin
```

[1] Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
[2] https://www.xilinx.com/products/design-tools/vivado.html

```
27              wait until rising_edge(clk);
28                  if mode = '1' then
29                      if config_delay = '1' then
30                          delay <= data;
31                      end if;
32
33                      if config_duration = '1' then
34                          duration <= data;
35                      end if;
36
37                      if rst = '1' then
38                          duration <= "01000000";
39                          delay <= "01000000";
40                      end if;
41                  end if;
42
43
44                  output_delay <= delay;
45                  output_duration <= duration;
46
47     end process;
48
49
50 end Behavioral;
```

```
1  -- Author: Nahom Belay
2  -- Module Name: main_glitcher - Behavioral
3
4  library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;
6
7  -- Uncomment the following library declaration if using
8  -- arithmetic functions with Signed or Unsigned values
9  --use IEEE.NUMERIC_STD.ALL;
10
11 -- Uncomment the following library declaration if instantiating
12 -- any Xilinx leaf cells in this code.
13 --library UNISIM;
14 --use UNISIM.VComponents.all;
15
16 entity main_glitcher is
17     Port ( clk : in STD_LOGIC;
18            rst : in STD_LOGIC;
19            mode : in STD_LOGIC;
20            trigger: in STD_LOGIC;
21            config_duration : in STD_LOGIC;
22            config_delay : in STD_LOGIC;
23            data : in STD_LOGIC_VECTOR (7 downto 0);
24            output_signal : out STD_LOGIC;
25            led : out std_logic
26        );
27     end main_glitcher;
```

```
28
29 architecture Behavioral of main_glitcher is
30     component config is
31     port(
32         clk : in STD_LOGIC;
33         rst : in STD_LOGIC;
34         config_duration: in std_logic;
35         config_delay: in std_logic;
36         mode: in std_logic;
37         data: in std_logic_vector (7 downto 0);
38         output_duration: out std_logic_vector (7 downto 0);
39         output_delay: out std_logic_vector (7 downto 0)
40     );
41     end component;
42
43     component glitch is
44     port(
45         trigger: in std_logic;
46         clk: in std_logic;
47         rst: in std_logic;
48         mode: in std_logic;
49         delay: in std_logic_vector (7 downto 0);
50         duration: in std_logic_vector (7 downto 0);
51         output_signal: out std_logic;
52         led: out std_logic
53     );
54     end component;
55
56     --Auxiliary signal
57     signal configured_duration: std_logic_vector (7 downto 0);
58     signal configured_delay: std_logic_vector (7 downto 0);
59
60
61 begin
62
63     --Instanciate config component
64     instance_config : config port map (
65         clk => clk,
66         rst => rst,
67         mode => mode,
68         config_delay => config_delay,
69         config_duration => config_duration,
70         data => data,
71         output_delay => configured_delay,
72         output_duration => configured_duration
73     );
74
75     --Instanciate glitch component
76     instance_glitch : glitch port map (
77         clk => clk,
78         rst => rst,
79         mode => mode,
```

```
80          trigger => trigger,
81          delay => configured_delay,
82          duration => configured_duration,
83          output_signal => output_signal,
84          led => led
85      );
86
87      process
88          begin
89              wait until rising_edge(clk);
90      end process;
91
92  end Behavioral;
```

```
1   -- Author: Nahom Belay
2   -- Module Name: glitch - Behavioral
3
4   library IEEE;
5   use IEEE.STD_LOGIC_1164.ALL;
6   use IEEE.NUMERIC_STD.ALL;
7   use IEEE.std_logic_arith.all;
8   use IEEE.std_logic_unsigned.all;
9
10  entity glitch is
11    Port (
12    trigger: in std_logic;
13    clk: in std_logic;
14    rst: in std_logic;
15    mode: in std_logic;
16    delay: in std_logic_vector (7 downto 0);
17    duration: in std_logic_vector (7 downto 0);
18    output_signal: out std_logic;
19    led : out std_logic
20    );
21  end glitch;
22
23  architecture Behavioral of glitch is
24
25  constant configuration_in_rising_edges: std_logic_vector (31 downto 0)
        := "00000000000000000000000000000001";
26  --delay in milliseconds (multiply by 100000 not 1000000 because la
        periode is 10ns)
27  constant configuration_in_milliseconds : std_logic_vector (31 downto 0)
         := "00000000000000011000011010100000";
28  --delay in microseconds (multiply by 100 not 1000 for same reason above
        )
29  constant configuration_in_microseonds : std_logic_vector (31 downto 0)
        := "00000000000000000000000001100100";
30
31  constant glitch_mode : std_logic := '0';
32
```

```
33 constant multiplication_factor_duration : std_logic_vector (31 downto
       0) := configuration_in_rising_edges;
34 constant multiplication_factor_delay : std_logic_vector (31 downto 0)
       := configuration_in_rising_edges;
35
36 --glitch approximxated to around 200 ms so delay will be 600ms + delay
       configured from config_block
37 --constant base_delay : std_logic_vector (15 downto 0) :=
       "0000000011001000";
38 constant base_delay : std_logic_vector (15 downto 0) :=
       "0000000000000001";
39
40 -- delay in number of rising edges
41 --constant multiplication_factor_duration : std_logic_vector (31 downto
        0) := "00000000000000000000000000000001";
42 --constant multiplication_factor_delay : std_logic_vector (31 downto 0)
        := "00000000000000000000000000000001";
43
44 --delay in milliseconds (multiply by 100000 not 1000000 because la
       periode is 10ns)
45 --constant multiplication_factor_duration : std_logic_vector (31 downto
        0) := "00000000000000011000011010100000";
46 --constant multiplication_factor_delay : std_logic_vector (31 downto 0)
        := "00000000000000011000011010100000";
47
48 --delay in microseconds (multiply by 100 not 1000 for same reason above
       )
49 --constant multiplication_factor_duration : std_logic_vector (31 downto
        0) := "00000000000000000000000001100100";
50 --constant multiplication_factor_delay : std_logic_vector (31 downto 0)
        := "00000000000000000000000001100100";
51
52 constant idle_state : std_logic_vector (2 downto 0) := "000" ;
53 constant initial_delay : std_logic_vector (2 downto 0) := "001" ;
54 constant waiting_trigger_glitch_state: std_logic_vector (2 downto 0) :=
        "010" ;
55 constant glitch_state : std_logic_vector (2 downto 0) := "011" ;
56 constant holdoff : std_logic_vector (2 downto 0) := "100" ;
57
58 --constant holdoff_duration: std_logic_vector (31 downto 0) :=
       "00000000000001111010000100100000";
59 --equivalent to 300000000 rising edges which is about 3s I think
60 constant holdoff_duration: std_logic_vector (31 downto 0) :=
       "00010001111000011010001100000000";
61
62 constant normal_output: std_logic := '1';
63 constant glitch_output: std_logic := '0';
64
65 signal delay_counter: std_logic_vector (63 downto 0) := (others => '0')
       ;
66 signal duration_counter: std_logic_vector (63 downto 0) := (others =>
       '0');
```

```
67 signal glitch_register: std_logic := '0';
68 signal state: std_logic_vector (2 downto 0) := (others => '0');
69
70 begin
71     process
72         begin
73             wait until rising_edge(clk);
74             if mode = glitch_mode then
75
76                 case state is
77                     when idle_state =>
78                         if trigger = '1' then
79                             glitch_register <= normal_output;
80                             led <= '1';
81                             state <= initial_delay ;
82                             delay_counter <= (others => '0');
83                             --delay_counter <= delay_counter + "1";
84                         end if;
85
86                     when initial_delay =>
87                         glitch_register <= normal_output;
88                         delay_counter <= delay_counter + "1";
89                         if delay_counter = ( base_delay *
    multiplication_factor_delay - "1") then
90                             state <= waiting_trigger_glitch_state;
91                             delay_counter <= (others => '0');
92                         end if;
93
94                     when waiting_trigger_glitch_state =>
95                         glitch_register <= normal_output;
96                         delay_counter <= delay_counter + "1";
97                         if delay_counter = (delay  *
    multiplication_factor_delay - "1") then
98                             glitch_register <=  glitch_output;
99                             state <= glitch_state;
100                            delay_counter <= (others => '0');
101                        end if;
102
103                    when glitch_state =>
104                        duration_counter <= duration_counter + "1";
105                        glitch_register <= glitch_output;
106                        if duration_counter = (duration *
    multiplication_factor_duration - "1") then
107                            duration_counter <= (others => '0');
108                            glitch_register <= normal_output;
109                            state <= holdoff;
110                        end if;
111
112                    when holdoff =>
113                        duration_counter <= duration_counter + "1";
114                        glitch_register <= normal_output;
115                        if duration_counter = holdoff_duration then
```

```
116                             state <= idle_state;
117                             duration_counter <= (others => '0');
118                         end if;
119                     when others => null;
120                 end case;
121 
122                 if rst = '1' then
123                     glitch_register <= glitch_output;
124                     delay_counter <= (others => '0');
125                     duration_counter <= (others => '0');
126                     led <= '0';
127                     state <= idle_state;
128                 end if;
129 
130             end if;
131 
132             output_signal <= glitch_register;
133    end process;
134 
135 end Behavioral;
```

B.1.2 Testbench code

```
1  -- Author: Nahom Belay
2  -- Module Name: main_glicher_test - Behavioral
3 
4  library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;
6 
7  entity main_glitcher_test is
8  --  Port ( );
9  end main_glitcher_test;
10 
11 architecture Behavioral of main_glitcher_test is
12     component main_glitcher is
13     port (
14         clk : in STD_LOGIC;
15         rst : in STD_LOGIC;
16         trigger: in STD_LOGIC;
17         mode: in std_logic;
18         config_duration: in std_logic;
19         config_delay: in std_logic;
20         data: in std_logic_vector (7 downto 0);
21         output_signal : out STD_LOGIC;
22         led : out std_logic
23     );
24     end component;
25 
26     --Inputs
27     signal clk :  STD_LOGIC;
28     signal rst :  STD_LOGIC;
29     signal mode:  STD_LOGIC;
```

```
30      signal trigger: STD_LOGIC;
31      signal config_duration: std_logic;
32      signal config_delay: std_logic;
33      signal data: STD_LOGIC_VECTOR (7 downto 0);
34
35      --output
36      signal output_signal: std_logic;
37      signal led: std_logic;
38
39      constant CLK_period : time := 10 ns;
40
41  begin
42      uut: main_glitcher port map (
43          clk => clk,
44          rst => rst,
45          mode => mode,
46          trigger => trigger,
47          config_delay => config_delay,
48          config_duration => config_duration,
49          data => data,
50          output_signal => output_signal,
51          led => led
52      );
53
54      -- Clock process definitions
55      CLK_process :process
56      begin
57          clk <= '0';
58          wait for CLK_period/2;
59          clk <= '1';
60          wait for CLK_period/2;
61      end process;
62
63      -- Stimulus process
64      stim_proc: process
65      begin
66          mode <= '1';
67          trigger <= '0';
68          rst <= '0';
69          config_delay <= '0';
70          config_duration <= '0';
71          data <= "00000000";
72          wait for 100 ns;
73          data <= "00001001";
74          wait for 10 ns;
75          config_delay <= '1';
76          wait for 20 ns;
77          config_delay <= '0';
78          wait for 20 ns;
79          config_duration <= '1';
80          data <= "00000100";
81          wait for 10 ns;
```

```
82          config_duration <= '0';
83          mode <= '0';
84          data <= "11111111";
85          wait for 10 ns;
86          trigger <= '1';
87          wait for 20 ns;
88          trigger <= '0';
89          wait for 10 us;
90      end process;
91  end Behavioral;
```