Project Report

On

Secure Boot in Embedded Systems

Submitted
Inpartial fulfilment
Fortheaward of the Degree of

PG-Diploma in Embedded Systems and Design (PG-DESD)

C-DAC,ACTS (Pune)

Guided By:

Mr. Arif Sir

SubmittedBy:

Kapil Kumar(230940130033)

Sanket Patil (230940130052)

Diwakar Varansi (230940130022)

Amey Raghatate (230940130010)

Centre for Development of Advanced Computing (C-DAC), ACTS (Pune-411008)

Acknowledgement

This is to acknowledge our in debtedness to our Project Guide, **Mr. Surendra Billa** C-DAC ACTS, Pune for her constant guidance and helpful suggestion for preparing this project **Secure Boot in Embedded Systems.**We express our deep gratitude towards her for inspiration, personal involvement, constructive criticism that he provided us along with technical guidance during the course of this project.

We take this opportunity to thank Head of the department **Mr.Gaur Sunder** for providing us such a great infrastructure and environment for our overall development.

We express sincere thanks to **Mrs. Namrata Ailawar**, Process Owner, for their kind cooperation and extendible support towards the completion of our project.

It is our great pleasure in expressing sincere and deep gratitude towards **Mrs. Risha P. R. (Program Head)** and **Mrs. Srujana Bhamidi** (Course Coordinator, PG-DESD) for their valuable guidance and constant support through out this work and help to pursue additional studies.

Also, our warm thanks to **C-DAC ACTS Pune**, which provided us this opportunity to carry out, this prestigious Project and enhance our learning in various technical fields.

Kapil Kumar(230940130033)

Sanket Patil (230940130052)

Diwakar Varansi (230940130022)

Amey Raghatate (230940130010)

ABSTRACT

With the proliferation to the need of prioritize the overall system security is more imperative than ever. Internet connectivity brought by the IoT exposes such previously isolated internal device structures to cyber-attacks through the Internet, which opens new attack vectors and vulnerabilities. For example, a malicious user can modify the firmware or operating system by using a remote connection, aiming to deactivate standard defenses against malware. The criticality of applications, for example, in the Industrial IoT (IIoT) further underlines the need to ensure the integrity of the embedded software.

One common approach to ensure system integrity is to verify the operating system and application software during the boot process. However, safety-critical IoT devices have constrained boot-up times, and home IoT devices should become available quickly after being turned on. Therefore, the boot-time can affect the usability of a device. This paper analyses performance trade-offs of secure boot for medium-scale embedded systems, such as Beaglebone. We evaluate two secure boot techniques, one is only software-based, and the second is supported by a hardware-based cryptographic storage unit. For the software-based method, we show that secure boot merely increases the overall boot time by 4%. Moreover, the additional cryptographic hardware storage increases the boot-up time by 36%.

Index Terms—Embedded Systems, Secure Boot, System Security

TableofContents

Chapter 1
Introduction

1.1 Introduction

Secure boot is a critical feature in embedded systems that ensures that only trusted software can run on the system. It protects against malware and other attacks by verifying the integrity and authenticity of the software before it is loaded. At its core, secure boot involves verifying the digital signature of the software before it is loaded. This ensures that the software has not been tampered with and that it comes from a trusted source.

To implement secure boot, a hardware root of trust is required. This can be provided by a secure boot ROM, which is a small piece of code that is burned into the hardware and is responsible for verifying the digital signature of the software.

The secure boot process typically involves several stages. At each stage, the software is verified before it is loaded. This ensures that even if one stage is compromised, the rest of the system remains secure.

The first stage of secure boot is typically the boot ROM, which is responsible for loading the second stage bootloader. The second stage bootloader is responsible for verifying the digital signature of the operating system kernel.

Once the kernel has been loaded, it is responsible for verifying the digital signature of the user-space applications before they are executed. This ensures that only trusted applications are run on the system.

In addition to verifying the digital signature of the software, secure boot can also involve other security features, such as hardware-enforced memory protection and encryption. These features help protect against attacks that try to bypass or exploit vulnerabilities in the system.

In conclusion, secure boot is a critical feature in embedded systems that ensures that only trusted software can run on the system. By verifying the digital signature of the software before it is loaded, secure boot protects against malware and other attacks.

With a hardware root of trust and a multistage verification process, secure boot helps ensure the security and integrity of embedded systems.

into printed, handwritten cursive, handwritten discrete and semi-printed by an input classifier. Text in the image is predicted using corresponding models. The predicted text is therefore available as machine editable text which can be retrieved easily whenever necessary.

1.2 Objective

The objectives of the project work are as -

- To test a secure boot.
- To have better understanding of Hashing, Encryption, Decryption, Digital Signature,Digital Certificate, Booting Sequence.
- To understand the Boot process and steps involved in the Booting

Chapter 2

Methodology and Techniques

3.1 Methodology:

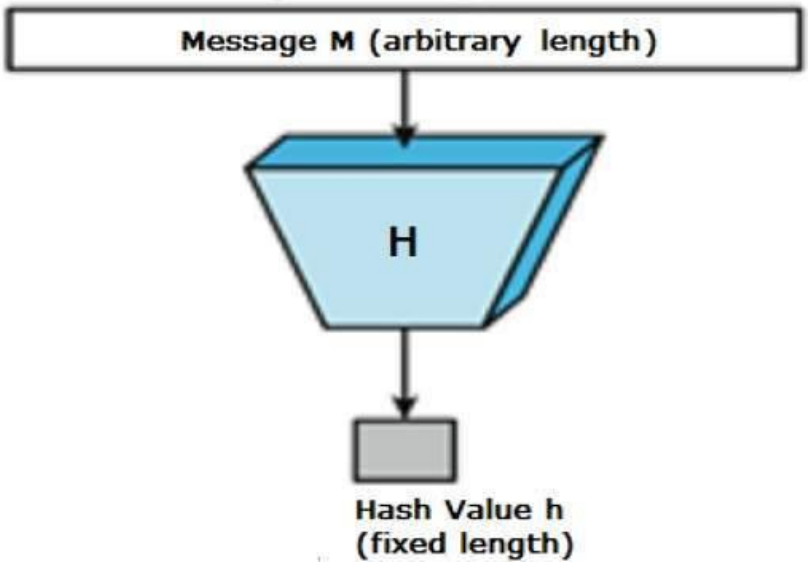
3.1.1 Hashing:

Hashing is the process of the transforming any given key or a string of character into another value. This is usually represented by a shorter, fixed-length value or a key that represents and make it easier to find or employ the original string.

3.1.2 Hash-Function:

A hash function is a mathematical function that converts a numerical input value into another compressed numerical value. The input to the hash function is of arbitrary length but output is always of fixed length.

Values returned by a hash function are called **message digest** or simply **hash values**. The following picture illustrated hash function.



Original Data	+	Hash Function	=	Hash Value/Digest
For eg Password 123				D3%f@43*1

Website use hashing to store the passwords

When a new user signs-up, the new password is passed through the hash function and the digest is stored on the server

So when the next time the user comes to login then the input password is passed through the function again and the digest is compared to the one which was stored earlier on the server.

If the Hash calculated digest is same as before then the password is right and further the login is been allowed to the user if doesn't matches then the login is not allowed to the user

D3%f@g43*1	==	D3%f@g43*1
Re-calculated Digest		Hash Stored on the server

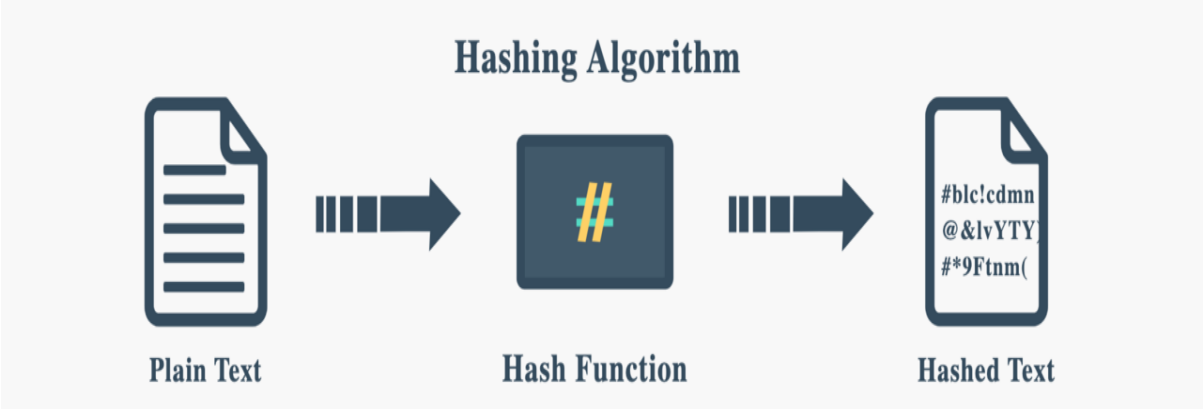
Login is **allowed**

R2#h9Ln7q&	!=	D3%f@g43*1
Re-calculated Digest		Hash Stored on the server

Login is **denied**

3.1.3 Hash Algorithm:

A set of instructions that takes in a piece of data of any length, and outputs a fixed length of data



***MD5*:**

This is the fifth version of the Message Digest algorithm. MD5 creates 128-bit outputs. MD5 was a very commonly used hashing algorithm. That was until weaknesses in the algorithm started to surface. Most of these weaknesses manifested themselves as collisions. Because of this, MD5 began to be phased out.

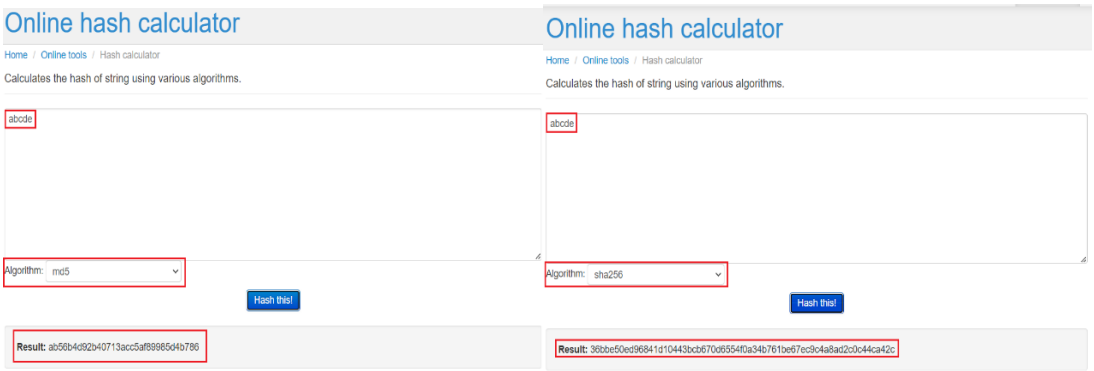
SHA(Secure Hash Algorithm):

***SHA-1*:**

This is the second version of the Secure Hash Algorithm standard, SHA-0 being the first. SHA-1 creates 160-bit outputs. SHA-1 is one of the main algorithms that began to replace MD5, after vulnerabilities were found. SHA-1 gained widespread use and acceptance. SHA-1 was actually designated as a FIPS 140 compliant hashing algorithm.

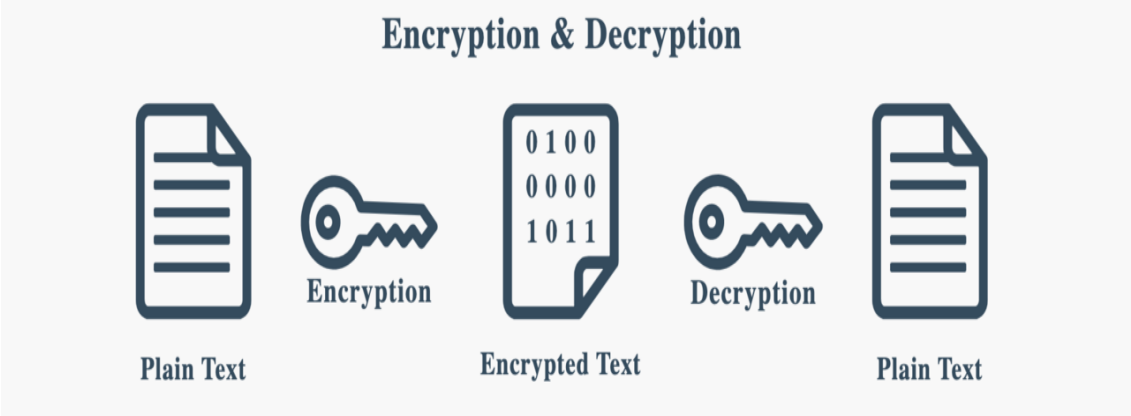
***SHA-2*:**

This is actually a suite of hashing algorithms. The suite contains SHA-224, SHA-256, SHA-384, and SHA-512. Each algorithm is represented by the length of its output. SHA-2 algorithms are more secure than SHA-1 algorithms, but SHA-2 has not gained widespread use.

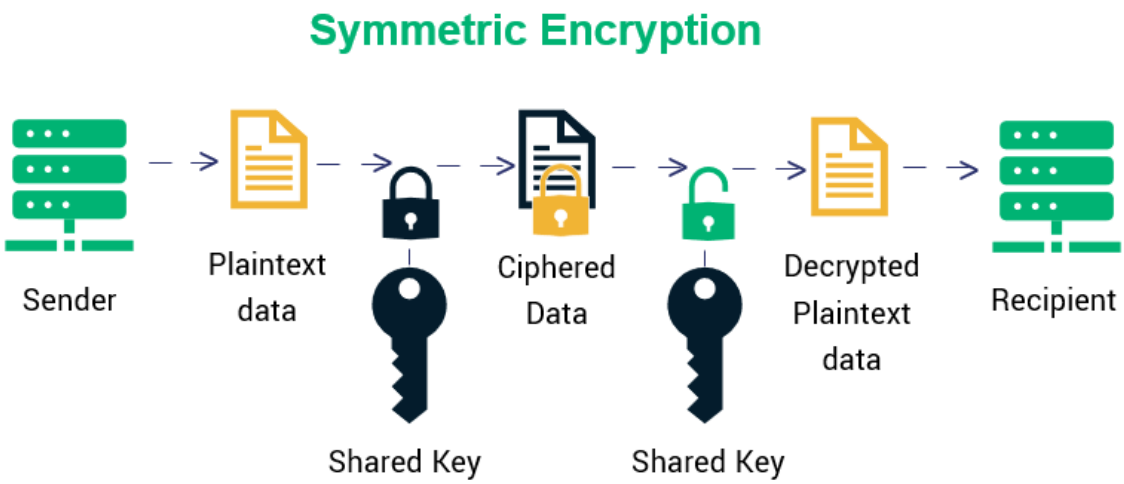


3.2.1 Encryption:

Encryption is a form of data security in which information is converted to ciphertext. Only authorized people who have the **key** can decrypt the code and access the original plaintext information.



3.2.2 Types of Encryption:

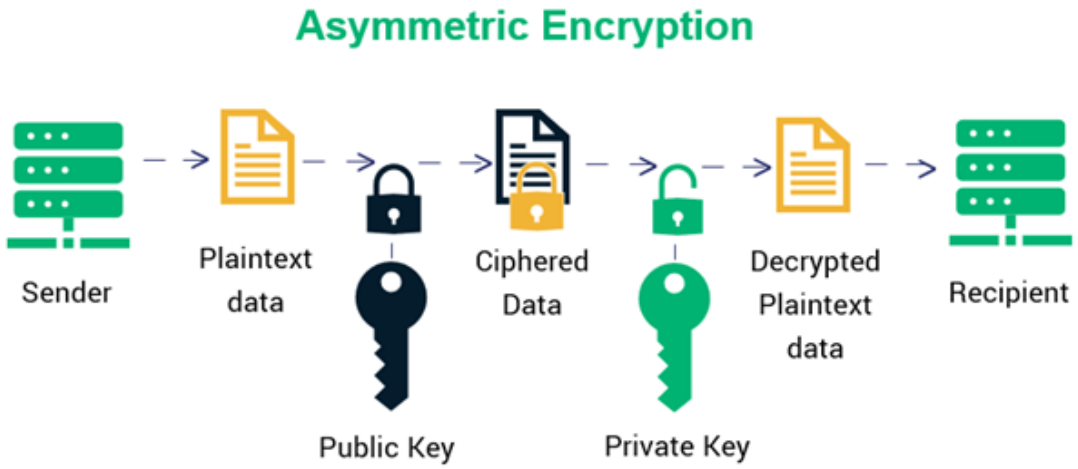


Symmetric Encryption

This type of encryption is reciprocal, meaning that the **same key** is used to encrypt and decrypt data. This is great for large batches of data but has issues in terms of key distribution and management.

Symmetric encryption algorithms include AES-128, AES-192, and AES-256. Because it is less complex and executes faster, symmetric encryption is the preferred method for transmitting data in bulk.

Asymmetric Encryption:

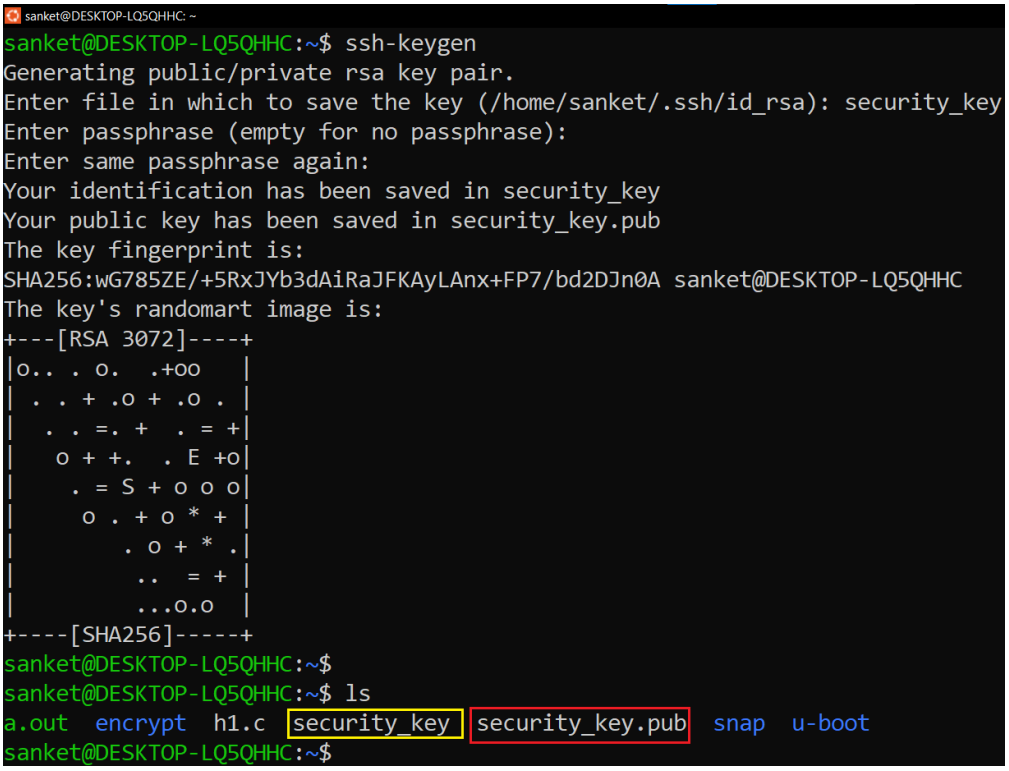


Asymmetric encryption is a type of encryption that uses two separates yet mathematically related keys to encrypt and decrypt data. The **public key** encrypts data while its corresponding **private key** decrypts it. This is why it's also known as **public key encryption**, **public key cryptography**, and **asymmetric key encryption**.

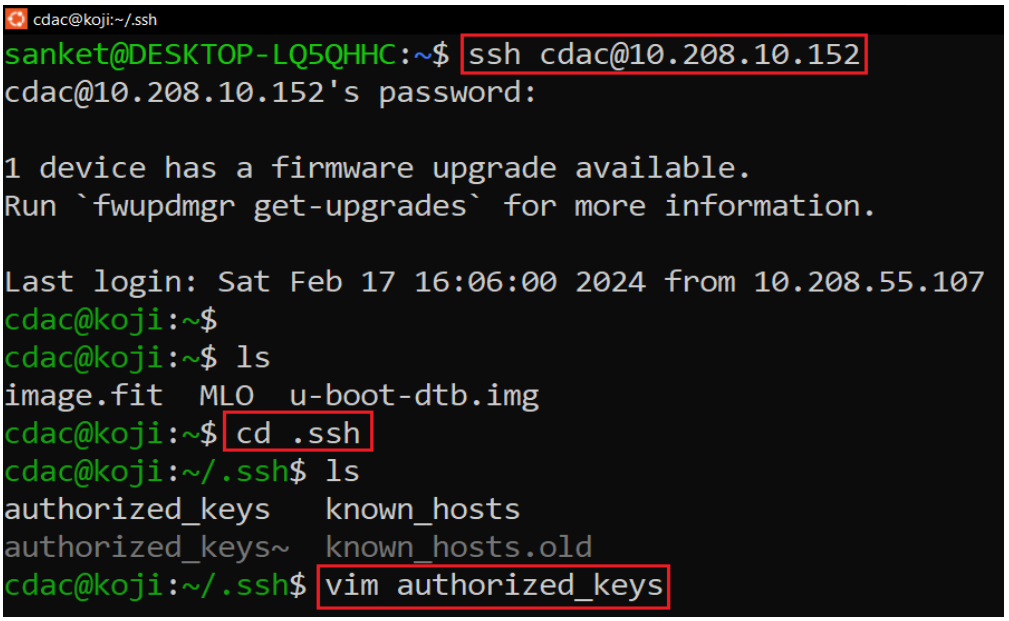
The public key is open to everyone. Anyone can access it and encrypt data with it. However, once encrypted, that data can only be unlocked by using the corresponding private key. As you can imagine, the private key must be kept secret to keep it from becoming compromised. So, only the authorized person, server, machine, or instrument has access to the private key.

3.2.3 Generating Keys:

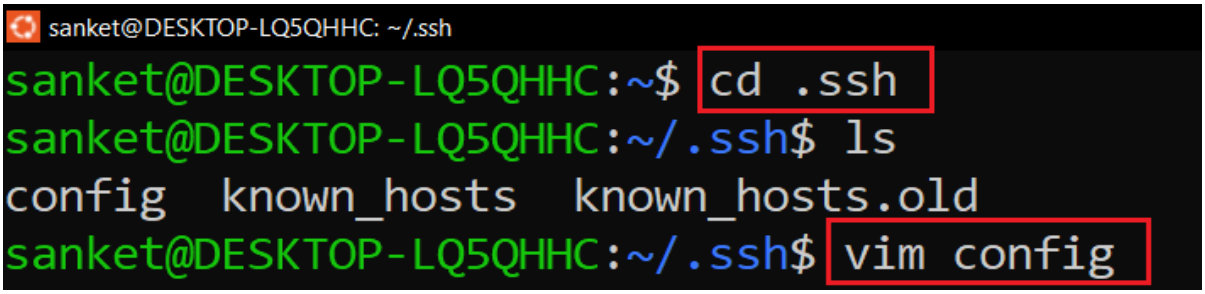
Here we have created the keys i.e Public and Private key

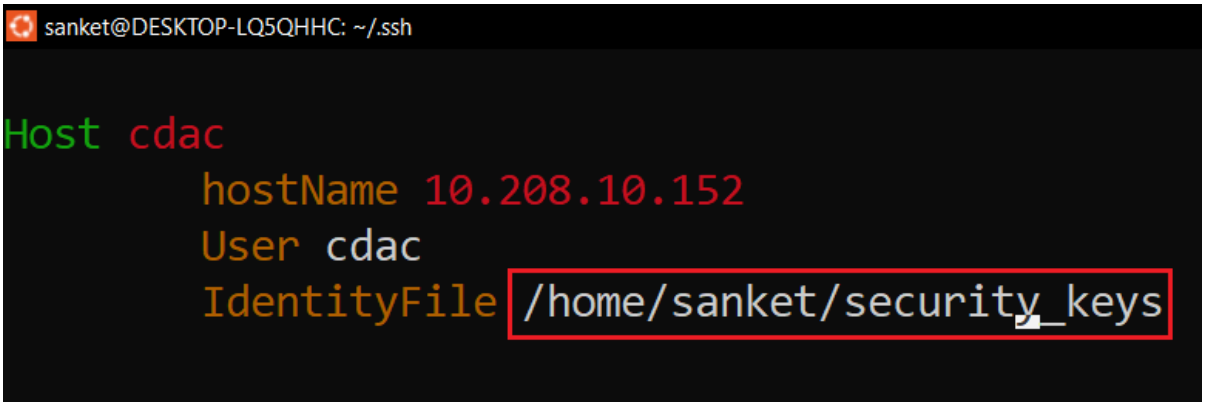


Here in above Image two keys get generated one is **Private key**(**Yellow** indication) and other one is the **Public key**(**Red** indication) after generation of the keys share the Public key with the server and get the access of the server without entering servers password one time we need to enter the server password to put the Public key in the path **.ssh/authorized keys** as shown in below image

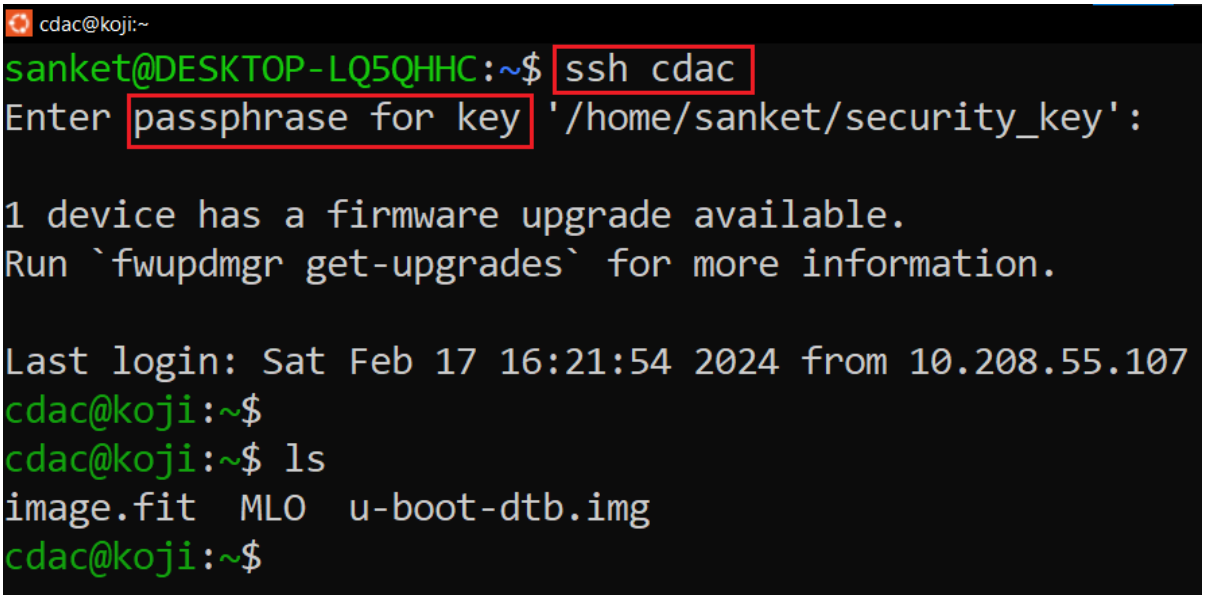


Then in client side open the config file and give the Host and its IP address and the path where the private key is stored this is basically for our convenience writing in such format





So after completing all the process we can directly access our server here our server is CDAC without entering its password here passphrase is used to protect our private key.



3.3 Digital Signature:

A digital signature is a mathematical technique used to validate the authenticity and integrity of a message, software, or digital document.

1. **Key Generation Algorithms:** Digital signature is electronic signatures, which assure that the message was sent by a particular sender. While performing digital transactions authenticity and integrity should be assured, otherwise, the data can be altered or someone can also act as if he was the sender and expect a reply.
2. **Signing Algorithms:** To create a digital signature, signing algorithms like email programs create a one-way hash of the electronic data which is to be signed. The signing algorithm then encrypts the hash value using the private key (signature key). This encrypted hash along with other information like the hashing algorithm is the digital signature. This digital signature is appended with the data and sent to the verifier. The reason for encrypting the hash instead of the entire message or document is that a hash function converts any arbitrary input into a much shorter fixed-length value. This saves time as now instead of signing a long message a shorter hash value has to be signed and moreover hashing is much faster than signing.
3. **Signature Verification Algorithms:** Verifier receives Digital Signature along with the data. It then uses Verification algorithm to process on the digital signature and the public key (verification key) and generates some value. It also applies the same hash function on the received data and generates a hash value. If they both are equal, then the digital signature is valid else it is invalid.

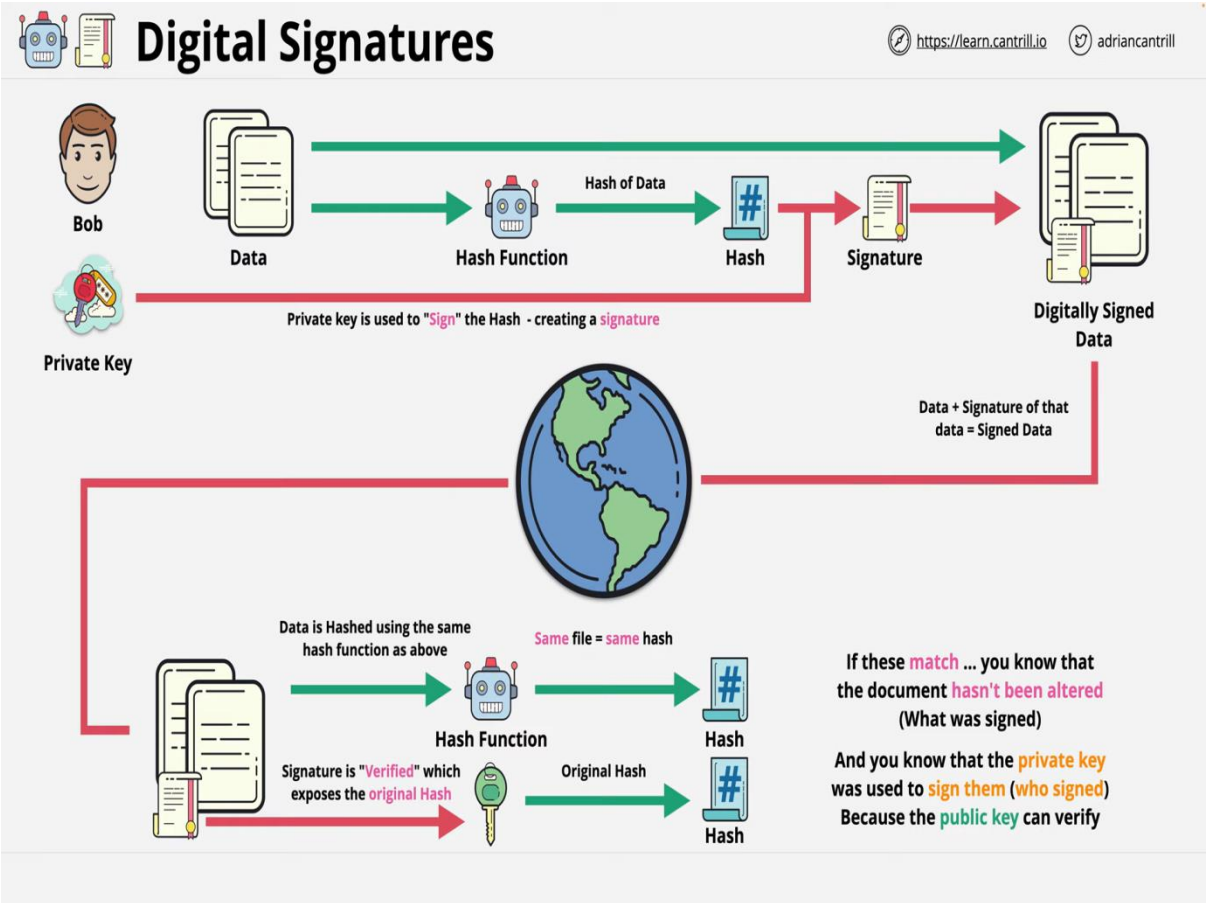
The steps followed in creating digital signature are

1. Message digest is computed by applying hash function on the message and then message digest is encrypted using private key of sender to form the digital signature. (digital signature = encryption (private key of sender, message digest) and message digest = message digest algorithm(message)).
2. Digital signature is then transmitted with the message(message + digital signature is transmitted)
3. Receiver decrypts the digital signature using the public key of sender(This

assures authenticity, as only sender has his private key so only sender can encrypt using his private key which can thus be decrypted by sender's public key).

4. The receiver now has the message digest.
5. The receiver can compute the message digest from the message (actual message is sent with the digital signature).
6. The message digest computed by receiver and the message digest (got by decryption on digital signature) need to be same for ensuring integrity.

Message digest is computed using one-way hash function, i.e. a hash function in which computation of hash value of a message is easy but computation of the message from hash value of the message is very difficult.



Assurances about digital signatures:

The definitions and words that follow illustrate the kind of assurances that digital signatures offer.

0 seconds of 0 secondsVolume 0%

1. **Authenticity:** The identity of the signer is verified.
2. **Integration:** Since the content was digitally signed, it hasn't been altered or interfered with.
3. **Non-repudiation:** demonstrates the source of the signed content to all parties. The act of a signer denying any affiliation with the signed material is known as repudiation.
4. **Notarization:** Under some conditions, a signature in a Microsoft Word, Microsoft Excel, or Microsoft PowerPoint document that has been time-stamped by a secure time-stamp server is equivalent to a notarization.

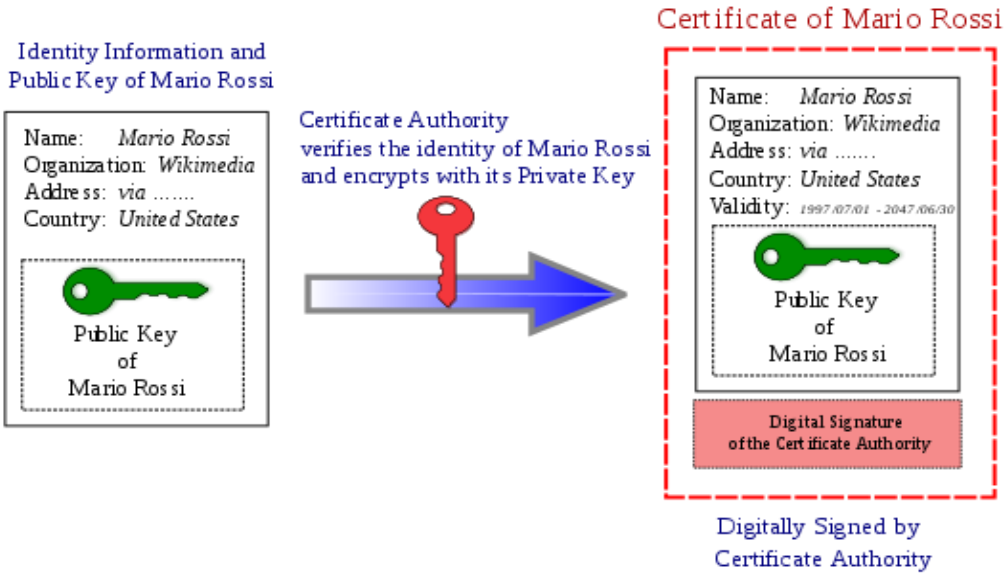
Benefits of Digital Signatures

- **Legal documents and contracts:** Digital signatures are legally binding. This makes them ideal for any legal document that requires a signature authenticated by one or more parties and guarantees that the record has not been altered.
- **Sales contracts:** Digital signing of contracts and sales contracts authenticates the identity of the seller and the buyer, and both parties can be sure that the signatures are legally binding and that the terms of the agreement have not been changed.
- **Financial Documents:** Finance departments digitally sign invoices so customers can trust that the payment request is from the right seller, not from a bad actor trying to trick the buyer into sending payments to a fraudulent account.
- **Health Data:** In the healthcare industry, privacy is paramount for both patient records and research data. Digital signatures ensure that this confidential information was not modified when it was transmitted between the consenting parties.

Drawbacks of Digital Signature

- **Dependency on technology:** Because digital signatures rely on technology, they are susceptible to crimes, including hacking. As a result, businesses that use digital signatures must make sure their systems are safe and have the most recent security patches and upgrades installed.
- **Complexity:** Setting up and using digital signatures can be challenging, especially for those who are unfamiliar with the technology. This may result in blunders and errors that reduce the system's efficacy. The process of issuing digital signatures to senior citizens can occasionally be challenging.
- **Limited acceptance:** Digital signatures take time to replace manual ones since technology is not widely available in India, a developing nation.

3.3 Digital Certificates:



Digital certificate is issued by a trusted third party which proves sender's identitytothe receiver and receiver's identity to the sender.A digital certificate is a certificate issued by a Certificate Authority (CA) to verify the identity of the certificate holder. Digital certificate is used to attach public key with a particular individual or an entity.

Digital certificate contains

- Name of certificate holder.
- Serial number which is used to uniquely identify a certificate, the individual or the entity identified by the certificate
- Expiration dates.
- Copy of certificate holder's public key(used for decrypting messages and digital signatures)
- Digital Signature of the certificate issuing authority.

Digital certificate is also sent with the digital signature and the message.

Advantages of Digital Certificate:

- **NETWORK SECURITY:** A complete, layered strategy is required by modern cybersecurity methods, wherein many solutions cooperate to offer the highest level of protection against malevolent actors. An essential component of this puzzle is digital certificates, which offer strong defence against manipulation and man-in-the-middle assaults.
- **VERIFICATION:** Digital certificates facilitate cybersecurity by restricting access to sensitive data, which makes authentication a crucial component of cybersecurity. Thus, there is a decreased chance that hostile actors will cause chaos. At many different endpoints, certificate-based authentication provides a dependable method of identity verification. Compared to other popular authentication methods like biometrics or one-time passwords, certificates are more flexible.
- **BUYER SUCCESS:** Astute consumers demand complete assurance that the websites they visit are reliable. Because digital certificates are supported by certificate authority that users' browsers trust, they offer a readily identifiable indicator of reliability.

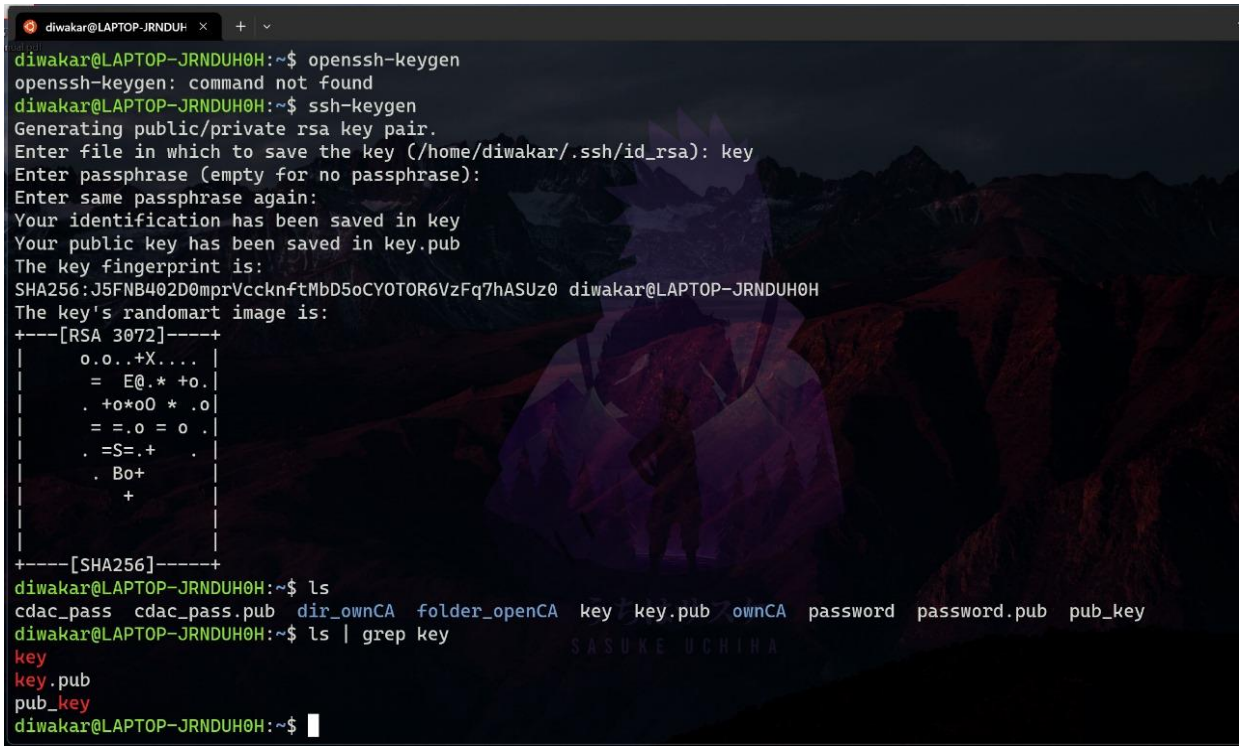
Disadvantages of Digital Certificate:

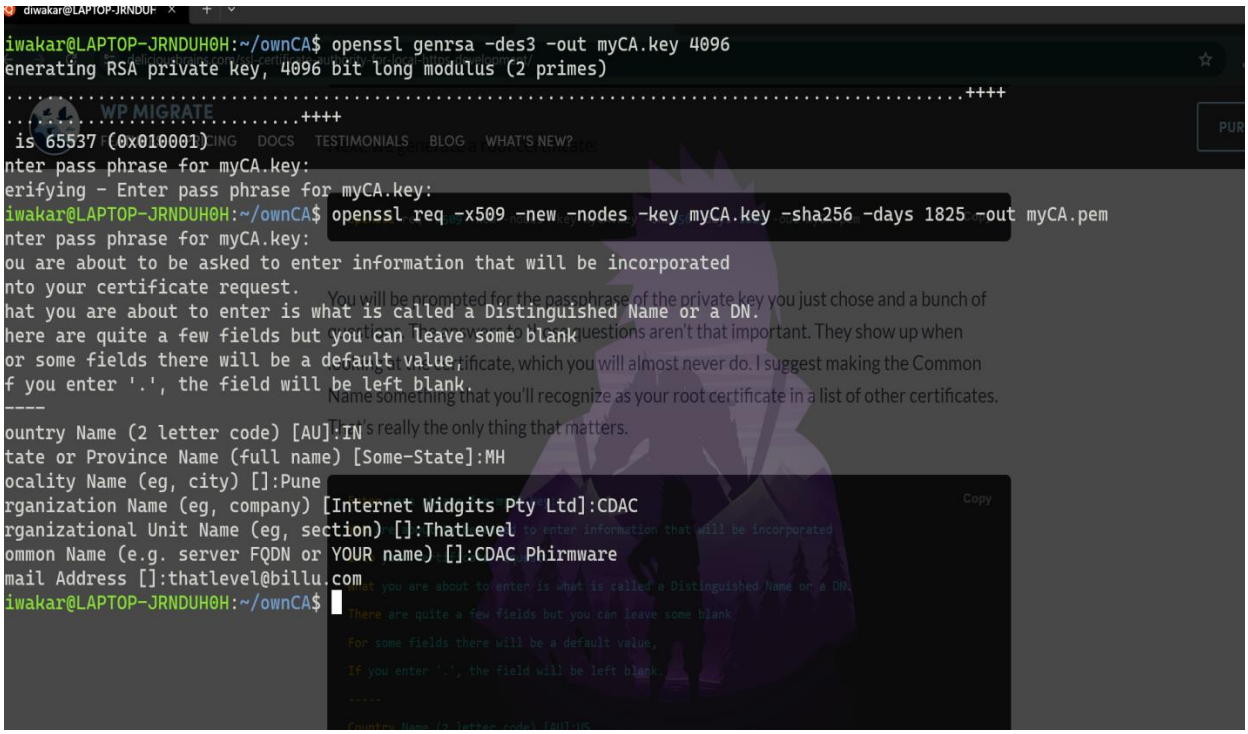
- **Phishing attacks:** To make their websites look authentic, attackers can fabricate bogus websites and obtain certificates. Users may be fooled into providing sensitive information, such as their login credentials, which the attacker may then take advantage of.
- **Weak encryption:** Older digital certificate systems may employ less secure encryption methods that are open to intrusions.
- **Misconfiguration:** In order for digital certificates to work, they need to be set up correctly. Websites and online interactions can be attacked due to incorrectly configured certificates.

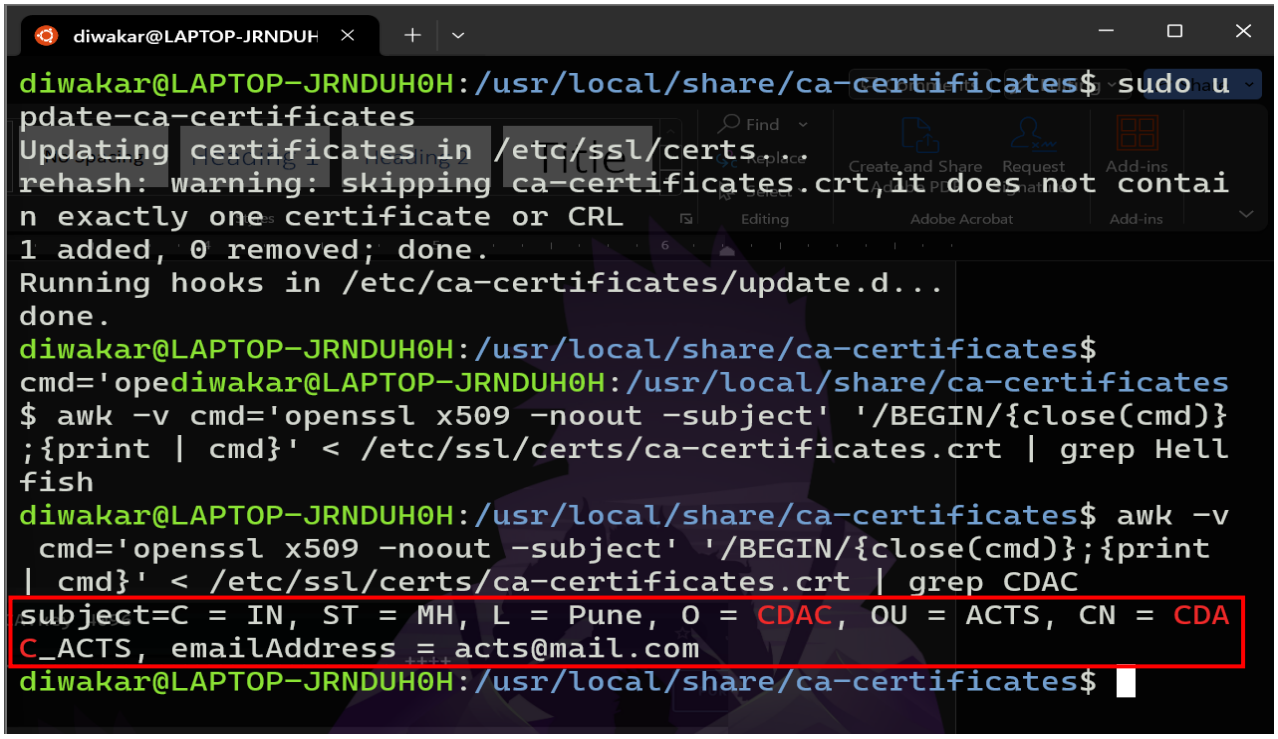
Digital certificate vs digital signature:

Digital signature is used to verify authenticity, integrity, non-repudiation,i.e. it is assuring that the message is sent by the known user and not modified, while digital certificate is used to verify the identity of the user, maybe sender or receiver. Thus, digital signature and certificate are different kind of things but both are used for security.

Most websites use digital certificate to enhance trust of their users.


diwakar@LAPTOP-JRNDUH0H:~$ openssh-keygen
openssh-keygen: command not found
diwakar@LAPTOP-JRNDUH0H:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/diwakar/.ssh/id_rsa): key
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in key
Your public key has been saved in key.pub
The key fingerprint is:
SHA256:J5FNB402D0mprVcckn ftMbD5oCYOTOR6VzFq7hASUz0 diwakar@LAPTOP-JRNDUH0H
The key's randomart image is:
+---[RSA 3072]----+
| o.o..+X.... |
| = E@.* +o.|
| . +o*oO * .o|
| = =.o = o .|
| . =S=.+   . |
| . Bo+ |
| + |
| |
| |
+----[SHA256]-----+
diwakar@LAPTOP-JRNDUH0H:~$ ls
cdac_pass cdac_pass.pub dir_ownCA folder_openCA key key.pub ownCA password password.pub pub_key
diwakar@LAPTOP-JRNDUH0H:~$ ls | grep key
key
key.pub
pub_key
diwakar@LAPTOP-JRNDUH0H:~$


iwakar@LAPTOP-JRNDUH0H:~/ownCA$ openssl genrsa -des3 -out myCA.key 4096
enerating RSA private key, 4096 bit long modulus (2 primes)
.....++++
.....++++
is 65537 (0x010001)
nter pass phrase for myCA.key:
erifying - Enter pass phrase for myCA.key:
iwakar@LAPTOP-JRNDUH0H:~/ownCA$ openssl req -x509 -new -nodes -key myCA.key -sha256 -days 1825 -out myCA.pem
nter pass phrase for myCA.key:
ou are about to be asked to enter information that will be incorporated
nto your certificate request.
hat you are about to enter is what is called a Distinguished Name or a DN.
here are quite a few fields but you can leave some blank
or some fields there will be a default value,
f you enter '.', the field will be left blank.
---
ountry Name (2 letter code) [AU]:IN
tate or Province Name (full name) [Some-State]:MH
ocality Name (eg, city) []:Pune
rganization Name (eg, company) [Internet Widgits Pty Ltd]:CDAC
rganizational Unit Name (eg, section) []:ThatLevel
ommon Name (e.g. server FQDN or YOUR name) []:CDAC Phirmware
mail Address []:thatlevel@billu.com
iwakar@LAPTOP-JRNDUH0H:~/ownCA$


diwakar@LAPTOP-JRNDUH
diwakar@LAPTOP-JRNDUH0H:/usr/local/share/ca-certificates$ sudo u
pdate-ca-certificates
Updating certificates in /etc/ssl/certs...
rehash: warning: skipping ca-certificates.crt,it does not contai
n exactly one certificate or CRL
1 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...
done.
diwakar@LAPTOP-JRNDUH0H:/usr/local/share/ca-certificates$
cmd='opediwakar@LAPTOP-JRNDUH0H:/usr/local/share/ca-certificates
$ awk -v cmd='openssl x509 -noout -subject' '/BEGIN/{close(cmd)}
;{print | cmd}' < /etc/ssl/certs/ca-certificates.crt | grep Hell
fish
diwakar@LAPTOP-JRNDUH0H:/usr/local/share/ca-certificates$ awk -v
cmd='openssl x509 -noout -subject' '/BEGIN/{close(cmd)};{print
| cmd}' < /etc/ssl/certs/ca-certificates.crt | grep CDAC
subject=C = IN, ST = MH, L = Pune, O = CDAC, OU = ACTS, CN = CDA
C_ACTS, emailAddress = acts@mail.com
diwakar@LAPTOP-JRNDUH0H:/usr/local/share/ca-certificates$

Chapter 3

BACKGROUND AND DEFINITIONS

In this section, we introduce the required background for both the boot process of a medium-scale embedded device and secure boot in particular.

Boot Process: Commonly, manufacturers of embedded devices divide the boot-up process.The purpose of each stage is to prepare the CPU and transfer code fragments from external to internal memory. Eventually, the boot-loader loads the operating system and starts the execution of the kernel. The boot process begins with the Boot ROM, provided by the manufacturer, which sets and initializes the peripherals. The Boot ROM prepares the system to execute the primary boot stage. The primary stage configures the system and prepares the memory for loading the secondary stage boot-loader. The secondary stage is the actual bootloader of the operating system.

Commonly, both the Boot ROM and the primary stage are tamper-proof, as both are hard-coded in the device firmware. However, manufacturers allow the modification of the second stage to provide flexibility and support for different bootloaders and operating systems. As a result, a secure boot process has to ensure the integrity of the kernel and application code before executing the secondary stage.

Security Challenge: The code in each stage can change the overall status of the system and often the next-stage software. As a result, we cannot trust a self-verified software, as it can be modified to provide a false verification status. Therefore, we need to verify in advance, and before we run each piece of software that we give control over the system.

Secure Boot: In a secure boot process, an inherently trusted component triggers the boot process, which is a tamper-proof component referred to as the Roots of Trust

(RoT). The Trusted Computing Group (TCG) defines RoT as a set of functions designed to be trusted by the operating system\

In embedded systems, RoT can be the Boot ROM which verifies the next-stage software and executes only authentic software. Each stage verifies the integrity of the next one leading to a Chain of Trust. For the verification, we can use a dedicated monitoring hardware co-processor. TCG has defined an international standard called the Trusted Platform Module (TPM), which defines the properties that those modules need to fulfill.

We note that different standards and vendors use various terminology to describe a secured boot process: Common terms include, for example, Secure boot, Trusted Boot and Verified boot. Different solutions have been defined and implemented in specific environments including personal computers, data centers, routers, and mobile phones. Especially, Secure Boot has been among the standard techniques to define a secured process to assure the integrity of each booting steps. In table I we compare the terminology for the existing techniques.

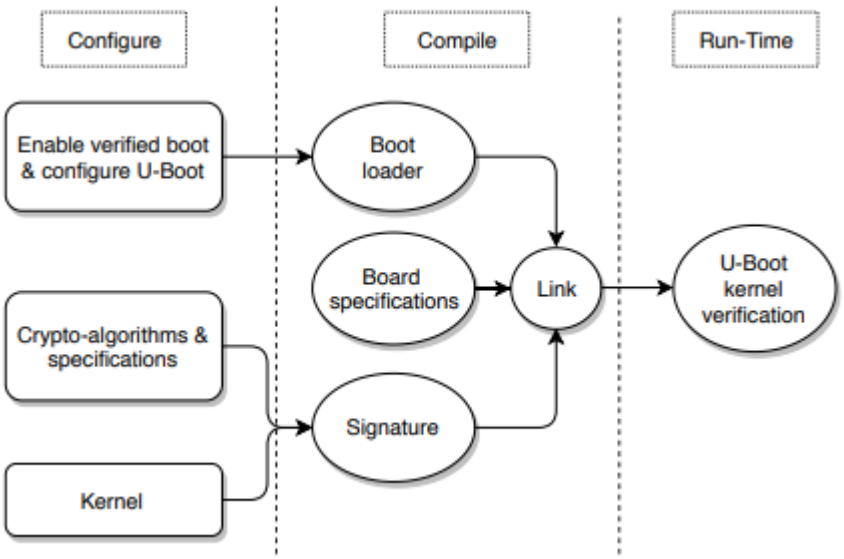
Chapter 4

SYSTEMS OVERVIEW

In this section, we present and discuss system designs to secure the boot process with an operating system such as embedded Linux:

A. Software-based Secure Boot with U-Boot

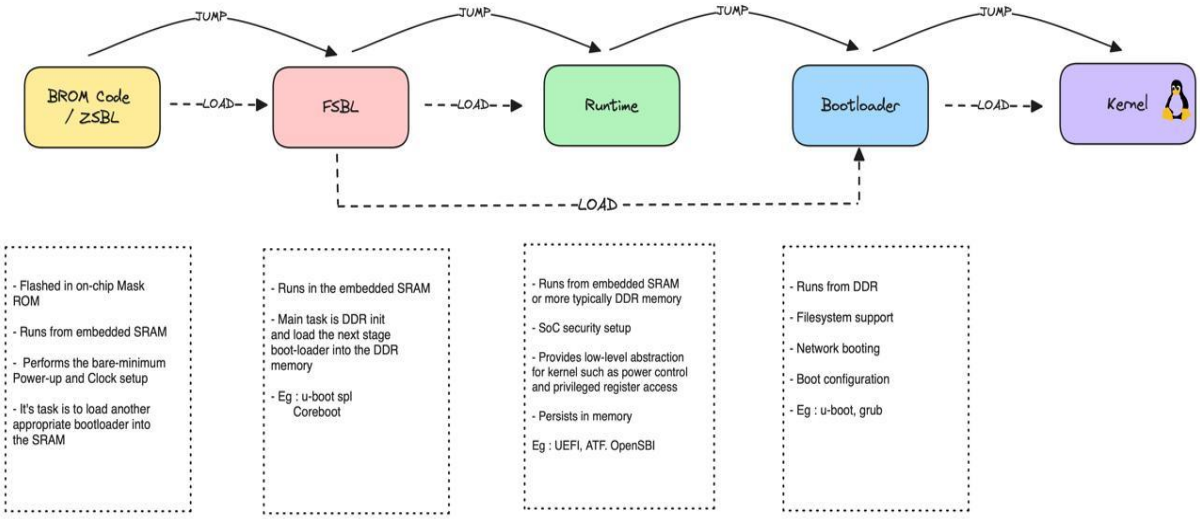
For the software-based method, we rely on the U-Boot bootloader to verify the integrity of the operating system. In our system design, we make the following assumptions. Firstly, the pre-boot environment of U-Boot has to be trusted,



From U-Boot configuration to deployment: First, we configure the U-Boot to include the verification module. Next, we link the object files to produce the secure version of U-Boot. Finally, we deploy the executable file on the platform. meaning that the security of the boot stages before U-Boot cannot be modified. Typically, the manufacturer embeds the first stage in the Boot ROM. Secondly, U-Boot has to be placed in read-only memory since there is no prior verification of the booting process. Lastly, this design requires read-only storage of the cryptographic hashes used to verify the integrity of the operating system. U-Boot divides the security process into three steps: Configure, Compile and Run-Time. The software-based secure boot process extends the boot process with an additional verification step.

As a result, the bootloader only boots the operating system once it has successfully verified the integrity of the operating system. In practice, U-Boot binds the kernel with the hardware information of the board. Thus, U-Boot verifies that the kernel is correct and it will run on the specific hardware configuration. To verify the integrity of the kernel efficiently, we need to resolve the digital block data of each image to a single value; a conventional method is to use cryptographic hash functions. Cryptographic hash functions map an arbitrarily long data to a small and fixed output, but they need to fulfill specific properties to be considered cryptographically secure [27]. U-Boot supports three cryptographic hash functions, namely MD5, SHA-1 & SHA-256 [27]. The hash functions have the following digest sizes: (1) MD5: 128-bit (2) SHA-1: 160-bit (3) SHA-256: 256-bit. Finally, the hash digest is being signed by the private key, and the bootloader (U-Boot) can verify the authenticity of the hash value by applying its public key. Various public key algorithms could verify the hash digest of the image. Popular public key cryptographic algorithms are RSA [27] and Elliptic Curve Cryptography (ECC) [27]. U-Boot currently supports only RSA, and the two supported key sizes are 2048-bit and 4096-bit. The key size is the critical factor of public key algorithms; bigger key sizes are more difficult to break. On the other hand, the larger the key size.

3.4 Simple Boot Flow:



1. Power-On: The processor initializes and begins execution from a predefined memory location, often pointing to ROM or a specific address in SRAM.

2. Zeroth Stage Boot Loader (ROM Code): This initial boot loader, residing in ROM, sets up basic hardware components including the CPU registers, memory controller, and interrupt controller. It may also initialize small amounts of SRAM for temporary storage.

3. First Stage Boot Loader (Boot ROM): This loader, typically stored in non-volatile memory like ROM or flash, initializes more complex hardware components like storage devices. It may use SRAM for storing temporary data during the boot process.

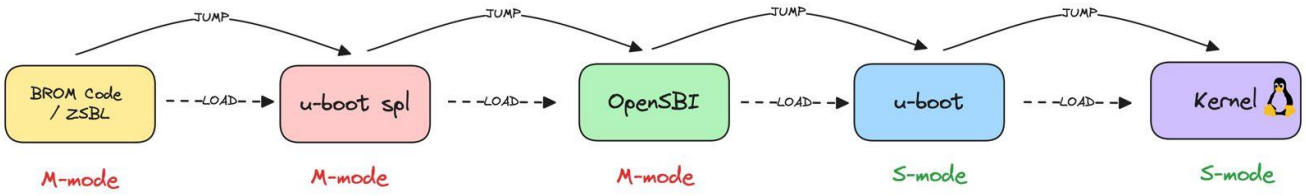
4.Second Stage Boot Loader (Boot Loader): After loading from the boot device, this loader resides in memory, often loaded into DRAM. It further initializes system components, including DRAM itself if it wasn't already initialized, and loads the operating system kernel into memory.

5.Runtime: Once the operating system kernel is loaded into DRAM and begins execution, it takes control of the system. The kernel initializes device drivers, sets up the memory management system (including managing DRAM), and starts system services. User processes are then launched, providing a fully functional operating system environment.

Throughout the boot process, SRAM is typically used for fast and temporary storage,

such as storing critical data structures or code during the bootstrapping process. DRAM, being larger but slower than SRAM, is used for storing the operating system kernel, device drivers, and user processes during system runtime.

3.5 RISC V Boot Flow:



1.Zeroth Stage Boot Loader (ZSBL):

The ZSBL initializes essential hardware components, including SRAM if available. SRAM is often used for storing critical system firmware and configuration data due to its fast access times and non-volatile nature.

If the system has SRAM, the ZSBL may copy its initial code and data from ROM to SRAM for faster access during subsequent stages.

2.u-boot SPL (Secondary Program Loader):

The SPL continues the initialization process by configuring more complex hardware components such as DRAM controllers.

DRAM initialization is crucial at this stage because it's the main memory used for storing bootloader code, kernel images, and other runtime data.

The SPL typically performs memory training and configuration routines to set up the DRAM for reliable operation at the highest possible speed.

3.OpenSBI (Open Source Boot Loader):

OpenSBI, after being loaded into DRAM by the SPL, utilizes DRAM for its data structures and runtime operations.

It might perform additional DRAM initialization or verification steps as part of its platform setup process to ensure proper memory operation before handing control over to the secondary bootloader or operating system kernel.

4.Secondary Bootloader (Optional):

If there's a secondary bootloader stage, it also relies on DRAM for loading and executing its code.

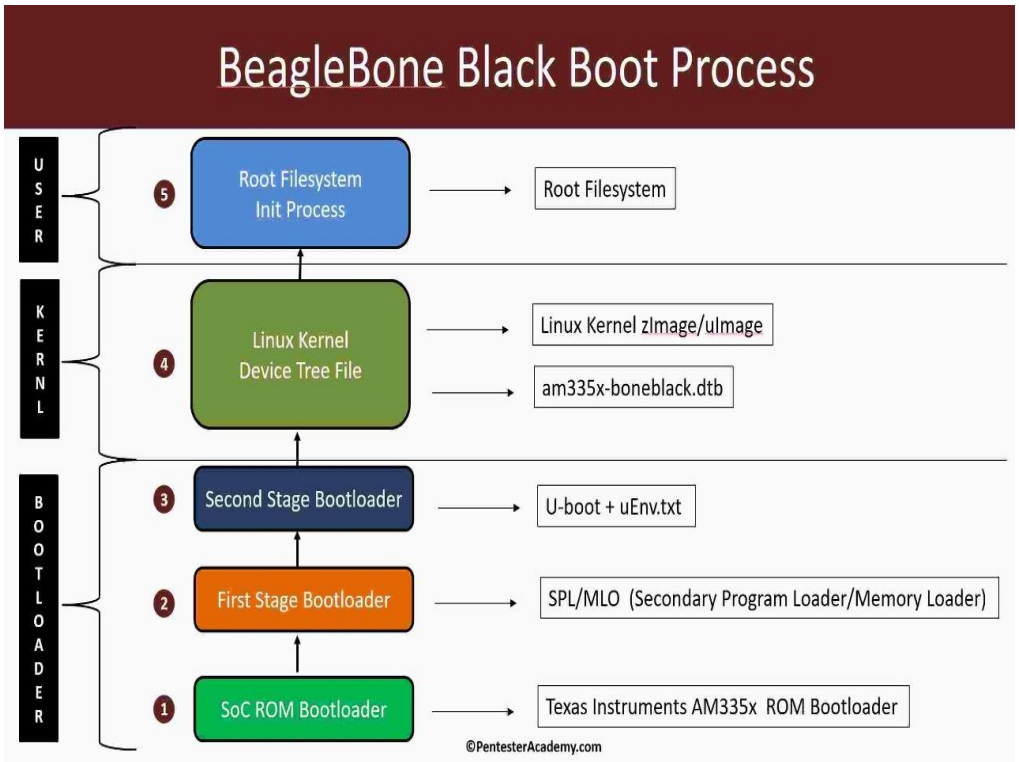
This stage might involve loading additional configuration data or performing platform-specific tasks that require access to DRAM.

5.Operating System Kernel:

Once the operating system kernel takes control, it heavily relies on DRAM for its execution and data storage needs.

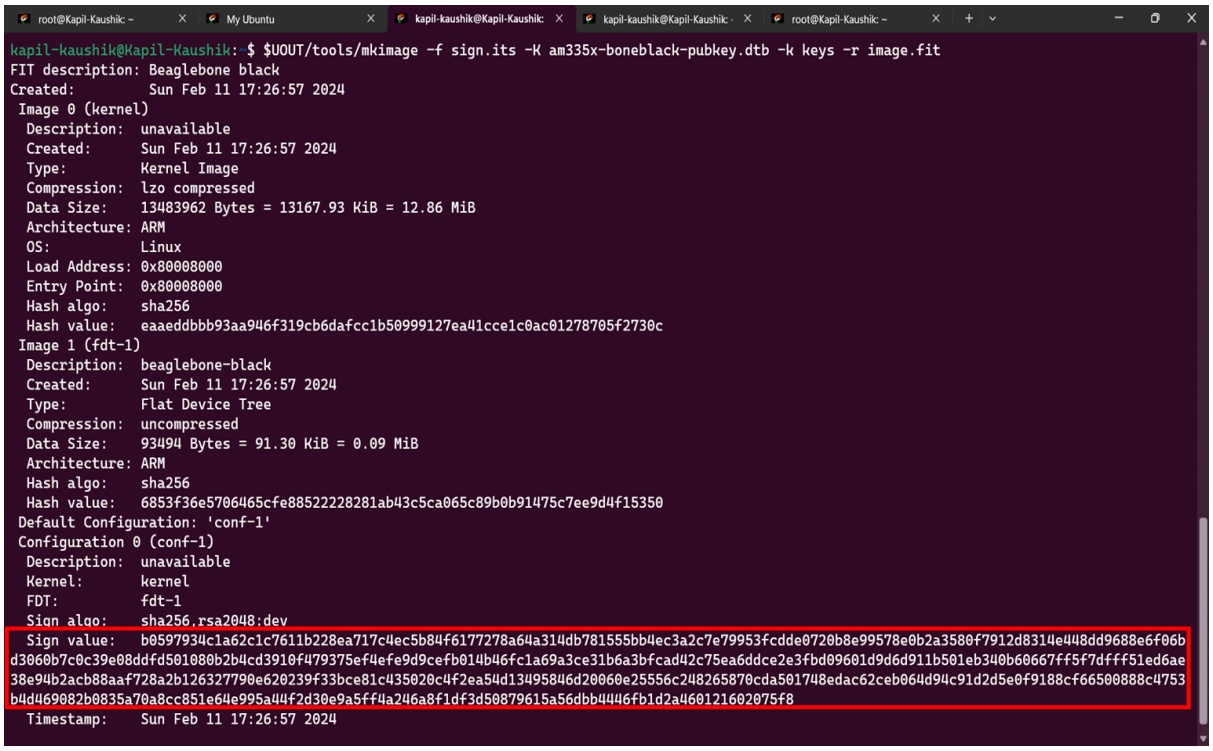
DRAM serves as the main memory for running user-space processes, storing kernel data structures, and caching disk I/O operations, among other tasks.

Throughout the boot flow, both SRAM and DRAM play critical roles in storing and accessing bootloader code, runtime data, and configuration information. SRAM typically provides fast, low-latency access for critical firmware and initialization routines, while DRAM offers larger storage capacity and serves as the primary memory for running software on the system.



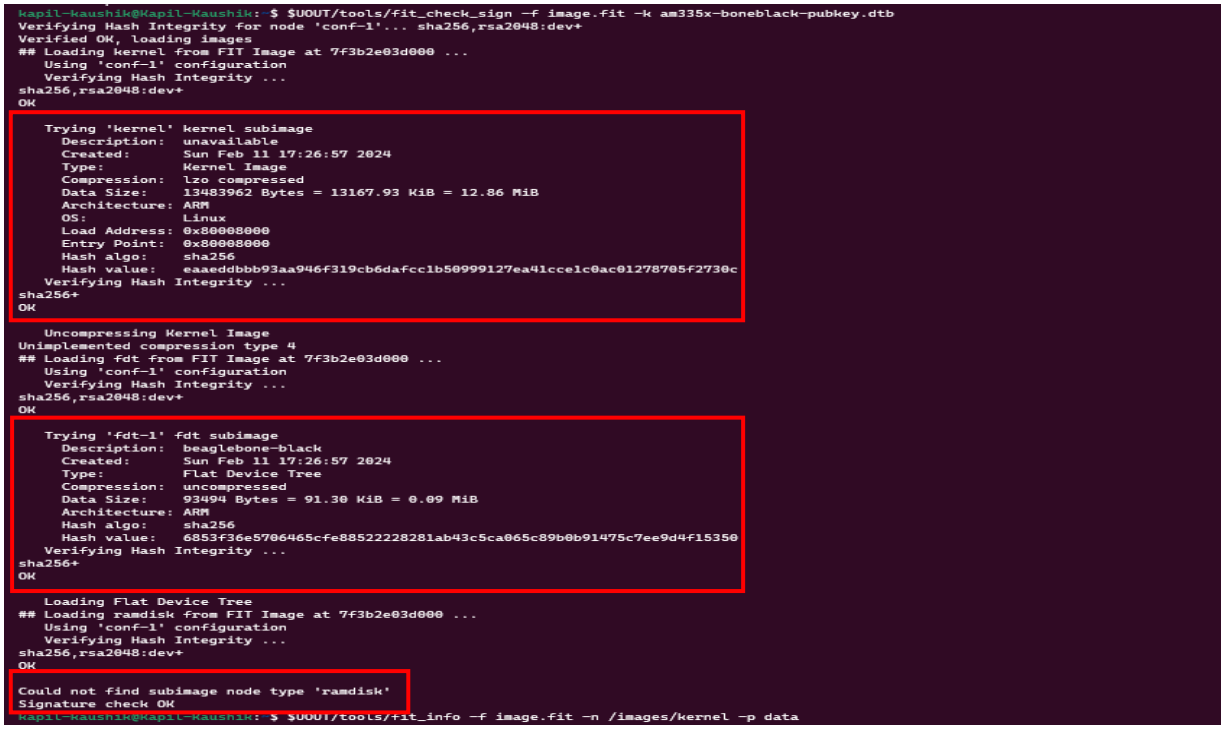
Implementation

3.6Signing Kernel:



```
kapil-kaushik@Kapil-Kaushik:~$ $UOUT/tools/mkimage -f sign.its -K am335x-boneblack-pubkey.dtb -k keys -r image.fit
FIT description: Beaglebone black
Created:         Sun Feb 11 17:26:57 2024
 Image 0 (kernel)
  Description:  unavailable
  Created:      Sun Feb 11 17:26:57 2024
  Type:         Kernel Image
  Compression:  lzo compressed
  Data Size:    13483962 Bytes = 13167.93 KiB = 12.86 MiB
  Architecture: ARM
  OS:           Linux
  Load Address: 0x80008000
  Entry Point:  0x80008000
  Hash algo:    sha256
  Hash value:   eaaeddbbb93aa946f319cb6dafcc1b50999127ea41cce1c0ac01278705f2730c
 Image 1 (fdt-1)
  Description:  beaglebone-black
  Created:      Sun Feb 11 17:26:57 2024
  Type:         Flat Device Tree
  Compression:  uncompressed
  Data Size:    93494 Bytes = 91.30 KiB = 0.09 MiB
  Architecture: ARM
  Hash algo:    sha256
  Hash value:   6853f36e5706465cfe88522228281ab43c5ca065c89b0b91475c7ee9d4f15350
 Default Configuration: 'conf-1'
 Configuration 0 (conf-1)
  Description:  unavailable
  Kernel:       kernel
  FDT:          fdt-1
  Sign algo:    sha256,rsa2048:dev
  Sign value:   b0597934c1a62c1c7611b228ea717c4ec5b84f6177278a64a314db781555bb4ec3a2c7e79953fcdde0720b8e99578e0b2a3580f7912d8314e448dd9688e6f06b
d3060b7c0c39e08ddfd501080b2b4cd3910f479375ef4efe9d9cefb014b46fc1a69a3ce31b6a3bfcad42c75ea6ddce2e3fbd09601d9d6d911b501eb340b60667ff5f7dfff51ed6ae
38e94b2acb88aaf728a2b126327790e620239f33bce81c435020c4f2ea54d134958046d20060e25556c248265870cda501748edac62ceb064d94c91d2d5e0f9188cf66500888c4753
b4d469082b0835a70a8cc851e64e995a44f2d30e9a5ff4a246a8f1df3d50879615a56dbb4446fb1d2a46012160207 5f8
  Timestamp:    Sun Feb 11 17:26:57 2024
```

3.6.1Verifying Kernel with Public Key & Signature:



```
kapil-kaushik@Kapil-Kaushik: $ $UOUT/tools/fit_check_sign -f image.fit -k am335x-boneblack-pubkey.dtb
Verifying Hash Integrity for node 'conf-1'... sha256,rsa2048:dev+
Verified OK, loading images
## Loading kernel from FIT Image at 7f3b2e03d000 ...
   Using 'conf-1' configuration
   Verifying Hash Integrity ...
sha256,rsa2048:dev+
OK
   Trying 'kernel' kernel subimage
     Description:  unavailable
     Created:      Sun Feb 11 17:26:57 2024
     Type:         Kernel Image
     Compression:  lzo compressed
     Data Size:    13483962 Bytes = 13167.93 KiB = 12.86 MiB
     Architecture: ARM
     OS:           Linux
     Load Address: 0x80008000
     Entry Point:  0x80008000
     Hash algo:    sha256
     Hash value:   eaaeddbbb93aa946f319cb6dafcc1b50999127ea41cce1c0ac01278705f2730c
   Verifying Hash Integrity ...
sha256+
OK
   Uncompressing Kernel Image
Unimplemented compression type 4
## Loading fdt from FIT Image at 7f3b2e03d000 ...
   Using 'conf-1' configuration
   Verifying Hash Integrity ...
sha256,rsa2048:dev+
OK
   Trying 'fdt-1' fdt subimage
     Description:  beaglebone-black
     Created:      Sun Feb 11 17:26:57 2024
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Size:    93494 Bytes = 91.30 KiB = 0.09 MiB
     Architecture: ARM
     Hash algo:    sha256
     Hash value:   6853f36e5706465cfe88522228281ab43c5ca065c89b0b91475c7ee9d4f15350
   Verifying Hash Integrity ...
sha256+
OK
   Loading Flat Device Tree
## Loading ramdisk from FIT Image at 7f3b2e03d000 ...
   Using 'conf-1' configuration
   Verifying Hash Integrity ...
sha256,rsa2048:dev+
OK
Could not find subimage node type 'ramdisk'
Signature check OK
kapil-kaushik@Kapil-Kaushik: $ $UOUT/tools/fit_info -f image.fit -n /images/kernel -p data
```

3.6.2Beagle bone Board Booting:

Chapter 6

Conclusion

6.1 Conclusion

Here at the end we concluded that secure boot is a critical feature in embedded systems that ensures that only trusted software can run on the system. By verifying the digital signature of the software before it is loaded, secure boot protects against malware and other attacks.

6.2 Future Enhancement –

For future works, we plan to evaluate the impact of different system configurations and kernel configurations on the performance of secure boot. Moreover, we will focus on the design, implementation, and evaluation of secure boot on smaller and more constrained devices (e.g., ARM A8).

Chapter 7

References

[1] https://docs.u-boot.org/en/latest/usage/fit/beaglebone_vboot.html

[2] https://www.beagleboard.org/boards/beaglebone-black

[3] https://github.com/torvalds/linux

[4] https://github.com/u-boot/u-boot

[5]https://bootlin.com/doc/training/embedded-linux-bbb/embedded-linux-bbb-labs.pdf