

Deggendorf Institute of Technology
Faculty of Computer Science
Master of Applied Computer Science

Embedded Security

Dr. Stefanie Merz

Submitted by
Mahmoud Gahelrasoul 00775836
Yasin Elmezayen 22306157

Table of Content

1. Introduction	4
2. Software Application	5
2.1 Generating Keys and Certificates	5
2.1.1 Public and Private Key Pair	5
2.1.2 DER Extension of Public Key	6
2.1.3 Self-signed Certificate	6
2.1.4 Combined Certificate	7
2.2. Adding Certificate to Kernel	7
2.2.1 Transfer of Certificate	7
2.2.2 Kernel Configuration	7
2.3 Kernel Building	9
2.3.1 Kernel Compiling	9
2.3.2 Kernel Install	9
2.3.3 Updating the Bootloader	10
2.3.4 System Rebooting	10
2.3.5 Kernel Test	10
2.4. Implementing and Signing Kernel Modules	11
2.4.1 Creating a Kernel Module	11
2.4.2 Compiling the Kernel Module	11
2.4.3 Signing the Module	12
2.4.4 Loading and Unloading the Signed Kernel Module	13
2.4.5 Comparison with Unsigned Modules	15
3. Hardware Implementation	16
3.1 Approaches	16
3.1.1 Native Compilation	16
3.1.2 Virtual Machines or Emulation	16
3.1.3 Remote Compilation Servers	16
3.1.4 Cross Compilation	16
3.2 Building a Signed Kernel and Signed Module with the Same Keys and Certificate on a Raspberry Pi Using Cross-Compilation	17
3.2.1 Native Compilation	17
3.2.2 Cross-Compilation and Transition	18
3.2.3 Installing the Kernel Image	19
3.2.4 Installing Device Tree Blobs	19
3.2.5 Installing Kernel Modules	20
3.2.6 Updating Boot Configuration	20
3.2.7 Reboot the Raspberry Pi	20
3.2.8 Verifying the Installation	20
3.3 Compiling and signing the Kernel Module	21
3.4 Performing Loading and Unloading Checks for the Signed Module and unsigned one	21
4. Challenges and Future	22
5. Conclusion	23
5. References	24

Table of Figure

Figure 1: Public Key	5
Figure 2: Matching Certificate with the Public Key	6
Figure 3: Certificate and Private Key	7
Figure 4: Kernel Configuration	8
Figure 5: Kernel Compiling	9
Figure 6: Kernel Install	10
Figure 7: Kernel Check	10
Figure 8: Source Code	11
Figure 9: Makefile	11
Figure 10: Module Signing	12
Figure 11: Kernal Log Login	13
Figure 12: Kernel Logs Checks	14
Figure 13: Whole Kernel Log	15

Abstract:

This paper provides a thorough elaboration on securely implementing kernel modules. It covers the generation and management of cryptographic keys, configuring kernel settings, and testing signed versus unsigned modules to improve understanding of secure kernel practices. The project goal is a software program in Linux operating system should be designed to connect to Raspberry Pi, which accepts input signals from a linked hardware controller and adjusts a player character using the input signals.

1. Introduction:

The kernel module signing facility uses cryptographic techniques to sign modules during installation and verify signatures upon loading, enhancing kernel security by preventing the introduction of unsigned or invalid modules. This verification is performed by the kernel itself, eliminating the need for trusted user space components. Kernel modules extend Linux kernel functionality dynamically without requiring a reboot, making their integrity and authenticity crucial. Loadable kernel modules (LKMs) allow for a modular approach to kernel development, enabling dynamic insertion and removal of features like device drivers and system calls while the system is operational. This modular capability is essential for maintaining a minimal base kernel image in embedded Linux systems, with additional features loaded as needed, such as drivers for hot-pluggable devices. LKMs can be statically linked to the kernel image or dynamically loadable. Kernel modules can be built either statically linked to the kernel image or dynamically loadable. Static LKMs are integrated into the final kernel image during the build process, increasing the kernel's size and remaining in memory permanently since they cannot be unloaded. Dynamic LKMs, however, are compiled and linked separately, resulting in .ko files that can be loaded and unloaded from the kernel using user space programs like 'insmod', 'modprobe', and 'rmmod'. This project aims to create a kernel, dynamically loadable module, sign it with a self-generated private key, and have the kernel verify it using the corresponding public key. This paper outlines a detailed procedure for securely signing and loading kernel modules, thus enhancing kernel security. Implementing a signed kernel and signed modules on embedded hardware, such as a Raspberry Pi, involves integrating cryptographic signatures to ensure the authenticity and integrity of these software components. This process entails compiling the kernel and modules with embedded digital signatures using a private key, while the corresponding public key is utilized during system boot and module loading to verify their legitimacy. By performing these security checks, which include verifying cryptographic signatures against trusted certificates, the system can prevent the execution of tampered or unauthorized code, thereby enhancing the overall security and reliability of the embedded system. This methodology is crucial for maintaining the integrity of embedded devices in environments where security is paramount. A further discussion on this matter is being held later this paperwork.

2. Software Application:

2.1 Generating Keys and Certificates:

The securing kernel modules is the first step to generate the cryptographic keys and certificates needed for signing the modules. The securing kernel modules consists of creating a pair of public and private keys, a self-signed certificate, and a combined certificate that will be used by the kernel.

2.1.1 Public and Private Key Pair:

To generate the public and private key pair, the OpenSSL tool is used. The following command creates a 2048-bit RSA key pair, which is stored in two files: '**private_key.pem**' and '**public_key.pem**'. The private key is essential for signing the kernel modules, while the public key will be included in the certificate.

```
# Generating private key and public key pair command:
openssl req -new -newkey rsa:2048 -days 3650 -nodes -x509 -keyout private_key.pem -out
public_key.pem
```

- The '**-newkey rsa:2048**' option specifies the creation of a new RSA key with a size of 2048 bits.
- The '**-days 3650**' option sets the certificate validity period to 10 years.
- The '**-nodes**' option indicates that the private key should not be encrypted.
- The '**-x509**' option is used to create a self-signed certificate.

➤ Public Key:

Figure 1 show the first step to create public key

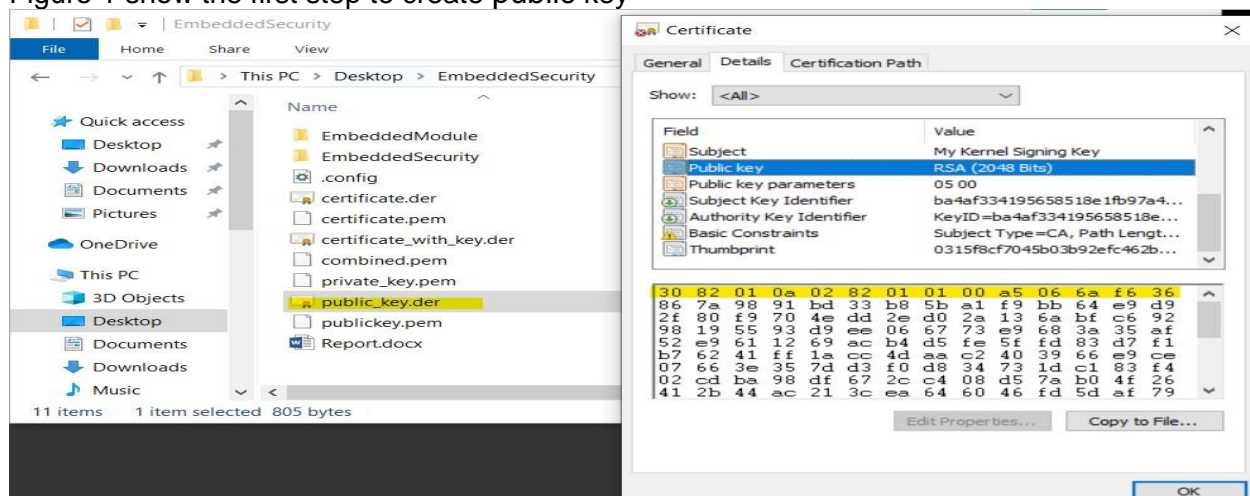


Figure 1: Public Key

2.1.2 DER Extension of Public Key:

Usually, the DER format is necessary to make the public key. The DER format is a binary form of the PEM format that can be used by various applications.

- The '**-outform der**' option specifies the output format as DER.
- This step is optional and is included for completeness.

```
# Generating a DER extension of the public key Command:  
openssl x509 -outform der -in public_key.pem -out public_key.der
```

2.1.3 Self-signed Certificate:

A self-signed certificate is generated using the private key. This certificate is necessary for the kernel to trust the signature on the modules.

- The '**-new**' option indicates a new certificate signing request.
- The '**-x509**' option specifies that a self-signed certificate should be created.
- The '**-key private_key.pem**' option points to the private key to be used for signing.
- The '**-subj**' option provides the subject information for the certificate.

```
# Generating a self-signed certificate  
openssl req -new -x509 -key private_key.pem -out certificate.pem -days 365 -subj  
"/CN=Your CN (hello)"
```

Figure 2 shows the Certificate which matches the Public Key.

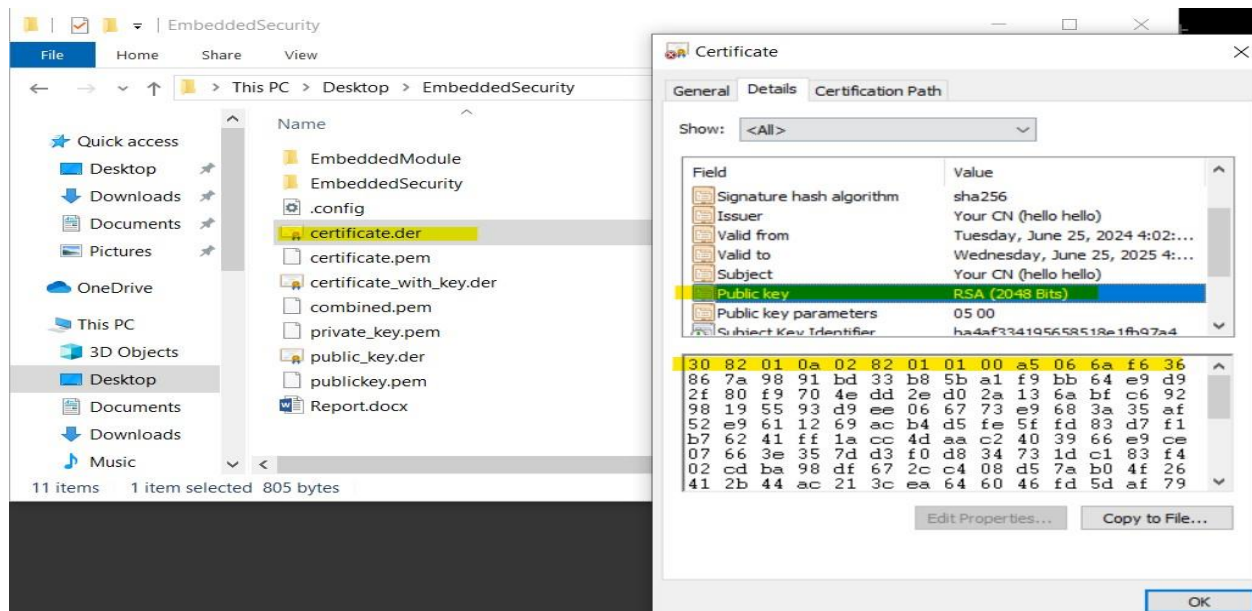


Figure 2: Matching Certificate with the Public Key

2.1.4 Combined Certificate:

The combined certificate is created by concatenating the private key and the self-signed certificate into a single file. This combined certificate is required for the kernel configuration.

```
#Generating a certificate of PEM extension that includes the private key as well:  
cat private_key.pem certificate.pem > combined.pem
```

- This command uses the 'cat' utility to concatenate with '**private_key.pem**' and '**certificate.pem**' into '**combined.pem**'.
- The combined certificate will be used to sign the kernel modules and should be placed in the appropriate kernel directory.

Figure 3 shows the Certificate and private key in one file (Combined certificate).

```
-----BEGIN PRIVATE KEY-----  
MTEVQGBADANBqkqhk1G9W0BAQEFAAOCBCKCwggS3AgeAAoIBAQC1emr2NoZ6m3G9  
v7hbof07Z0N2L4DSEZ7dLTAQE2g/2pKY3VWt2e4G23Ppado3r1LpYR3PFLTV/1/9  
g9fTe23B/XfMTArCQD1m6C4H2J41f6Pw2DRzHCgD9ALNupJf2YzECNV65E8mQ5TE  
CE86mRqV4dF31HqURKqQd323vQ1M3VMT1K5YQD1J27e2210J711fE91hPH2TE  
SLAQcWAVZ637sgdVQZEW845ACT05Wz2pOWxTbFHue41To//2N9E3fggkNj15FTCT  
e0AYF1F120W1eR1K1W5tH13EDnD88HfGEL16u17LcR62GEBR0/12F2L8K5S5Y2  
Y103DaxZAGmBAACcggADE3usf39B+4R581F8dUHCDOFF6admlUpTa95Yp7C01r1  
k1bFXt1OGG31Gt21Hx5dm34B592F5NW29APTUTNpysU7e/kKc3GxwggBwT0EgH3  
3AK03B13YdXtUbc3Jme5fC0mX31F552H1S0F7KVB87X2VUv4dw3UMXPR7hV8V633  
IKxpduQ1ENVO/M5XYAe3H1AFUB9x18UDJDrX3HvE2Kn2nG6mT1YcWghgq62qGrh5  
E2S21d0FPuST6YdP5NxxZef9P7Z2E5XenC51Q45F20u1E8mdC01TRybH1/X595  
45m07Jpu+ZQ/KXTY0FNVUE7hV5Vxgt8WabVL4UPYwQKBgQDGL32t3CRmoJ05BYKF  
7d19VU9CwZ195f/2112N7Kc3fY193G8T7tdNFHhHng7Dc4YamqVbVORnDudr0h+  
30SC13JfMFX//52F4n24M3XQc1Zw3558YpVrT55FJRT32r0B8pYp0rVUJ3GE01YTF1  
KNCAMK0Hf73154L03fC2R2QK8BQdVfKCC0K7MMBjH8911F93LHK838BKAY4XV  
4EDYMS9VdNdyB8BgAspdGaICOPH5HC2Y55qY98559XFPX13AX80A2PAnXgmUNR/X  
emw0S3RdY10G6QCYGNHAFrdGMFNQC403A3E5U1VXKXFPVROAXEdYpm3LNPNC80  
p3Y6tPwawBggqcnUeSg1X7XwmG+mc3JCCY3+wlF0uE9Trf9G/MEFLX3B1WU/Lu  
Fu11G+11hVb63Qe2t35AWdSA+C+V1Y951FCdp3/UTJ22YCF5BDNL05G29ekktE51z  
8f11WQXQb4uVbBn1EC07A//8EudA/12pYR3XMPJg7ZnCl2739UA71qBA0G8AJL2  
EOxd6BtUTC1ay11Y3W6+D5DF2H55PEmdTpgAK2uF7//1U/YKUG/HTWrr4GyLCIFLP8  
fDREJBCvPdeE3J8JW5/CV42h0YGVHrYsacmbd3fZLdndt9EQ0d8ay1xduRnMB5  
SHR2mDHYyKAFNsV1Lh1J64K55s+gnMR76zKE3eLVaOGAATYCTJ3t3b00AzYw+k8X+  
r1R224J2ACrPADAHGqE2KfqrWtYfP5hLKvVnR1W3CE1F2PA0B2Y82676/FR4W63  
FRKJ1BqgtFw2WIDB8WtEwFLSKUZ81N1h3JVP2LVZhwE5nm0DJ9McFwYd40/+WbG  
s1d+ynxYpJ33CfF3JxmgNBO+  
-----END PRIVATE KEY-----  
-----BEGIN CERTIFICATE-----  
MIIDITCCAmGAWIeAgUwEoFLOo01262jfnH1wB71mh1J10wDQY3ko2IhvcNAQEL  
BQAwIDeMBWGAUEAwVW91c1c1BDT1A0aGV5bG8gaGV5bG8pMmB4XDTIOMDYvNTE0  
MDI0N0E0XDTIOMDYvNTE0MDI0N0E0EAMBAQIUAwVW91c1c1BDT1A0aGV5bG8ga  
AGV5bG8pMmB4XDTI0ANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBcQKCAQEAQpQ2q9Jagap1R  
fT2w6Rf12F2L8K5S5Y2Y103DaxZAGmBAACcggADE3usf39B+4R581F8dUHCDOFF6admlUpTa95Yp7C01r1  
k1bFXt1OGG31Gt21Hx5dm34B592F5NW29APTUTNpysU7e/kKc3GxwggBwT0EgH3  
3AK03B13YdXtUbc3Jme5fC0mX31F552H1S0F7KVB87X2VUv4dw3UMXPR7hV8V633  
IKxpduQ1ENVO/M5XYAe3H1AFUB9x18UDJDrX3HvE2Kn2nG6mT1YcWghgq62qGrh5  
E2S21d0FPuST6YdP5NxxZef9P7Z2E5XenC51Q45F20u1E8mdC01TRybH1/X595  
45m07Jpu+ZQ/KXTY0FNVUE7hV5Vxgt8WabVL4UPYwQKBgQDGL32t3CRmoJ05BYKF  
7d19VU9CwZ195f/2112N7Kc3fY193G8T7tdNFHhHng7Dc4YamqVbVORnDudr0h+  
30SC13JfMFX//52F4n24M3XQc1Zw3558YpVrT55FJRT32r0B8pYp0rVUJ3GE01YTF1  
KNCAMK0Hf73154L03fC2R2QK8BQdVfKCC0K7MMBjH8911F93LHK838BKAY4XV  
4EDYMS9VdNdyB8BgAspdGaICOPH5HC2Y55qY98559XFPX13AX80A2PAnXgmUNR/X  
emw0S3RdY10G6QCYGNHAFrdGMFNQC403A3E5U1VXKXFPVROAXEdYpm3LNPNC80  
p3Y6tPwawBggqcnUeSg1X7XwmG+mc3JCCY3+wlF0uE9Trf9G/MEFLX3B1WU/Lu  
Fu11G+11hVb63Qe2t35AWdSA+C+V1Y951FCdp3/UTJ22YCF5BDNL05G29ekktE51z  
8f11WQXQb4uVbBn1EC07A//8EudA/12pYR3XMPJg7ZnCl2739UA71qBA0G8AJL2  
EOxd6BtUTC1ay11Y3W6+D5DF2H55PEmdTpgAK2uF7//1U/YKUG/HTWrr4GyLCIFLP8  
fDREJBCvPdeE3J8JW5/CV42h0YGVHrYsacmbd3fZLdndt9EQ0d8ay1xduRnMB5  
SHR2mDHYyKAFNsV1Lh1J64K55s+gnMR76zKE3eLVaOGAATYCTJ3t3b00AzYw+k8X+  
r1R224J2ACrPADAHGqE2KfqrWtYfP5hLKvVnR1W3CE1F2PA0B2Y82676/FR4W63  
FRKJ1BqgtFw2WIDB8WtEwFLSKUZ81N1h3JVP2LVZhwE5nm0DJ9McFwYd40/+WbG  
s1d+ynxYpJ33CfF3JxmgNBO+  
-----END CERTIFICATE-----  
(base)  
newname@NoName MINGW64 ~/Desktop/EmbeddedSecurity
```

Figure 3: Certificate and Private Key

2.2. Adding Certificate to Kernel:

Incorporating the newly generated certificate into the kernel is crucial for enabling the secure loading of signed modules. This step involves several sub-steps, including transferring the combined certificate to the appropriate directory within the kernel source tree, configuring the kernel to recognize and use this certificate, and rebuilding the kernel to include these changes.

2.2.1 Transfer of Certificate:

The combined certificate ('**combined.pem**') contains both the private key and the self-signed certificate, should be copied into the kernel source tree. This is typically done by placing the certificate into the '**certs**' directory of the kernel source:

2.2.2 Kernel Configuration:

```
#Cobieng the Combined Certificate into Kernel Source  
Directory:  
cp combined.pem /usr/src/linux-6.9.9/certs/
```


Next, the kernel must be configured to utilize this certificate for module signing. This involves enabling module signature verification and specifying the certificate to be used. This can be done by using the 'make menuconfig' utility:

```
cd /usr/src/linux-6.9.9/make menuconfig
```

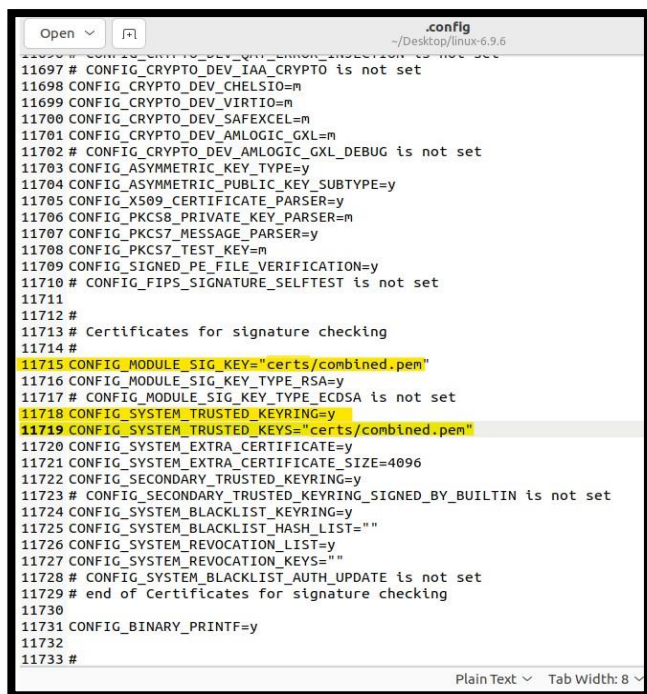
By Navigating through:

- **"Enable Loadable Module Support" -> "Module Signature Verification"**
- Set **"Require modules to be validly signed"** to 'Yes'
- Specify the path to the combined certificate

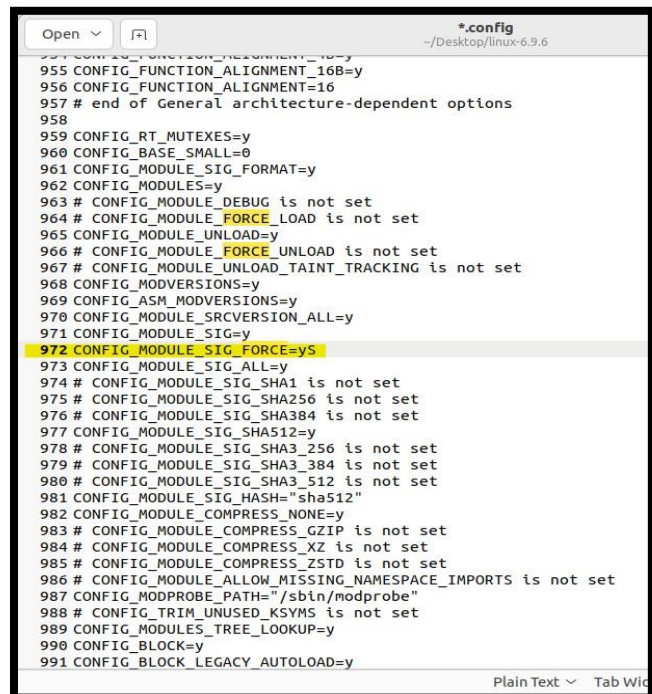
OR by editing the kernel configuration '**.config**' file directly in the kernel source directory which we did:

```
CONFIG_MODULE_SIG=y
CONFIG_MODULE_SIG_FORCE=y
CONFIG_MODULE_SIG_KEY="certs/combined.pem"
```

- **'CONFIG_MODULE_SIG=y'**: Enables module signature verification.
- **'CONFIG_MODULE_SIG_FORCE=y'**: Ensures that only signed modules will be loaded.
- **'CONFIG_MODULE_SIG_KEY="certs/combined.pem"'**: Specifies the path to the signing key.



```
Open  [icon] .config ~/Desktop/linux-6.9.6
11697 # CONFIG_CRYPTO_DEV_IAA_CRYPTO is not set
11698 CONFIG_CRYPTO_DEV_CHELSIO=m
11699 CONFIG_CRYPTO_DEV_VIRTIO=m
11700 CONFIG_CRYPTO_DEV_SAFEXCEL=m
11701 CONFIG_CRYPTO_DEV_AMLOGIC_GXL=m
11702 # CONFIG_CRYPTO_DEV_AMLOGIC_GXL_DEBUG is not set
11703 CONFIG_CRYPTO_DEV_ARM_Crypto=m
11704 CONFIG_CRYPTO_DEV_ARM_Crypto=m
11705 CONFIG_CRYPTO_DEV_ARM_Crypto=m
11706 CONFIG_CRYPTO_DEV_ARM_Crypto=m
11707 CONFIG_CRYPTO_DEV_ARM_Crypto=m
11708 CONFIG_CRYPTO_DEV_ARM_Crypto=m
11709 CONFIG_CRYPTO_DEV_ARM_Crypto=m
11710 CONFIG_CRYPTO_DEV_ARM_Crypto=m
11711
11712 #
11713 # Certificates for signature checking
11714 #
11715 CONFIG_MODULE_SIG_KEY="certs/combined.pem"
11716 CONFIG_MODULE_SIG_KEY_TYPE_RSA=y
11717 # CONFIG_MODULE_SIG_KEY_TYPE_ECDSA is not set
11718 CONFIG_MODULE_SIG_KEY_TYPE_ECDSA=y
11719 CONFIG_MODULE_SIG_KEY_TYPE_ECDSA=y
11720 CONFIG_MODULE_SIG_KEY_TYPE_ECDSA=y
11721 CONFIG_MODULE_SIG_KEY_TYPE_ECDSA=y
11722 CONFIG_MODULE_SIG_KEY_TYPE_ECDSA=y
11723 # CONFIG_MODULE_SIG_KEY_TYPE_ECDSA is not set
11724 CONFIG_MODULE_SIG_KEY_TYPE_ECDSA=y
11725 CONFIG_MODULE_SIG_KEY_TYPE_ECDSA=y
11726 CONFIG_MODULE_SIG_KEY_TYPE_ECDSA=y
11727 CONFIG_MODULE_SIG_KEY_TYPE_ECDSA=y
11728 # CONFIG_MODULE_SIG_KEY_TYPE_ECDSA is not set
11729 # end of Certificates for signature checking
11730
11731 CONFIG_BINARY_PRINTF=y
11732
11733 #
```



```
Open  [icon] *.config ~/Desktop/linux-6.9.6
955 CONFIG_FUNCTION_ALIGNMENT_16B=y
956 CONFIG_FUNCTION_ALIGNMENT_16B=y
957 # end of General architecture-dependent options
958
959 CONFIG_RT_MUTEXES=y
960 CONFIG_BASE_SMALL=0
961 CONFIG_MODULE_SIG_FORMAT=y
962 CONFIG_MODULES=y
963 # CONFIG_MODULE_DEBUG is not set
964 # CONFIG_MODULE_FORCE_LOAD is not set
965 CONFIG_MODULE_UNLOAD=y
966 # CONFIG_MODULE_FORCE_UNLOAD is not set
967 # CONFIG_MODULE_UNLOAD_TAINT_TRACKING is not set
968 CONFIG_MODULEVERSIONS=y
969 CONFIG_ASM_MODULEVERSIONS=y
970 CONFIG_MODULE_SRCVERSION_ALL=y
971 CONFIG_MODULE_SIG=y
972 CONFIG_MODULE_SIG_FORCE=y
973 CONFIG_MODULE_SIG_ALL=y
974 # CONFIG_MODULE_SIG_SHA1 is not set
975 # CONFIG_MODULE_SIG_SHA256 is not set
976 # CONFIG_MODULE_SIG_SHA384 is not set
977 CONFIG_MODULE_SIG_SHA512=y
978 # CONFIG_MODULE_SIG_SHA3_256 is not set
979 # CONFIG_MODULE_SIG_SHA3_384 is not set
980 # CONFIG_MODULE_SIG_SHA3_512 is not set
981 CONFIG_MODULE_SIG_HASH="sha512"
982 CONFIG_MODULE_COMPRESS_NONE=y
983 # CONFIG_MODULE_COMPRESS_GZIP is not set
984 # CONFIG_MODULE_COMPRESS_XZ is not set
985 # CONFIG_MODULE_COMPRESS_ZSTD is not set
986 # CONFIG_MODULE_ALLOW_MISSING_NAMESPACE_IMPORTS is not set
987 CONFIG_MODULE_PATH="/sbin/modprobe"
988 # CONFIG_TRIM_UNUSED_KSYMS is not set
989 CONFIG_MODULES_TREE_LOOKUP=y
990 CONFIG_BLOCK=y
991 CONFIG_BLOCK_LEGACY_AUTOLOAD=y
```

Figure 4: Kernel Configuration

2.3 Kernel Building:

The kernel will be configured to use the newly added certificate, then build the kernel. This process includes compiling the kernel and its modules, installing the new kernel, updating the bootloader, and rebooting the system to load the new kernel.

2.3.1 Kernel Compiling:

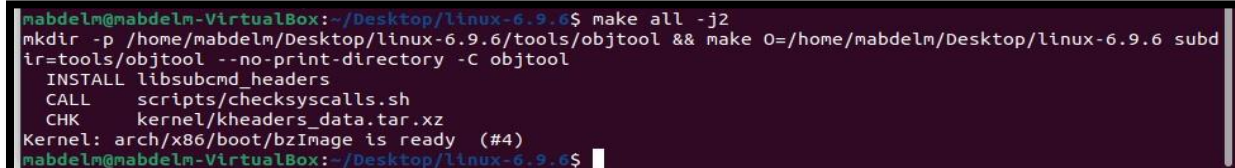
- The kernel source directory will be navigated and clean the build environment to ensure a fresh build:

```
cd /usr/src/linux-6.9.9/make clean
make mrproper
```

- **'make clean'**: Removes most generated files.
 - **'make mrproper'**: Removes additional files including the configuration file.
- Next, Configuration of the kernel:

```
make oldconfig
make -j$(nproc)
```

- **'make oldconfig'**: Uses the existing configuration file, updating it with any new options.
- **'make -j\$(nproc)'**: Compiles the kernel using all available CPU cores to speed up the process.

A terminal window with a dark purple background and light green text. The text shows the execution of 'make all -j2' in a directory ~/Desktop/linux-6.9.6. It lists various steps like 'mkdir', 'install', 'call', and 'chk' for different components, and finally reports 'Kernel: arch/x86/boot/bzImage is ready (#4)'.

```
mabdelm@mabdelm-VirtualBox:~/Desktop/linux-6.9.6$ make all -j2
mkdir -p /home/mabdelm/Desktop/linux-6.9.6/tools/objtool && make O=/home/mabdelm/Desktop/linux-6.9.6 subd
ir=tools/objtool --no-print-directory -C objtool
INSTALL libsubcmd_headers
CALL scripts/checksyscalls.sh
CHK kernel/kheaders_data.tar.xz
Kernel: arch/x86/boot/bzImage is ready (#4)
mabdelm@mabdelm-VirtualBox:~/Desktop/linux-6.9.6$
```

Figure 5: Kernel Compiling

2.3.2 Kernel Installing:

Once the kernel and modules are compiled, then installed them:

```
sudo make modules_install
sudo make install
```

- **'make modules_install'**: Installs the kernel modules.
- **'make install'**: Installs the kernel, System.map, and configuration files.

```
mabdelm@mabdelm-VirtualBox: ~/Desktop/linux-6.9.6
INSTALL /lib/modules/6.9.6/kernel/net/vmw_vsock/vmw_vsock_vmci_transport.ko
SIGN /lib/modules/6.9.6/kernel/net/vmw_vsock/vmw_vsock_vmci_transport.ko
INSTALL /lib/modules/6.9.6/kernel/net/vmw_vsock/vmw_vsock_virtio_transport.ko
SIGN /lib/modules/6.9.6/kernel/net/vmw_vsock/vmw_vsock_virtio_transport.ko
INSTALL /lib/modules/6.9.6/kernel/net/vmw_vsock/vmw_vsock_virtio_transport_common.ko
SIGN /lib/modules/6.9.6/kernel/net/vmw_vsock/vmw_vsock_virtio_transport_common.ko
INSTALL /lib/modules/6.9.6/kernel/net/vmw_vsock/hv_sock.ko
SIGN /lib/modules/6.9.6/kernel/net/vmw_vsock/hv_sock.ko
INSTALL /lib/modules/6.9.6/kernel/net/vmw_vsock/vsock_loopback.ko
SIGN /lib/modules/6.9.6/kernel/net/vmw_vsock/vsock_loopback.ko
INSTALL /lib/modules/6.9.6/kernel/net/nsh/nsh.ko
SIGN /lib/modules/6.9.6/kernel/net/nsh/nsh.ko
INSTALL /lib/modules/6.9.6/kernel/net/hsr/hsr.ko
SIGN /lib/modules/6.9.6/kernel/net/hsr/hsr.ko
INSTALL /lib/modules/6.9.6/kernel/net/qrtr/qrtr.ko
SIGN /lib/modules/6.9.6/kernel/net/qrtr/qrtr.ko
INSTALL /lib/modules/6.9.6/kernel/net/qrtr/qrtr-smc.ko
SIGN /lib/modules/6.9.6/kernel/net/qrtr/qrtr-smc.ko
INSTALL /lib/modules/6.9.6/kernel/net/qrtr/qrtr-tun.ko
SIGN /lib/modules/6.9.6/kernel/net/qrtr/qrtr-tun.ko
INSTALL /lib/modules/6.9.6/kernel/net/qrtr/qrtr-mhi.ko
SIGN /lib/modules/6.9.6/kernel/net/qrtr/qrtr-mhi.ko
DEPMOD /lib/modules/6.9.6
mabdelm@mabdelm-VirtualBox: ~/Desktop/linux-6.9.6$
```

Figure 6: Kernel Install

2.3.3 Updating the Bootloader:

After installation, update the GRUB bootloader to recognize the new kernel:

```
sudo update-grub
```

- This command regenerates the GRUB configuration file ('**grub.cfg**'), including the new kernel in the boot menu.

2.3.4 System Rebooting:

Finally, rebooting the system to load the newly compiled kernel with the incorporated certificate:

- Upon reboot, the kernel will utilize the specified certificate for module signing, ensuring that only signed modules are loaded.

```
sudo reboot
```

2.3.5 Kernel Test:

```
mabdelm@mabdelm-VirtualBox: ~/Desktop/linux-6.9.6
mabdelm@mabdelm-VirtualBox:~/Desktop/linux-6.9.6$ uname -r
5.9.6
mabdelm@mabdelm-VirtualBox:~/Desktop/linux-6.9.6$
```

Figure 7: Kernel Check

Figure 7 shows the kernel output after checking.

2.4. Implementing and Signing Kernel Modules:

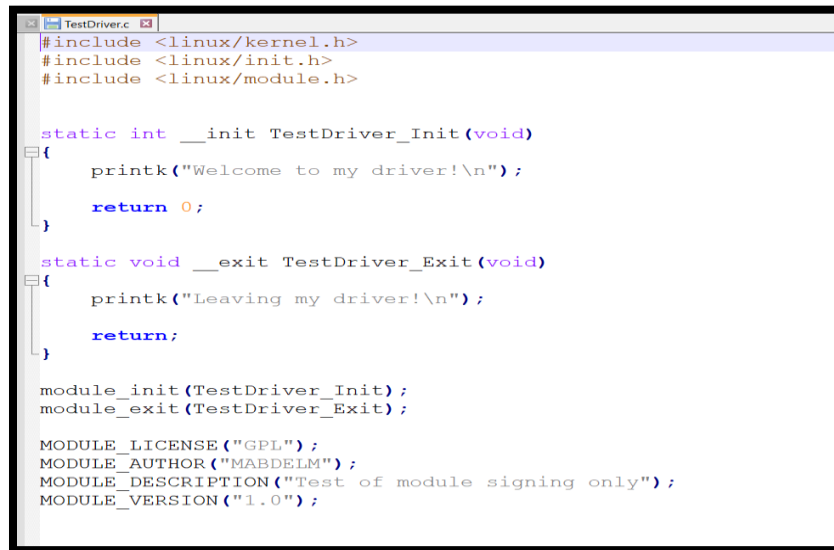
The implementation and signing of kernel modules are critical for ensuring that only authenticated code is loaded into the kernel, thereby maintaining system integrity and security. This section provides a detailed walkthrough on creating a kernel module, compiling it, and signing it using the previously generated cryptographic keys and certificates.

2.4.1 Kernel Module Creating:

The simple dummy kernel module will be created as an example. This module will serve as a demonstration for the signing process.

➤ Source Code Creation:

Create a source file named '**TestDriver.c**' with the following content:

A screenshot of a text editor window titled 'TestDriver.c'. The code is as follows:

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>

static int __init TestDriver_Init(void)
{
    printk("Welcome to my driver!\n");
    return 0;
}

static void __exit TestDriver_Exit(void)
{
    printk("Leaving my driver!\n");
    return;
}

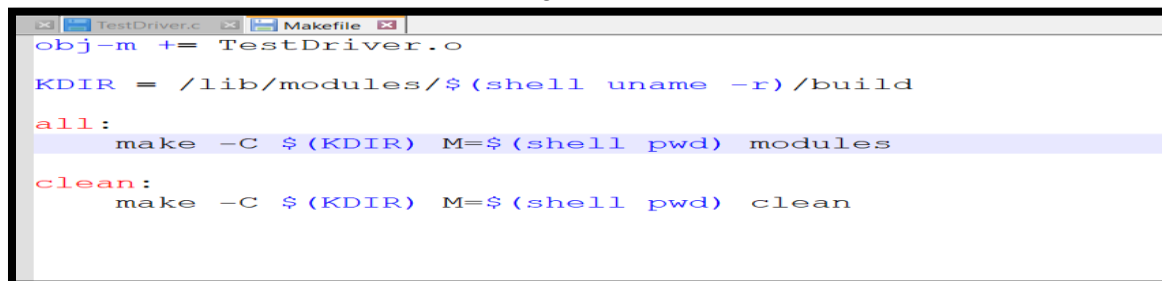
module_init(TestDriver_Init);
module_exit(TestDriver_Exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("MABDELM");
MODULE_DESCRIPTION("Test of module signing only");
MODULE_VERSION("1.0");
```

Figure 8: Source Code

➤ Makefile Creation:

Create a '**Makefile**' to facilitate the building of the kernel module:

A screenshot of a text editor window titled 'Makefile'. The content is as follows:

```
obj-m += TestDriver.o

KDIR = /lib/modules/$(shell uname -r)/build

all:
    make -C $(KDIR) M=$(shell pwd) modules

clean:
    make -C $(KDIR) M=$(shell pwd) clean
```

Figure 9: Makefile

2.4.2 Kernel Module Compiling:

With the source code and Makefile in place, the kernel module will be compiled as follows:

make

- This command leverages the kernel build system to compile the **'TestDriver.c'** source file into a loadable kernel object file (**'TestDriver.ko'**).

2.4.3 Signing the Module:

```
linux-6.9.6/scripts/sign-file sha256 /home/mabdelm/Desktop/private_key.pem
/home/mabdelm/Desktop/certificate.pem
/home/mabdelm/Desktop/EmbeddedModule/TestDriver.ko
```

Using this **'sign-file'** command script provided by the kernel to sign the module:

- **'sha256'** specifies the hashing algorithm.
- **'/path/to/private_key.pem'** is the path to the private key file.
- **'/path/to/certificate.pem'** is the path to the certificate file.
- **'TestDriver.ko'** is the kernel module to be signed.
- This command generates a **'p7s'** signature file and embeds the signature into the module file.

The screenshot shows a terminal window titled 'mabdelm@mabdelm-VirtualBox: ~/Desktop/EmbeddedSecurity'. The user runs the following commands:

```
mabdelm@mabdelm-VirtualBox:~/Desktop/EmbeddedSecurity$ make
make -C /lib/modules/6.9.6/build M=/home/mabdelm/Desktop/EmbeddedSecurity modules
make[1]: Entering directory '/home/mabdelm/Desktop/linux-6.9.6'
  CC [M] /home/mabdelm/Desktop/EmbeddedSecurity/TestDriver.o
  MODPOST /home/mabdelm/Desktop/EmbeddedSecurity/Module.symvers
  CC [M] /home/mabdelm/Desktop/EmbeddedSecurity/TestDriver.mod.o
  LD [M] /home/mabdelm/Desktop/EmbeddedSecurity/TestDriver.ko
make[1]: Leaving directory '/home/mabdelm/Desktop/linux-6.9.6'
mabdelm@mabdelm-VirtualBox:~/Desktop/EmbeddedSecurity$ ../linux-6.9.6/scripts/sign-file sha256
/home/mabdelm/Desktop/private_key.pem /home/mabdelm/Desktop/certificate.pem TestDriver.ko
mabdelm@mabdelm-VirtualBox:~/Desktop/EmbeddedSecurity$ sudo insmod TestDriver.ko
[sudo] password for mabdelm:
mabdelm@mabdelm-VirtualBox:~/Desktop/EmbeddedSecurity$ sudo dmesg
[ 0.000000] Linux version 6.9.6 (mabdelm@mabdelm-VirtualBox) (gcc (Ubuntu 11.4.0-1ubuntu1~22
.04) 11.4.0, GNU ld (GNU Binutils for Ubuntu) 2.38) #4 SMP PREEMPT_DYNAMIC Wed Jun 26 08:16:17
CEST 2024
[ 0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-6.9.6 root=UUID=b314611d-42d4-4cbf-a47a-d
890ec7fec69 ro quiet splash vt.handoff=7
[ 0.000000] KERNEL supported cpus:
[ 0.000000] Intel GenuineIntel
[ 0.000000] AMD AuthenticAMD
[ 0.000000] Hygon HygonGenuine
[ 0.000000] Centaur CentaurHauls
[ 0.000000] zhaoxin Shanghai
[ 0.000000] BIOS-provided physical RAM map:
```

Figure 10: Module Signing

2.4.4 Loading and Unloading the Signed Kernel Module:

To validate the signed kernel module, it must be loaded into and unloaded from the kernel. The kernel will verify the module's signature during these operations.

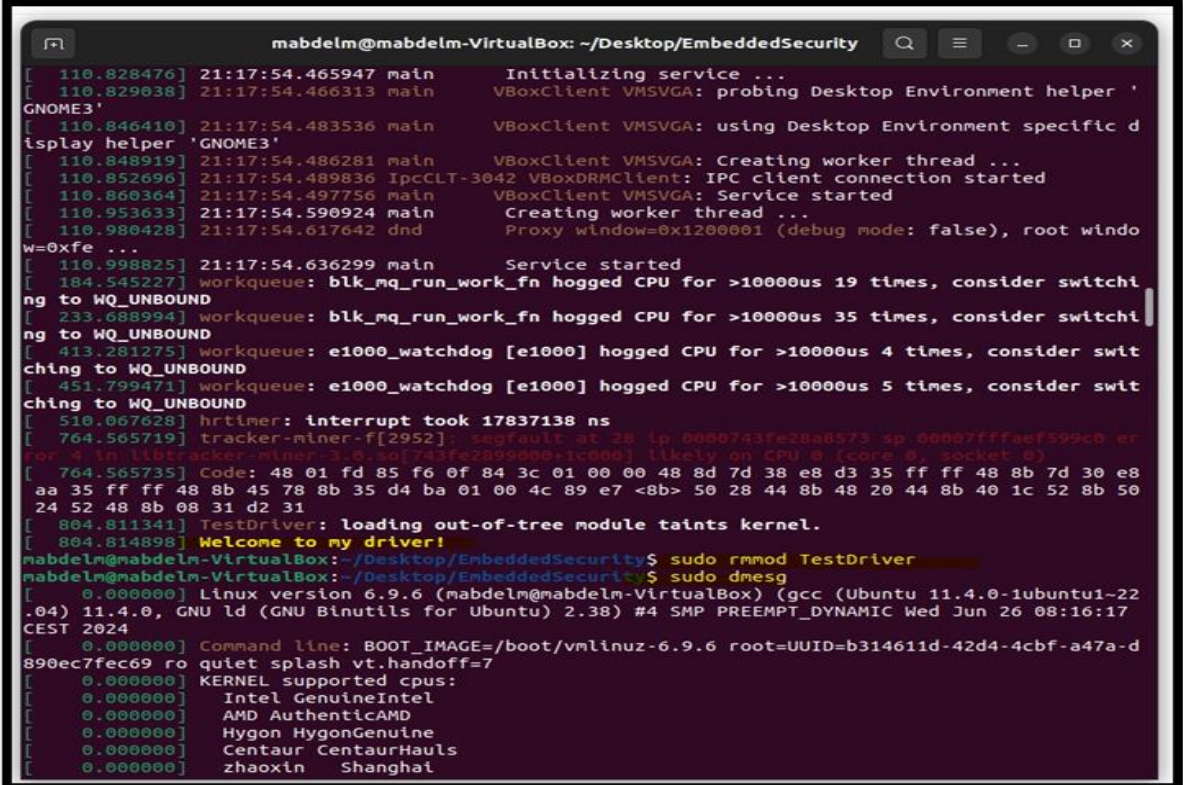
➤ Loading the Module:

```
# Load kernel module called TestDriver.ko to the kernel
dynamically
sudo insmod TestDriver.ko
```

- This command inserts the module into the kernel.

Verify successful loading via kernel logs:

```
sudo dmesg
```



```
mabdelm@mabdelm-VirtualBox: ~/Desktop/EmbeddedSecurity
[ 110.828476] 21:17:54.465947 main      Initializing service ...
[ 110.829038] 21:17:54.466313 main      VBoxClient VMSVGA: probing Desktop Environment helper '
GNOME3'
[ 110.846410] 21:17:54.483536 main      VBoxClient VMSVGA: using Desktop Environment specific d
isplay helper 'GNOME3'
[ 110.848919] 21:17:54.486281 main      VBoxClient VMSVGA: Creating worker thread ...
[ 110.852696] 21:17:54.489836 IpcCLT-3042 VBoxDRMClient: IPC client connection started
[ 110.860364] 21:17:54.497756 main      VBoxClient VMSVGA: Service started
[ 110.953633] 21:17:54.590924 main      Creating worker thread ...
[ 110.980428] 21:17:54.617642 dnd       Proxy window=0x1200001 (debug mode: false), root windo
w=0xfe ...
[ 110.998825] 21:17:54.636299 main      Service started
[ 184.545227] workqueue: blk_mq_run_work_fn hogged CPU for >10000us 19 times, consider switchi
ng to WQ_UNBOUND
[ 233.688994] workqueue: blk_mq_run_work_fn hogged CPU for >10000us 35 times, consider switchi
ng to WQ_UNBOUND
[ 413.281275] workqueue: e1000_watchdog [e1000] hogged CPU for >10000us 4 times, consider swit
ching to WQ_UNBOUND
[ 451.799471] workqueue: e1000_watchdog [e1000] hogged CPU for >10000us 5 times, consider swit
ching to WQ_UNBOUND
[ 510.067628] hrtimer: interrupt took 17837138 ns
[ 764.565719] tracker-miner-f[2952]: segfault at 28 ip 00007f43fe28a8573 sp 00007fff599cb er
ror 4 in libtracker-miner-3.0.so[743fe2899000-1c000] likely on CPU 0 (core 0, socket 0)
[ 764.565735] Code: 48 01 fd 85 f6 0f 84 3c 01 00 00 48 8d 7d 38 e8 d3 35 ff ff 48 8b 7d 30 e8
aa 35 ff ff 48 8b 45 78 8b 35 d4 ba 01 00 4c 89 e7 <8b> 50 28 44 8b 48 20 44 8b 40 1c 52 8b 50
24 52 48 8b 08 31 d2 31
[ 804.811341] TestDriver: loading out-of-tree module taints kernel.
[ 804.814898] Welcome to my driver!
mabdelm@mabdelm-VirtualBox: ~/Desktop/EmbeddedSecurity$ sudo rmmod TestDriver
mabdelm@mabdelm-VirtualBox: ~/Desktop/EmbeddedSecurity$ sudo dmesg
[ 0.000000] Linux version 6.9.6 (mabdelm@mabdelm-VirtualBox) (gcc (Ubuntu 11.4.0-1ubuntu1-22
.04) 11.4.0, GNU ld (GNU Binutils for Ubuntu) 2.38) #4 SMP PREEMPT_DYNAMIC Wed Jun 26 08:16:17
CEST 2024
[ 0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-6.9.6 root=UUID=b314611d-42d4-4cbf-a47a-d
890ec7fec69 ro quiet splash vt.handoff=7
[ 0.000000] KERNEL supported cpus:
[ 0.000000] Intel GenuineIntel
[ 0.000000] AMD AuthenticAMD
[ 0.000000] Hygon HygonGenuine
[ 0.000000] Centaur CentaurHauls
[ 0.000000] zhaoxin Shanghai
```

Figure 11: Kernel Log Login

➤ Unloading the Module:

```
# Remove kernel module called TestDriver.ko to the kernel
dynamically
sudo rmmod TestDriver
```

- This command removes the module from the kernel.

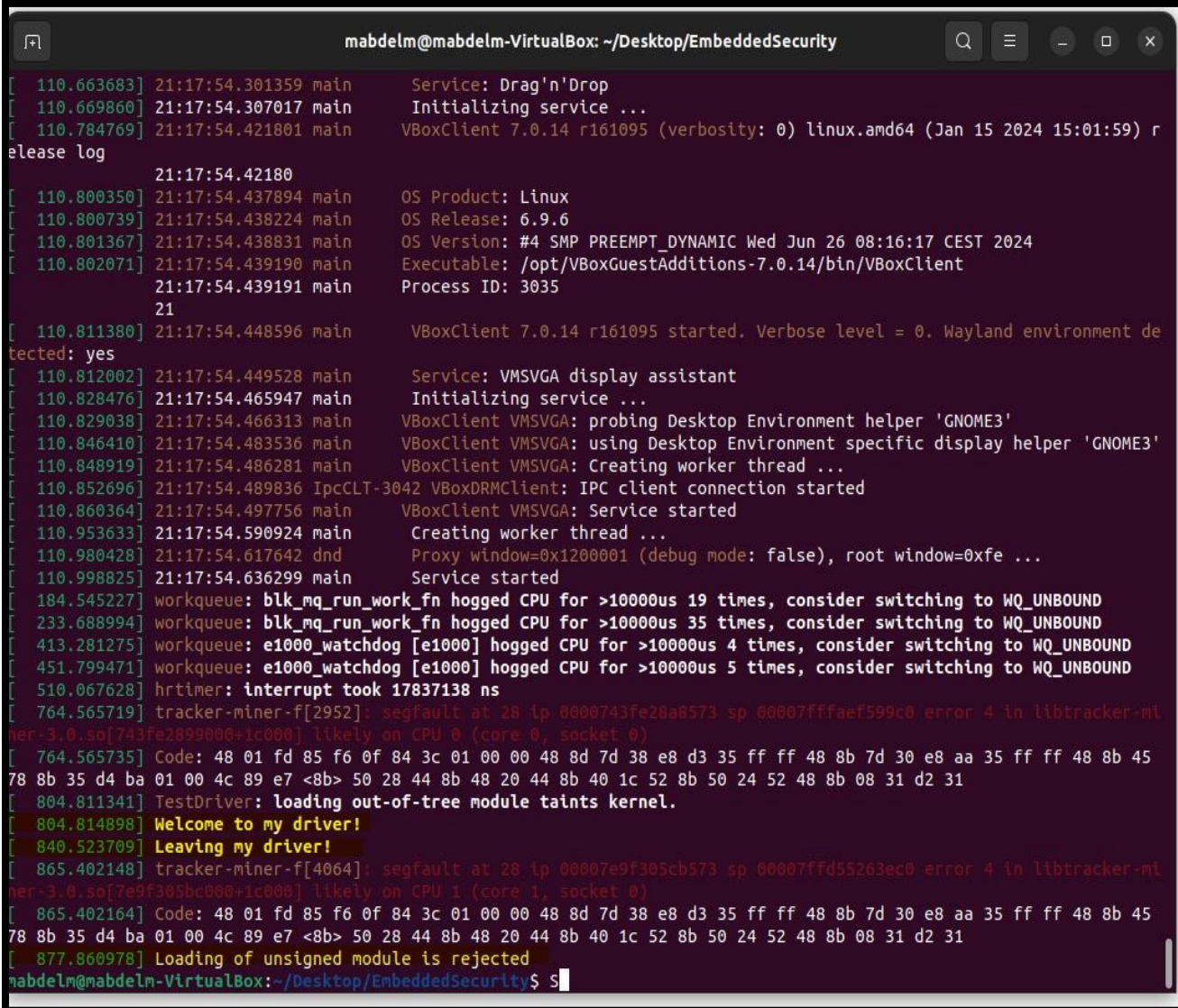
Checking the kernel logs to confirm unloading:

```
mabdelm@mabdelm-VirtualBox: ~/Desktop/EmbeddedSecurity
[ 110.848919] 21:17:54.486281 main VBoxClient VMSVGA: Creating worker thread ...
[ 110.852696] 21:17:54.489836 IpcCLT-3042 VBoxDRMClient: IPC client connection started
[ 110.860364] 21:17:54.497756 main VBoxClient VMSVGA: Service started
[ 110.953633] 21:17:54.590924 main Creating worker thread ...
[ 110.980428] 21:17:54.617642 dnd Proxy window=0x1200001 (debug mode: false), root window=0xfe ...
[ 110.998825] 21:17:54.636299 main Service started
[ 184.545227] workqueue: blk_mq_run_work_fn hogged CPU for >10000us 19 times, consider switching to WQ_UNBOUND
[ 233.688994] workqueue: blk_mq_run_work_fn hogged CPU for >10000us 35 times, consider switching to WQ_UNBOUND
[ 413.281275] workqueue: e1000_watchdog [e1000] hogged CPU for >10000us 4 times, consider switching to WQ_UNBOUND
[ 451.799471] workqueue: e1000_watchdog [e1000] hogged CPU for >10000us 5 times, consider switching to WQ_UNBOUND
[ 510.067628] hrtimer: interrupt took 17837138 ns
[ 764.565719] tracker-miner-f[2952]: segfault at 28 ip 0000743fe28a8573 sp 00007ffffae599c0 error 4 in libtracker-miner-3.0.so[743fe2899000+1c000] likely on CPU 0 (core 0, socket 0)
[ 764.565735] Code: 48 01 fd 85 f6 0f 84 3c 01 00 00 48 8d 7d 38 e8 d3 35 ff ff 48 8b 7d 30 e8 aa 35 ff ff 48 8b 45 78 8b 35 d4 ba 01 00 4c 89 e7 <8b> 50 28 44 8b 48 20 44 8b 40 1c 52 8b 50 24 52 48 8b 08 31 d2 31
[ 804.811341] TestDriver: loading out-of-tree module taints kernel.
[ 804.814898] Welcome to my driver!
[ 840.523709] Leaving my driver!
mabdelm@mabdelm-VirtualBox:~/Desktop/EmbeddedSecurity$ make clean
make -C /lib/modules/6.9.6/build M=/home/mabdelm/Desktop/EmbeddedSecurity clean
make[1]: Entering directory '/home/mabdelm/Desktop/linux-6.9.6'
CLEAN /home/mabdelm/Desktop/EmbeddedSecurity/Module.symvers
make[1]: Leaving directory '/home/mabdelm/Desktop/linux-6.9.6'
mabdelm@mabdelm-VirtualBox:~/Desktop/EmbeddedSecurity$ make
make -C /lib/modules/6.9.6/build M=/home/mabdelm/Desktop/EmbeddedSecurity modules
make[1]: Entering directory '/home/mabdelm/Desktop/linux-6.9.6'
CC [M] /home/mabdelm/Desktop/EmbeddedSecurity/TestDriver.o
MODPOST /home/mabdelm/Desktop/EmbeddedSecurity/Module.symvers
CC [M] /home/mabdelm/Desktop/EmbeddedSecurity/TestDriver.mod.o
LD [M] /home/mabdelm/Desktop/EmbeddedSecurity/TestDriver.ko
make[1]: Leaving directory '/home/mabdelm/Desktop/linux-6.9.6'
mabdelm@mabdelm-VirtualBox:~/Desktop/EmbeddedSecurity$ sudo insmod TestDriver.ko
insmod: ERROR: could not insert module TestDriver.ko: Key was rejected by service
```

Figure 12: Kernel Logs Checks

2.4.5 Comparison with Unsigned Modules:

To demonstrate the security benefit of signing, a try to load an unsigned module is attempted for comparing the behavior of loading both signed and unsigned module. So, if the kernel is configured to enforce signature verification, this command should fail. After Checking the kernel log for errors related to module signature verification, The log messages should indicate the failure due to an invalid or missing signature as shown next. Figure 13 represents the Whole kernel log demonstration of all cases:



```
mabdelm@mabdelm-VirtualBox: ~/Desktop/EmbeddedSecurity
[ 110.663683] 21:17:54.301359 main Service: Drag'n'Drop
[ 110.669860] 21:17:54.307017 main Initializing service ...
[ 110.784769] 21:17:54.421801 main VBoxClient 7.0.14 r161095 (verbosity: 0) linux.amd64 (Jan 15 2024 15:01:59) r
elease log
[ 110.800350] 21:17:54.421800 OS Product: Linux
[ 110.800739] 21:17:54.438224 main OS Release: 6.9.6
[ 110.801367] 21:17:54.438831 main OS Version: #4 SMP PREEMPT_DYNAMIC Wed Jun 26 08:16:17 CEST 2024
[ 110.802071] 21:17:54.439190 main Executable: /opt/VBoxGuestAdditions-7.0.14/bin/VBoxClient
[ 110.802071] 21:17:54.439191 main Process ID: 3035
[ 110.811380] 21:17:54.448596 main VBoxClient 7.0.14 r161095 started. Verbose level = 0. Wayland environment de
tected: yes
[ 110.812002] 21:17:54.449528 main Service: VMSVGA display assistant
[ 110.828476] 21:17:54.465947 main Initializing service ...
[ 110.829038] 21:17:54.466313 main VBoxClient VMSVGA: probing Desktop Environment helper 'GNOME3'
[ 110.846410] 21:17:54.483536 main VBoxClient VMSVGA: using Desktop Environment specific display helper 'GNOME3'
[ 110.848919] 21:17:54.486281 main VBoxClient VMSVGA: Creating worker thread ...
[ 110.852696] 21:17:54.489836 IpcCLT-3042 VBoxDRMClient: IPC client connection started
[ 110.860364] 21:17:54.497756 main VBoxClient VMSVGA: Service started
[ 110.953633] 21:17:54.590924 main Creating worker thread ...
[ 110.980428] 21:17:54.617642 dnd Proxy window=0x1200001 (debug mode: false), root window=0xfe ...
[ 110.998825] 21:17:54.636299 main Service started
[ 184.545227] workqueue: blk_mq_run_work_fn hogged CPU for >10000us 19 times, consider switching to WQ_UNBOUND
[ 233.688994] workqueue: blk_mq_run_work_fn hogged CPU for >10000us 35 times, consider switching to WQ_UNBOUND
[ 413.281275] workqueue: e1000_watchdog [e1000] hogged CPU for >10000us 4 times, consider switching to WQ_UNBOUND
[ 451.799471] workqueue: e1000_watchdog [e1000] hogged CPU for >10000us 5 times, consider switching to WQ_UNBOUND
[ 510.067628] hrtimer: interrupt took 17837138 ns
[ 764.565719] tracker-miner-f[2952]: segfault at 28 ip 0000743fe28a8573 sp 00007ffffaef599c0 error 4 in libtracker-ml
ner-3.0.so[743fe2899800+1c000] likely on CPU 0 (core 0, socket 0)
[ 764.565735] Code: 48 01 fd 85 f6 0f 84 3c 01 00 00 48 8d 7d 38 e8 d3 35 ff ff 48 8b 7d 30 e8 aa 35 ff ff 48 8b 45
78 8b 35 d4 ba 01 00 4c 89 e7 <8b> 50 28 44 8b 48 20 44 8b 40 1c 52 8b 50 24 52 48 8b 08 31 d2 31
[ 804.811341] TestDriver: loading out-of-tree module taints kernel.
[ 804.814898] Welcome to my driver!
[ 840.523709] Leaving my driver!
[ 865.402148] tracker-miner-f[4064]: segfault at 28 ip 00007e9f305cb573 sp 00007ffdf55263ec0 error 4 in libtracker-ml
ner-3.0.so[7e9f305bc000+1c000] likely on CPU 1 (core 1, socket 0)
[ 865.402164] Code: 48 01 fd 85 f6 0f 84 3c 01 00 00 48 8d 7d 38 e8 d3 35 ff ff 48 8b 7d 30 e8 aa 35 ff ff 48 8b 45
78 8b 35 d4 ba 01 00 4c 89 e7 <8b> 50 28 44 8b 48 20 44 8b 40 1c 52 8b 50 24 52 48 8b 08 31 d2 31
[ 877.860978] Loading of unsigned module is rejected
mabdelm@mabdelm-VirtualBox:~/Desktop/EmbeddedSecurity$
```

Figure 13: Whole Kernel Log

3. Hardware Implementation:

3.1 Approaches:

In the domain of embedded systems and secure computing environments, maintaining the integrity and authenticity of software components like kernels and modules is of utmost importance. This is especially critical for devices such as the Raspberry Pi, where security vulnerabilities can lead to substantial repercussions. The process of implementing signed kernels and modules, accompanied by thorough security assessments, entails using cryptographic methods to validate the source and integrity of software components before permitting their execution. There are several approaches to implement and compile the discussed software project into an embedded hardware chip computer like a raspberry pi. These methods may offer specific advantages or be suitable depending on the development environment and requirements. Here is a quick elaboration on some notable techniques:

3.1.1 Native Compilation: involves compiling software directly on the target hardware platform, such as the Raspberry Pi. It's usually used in Small-scale projects or prototypes where performance is less critical such as educational purposes or development environments where simplicity outweighs compilation time.

3.1.2 Virtual Machines or Emulation: Using virtualization or emulation to run the target architecture on a host machine and compile software within that virtual environment. Usually used in Development environments where access to actual hardware is limited or impractical. Also, applied in testing software across different architectures or platforms.

3.1.3 Remote Compilation Servers: Using remote servers or building farms to perform compilation tasks on behalf of developers. It is used in Large-scale software projects where compilation times are critical. Continuous integration (CI) or automated build systems requiring distributed compilation resources.

3.1.4 Cross Compilation: It is the process of compiling code on one platform (the host) to produce executables for a different platform (the target). This is particularly useful for embedded systems like the Raspberry Pi, which have limited processing power and memory, making it impractical to perform the compilation natively on the device. It offers efficiency, speed, and access to better development tools but requires a complex setup, managing dependencies, and can present debugging challenges. Properly addressing these issues is key to successful cross-platform development.

Therefore, that is the used approach for this project demonstration and implementation onto a Raspberry Pi board. It is discussed with details in the next section.

3.2 Building a Signed Kernel and Signed Module with the Same Keys and Certificate on a Raspberry Pi Using Cross-Compilation:

There are two major steps to cross compile and insure a successful insertion of the application into a raspberry pi chip. Both have inside subsequent steps.

Firstly, a Native compilation of the kernel and its modules on a host machine with altered commands to suit the new environment of the target machine.

Secondly, transfer what is done in the host computer plus the private key and certificate to the target device which is the raspberry pi. Then putting the application in use by running the commands and the desired operations which in this case is making a successful load to a signed module.

Note: from this point on, the demonstration of commands and codes are standard in general forms. due to the lack of actual application on a hardware. So, the directories, usernames and libraries calling are to be assumed.

3.2.1 Native Compilation:

Reaching the point of successfully building and signing a kernel and with the corresponding certificate, is considered the first part of implementing this project into a raspberry pi but with some small differences and adjustments regarding cross compilation. More kernel headers might need to be installed depending on whether setting up the kernel from scratch or cloning it completely from preset kernel that is suitable for such implementation. These commands and steps are to be demonstrated with the preservation that the concept remains the same as elaborated.

- **Setting Up the Cross-Compilation Environment:** by installing the necessary tools for cross-compilation on your host machine and Downloading the Raspberry Pi kernel source code and the appropriate cross-compilation toolchain.

```
# Installment of the tools: sudo apt-get update sudo apt-get
install gcc-arm-linux-gnueabi make bc bison flex libssl-dev #
Downloading the Raspberry Pi kernel source code and toolchain: git
clone --depth=1 https://github.com/raspberrypi/linux cd linux
```

Configuring the Kernel: to include support for module signature verification and to use the generated keys.

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- bcmrpi_defconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
```

Navigate to Enable loadable module support and set the following options:

- Module signature verification (CONFIG_MODULE_SIG)
- Require modules to be validly signed (CONFIG_MODULE_SIG_ALL)
- File name or PKCS#11 URI of module signing key to the path of signing_key.priv

- **Build the Kernel:**

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j$(nproc) zImage  
modules dtbs
```

3.2.2 Cross-Compilation and Transition:

The step of installing the kernel and building the modules involves transferring the compiled kernel image and the certificate alongside the private key to the Raspberry Pi and properly configuring them to be used by the system. This procedure ensures that the Raspberry Pi can utilize the newly built kernel and modules for testing or deployment. The following outline demonstrates the steps to achieve this:

➤ **Preparation Prior to transferring, ensure the following:**

- **Compilation Completion:** Confirmation that the kernel and modules have been successfully compiled on the host machine using cross-compilation tools.
- **Network Connectivity:** Verify that both the host machine and the Raspberry Pi are connected to the same network, either through Ethernet or Wi-Fi.
- **SSH Access:** Ensure SSH access to the Raspberry Pi. If SSH is not enabled, activate it on the Raspberry Pi and confirm its accessibility from the host machine.

➤ **Transferring Files to the Raspberry Pi:**

Once the kernel and modules on the host machine are compiled, the next step is to transfer these files to the Raspberry Pi. This can be done using various methods, such as SCP (Secure Copy Protocol) or a USB drive.

- **Using SCP:**

Transfer the Kernel Image (zImage or Image):

```
scp arch/arm/boot/zImage pi@raspberrypi:/home/pi/
```

Transfer Device Tree Blobs (*.dtb files):

```
scp arch/arm/boot/dts/*.dtb pi@raspberrypi:/home/pi/
```

Transfer Kernel Modules:

```
scp -r output/modules/lib/modules/$(make kernelversion)  
pi@raspberrypi:/home/pi/modules/
```

Transfer the Signed Module to the Raspberry Pi (e.g., hello.ko):

```
scp hello.ko pi@raspberrypi:/home/pi/
```

- **Using a USB Drive:**

Copy Files to USB Drive: Copy the kernel image, device tree blobs, and modules directory to a USB drive.

Mount USB Drive on Raspberry Pi: Connect the USB drive to the Raspberry Pi and mount it:

```
sudo mkdir /mnt/usb
sudo mount /dev/sda1 /mnt/usb    # Adjust /dev/sda1 to your USB
drive's device identifier
```

Copy Files from USB to Raspberry Pi:

```
sudo cp /mnt/usb/zImage /home/pi/
sudo cp /mnt/usb/*.dtb /home/pi/
sudo cp -r /mnt/usb/modules/lib/modules/$(make kernelversion)
/home/pi/modules/
```

3.2.3 Installing the Kernel Image:

The kernel image needs to be put in the **‘/boot’** directory of the Raspberry Pi’s filesystem.

Backup Existing Kernel:

```
sudo cp /boot/kernel7.img /boot/kernel7.img.bak
```

Copy the New Kernel Image:

```
sudo cp /home/pi/zImage /boot/kernel7.img
```

If a different Raspberry Pi model or kernel version is in use, the kernel filename may differ (kernel.img, kernel8.img, etc.).

3.2.4 Installing Device Tree Blobs:

Device Tree Blobs (DTBs) describe the hardware layout to the kernel. They must be placed in the **‘/boot’** directory.

Copy the DTBs:

```
sudo cp /home/pi/*.dtb /boot/
```

3.2.5 Installing Kernel Modules:

Kernel modules must be placed in the appropriate directory within **'/lib/modules'**.

Copy Kernel Modules:

```
sudo cp -r /home/pi/modules/lib/modules/${make
kernelversion} /lib/modules/
```

Replace `$(make kernelversion)` with the actual version number of the compiled kernel if needed.

3.2.6 Updating Boot Configuration:

Ensure that the bootloader is configured to use the new kernel. Typically, the Raspberry Pi uses the **'config.txt'** file in the **'/boot'** directory to configure boot settings.

Edit /boot/config.txt:

```
sudo nano /boot/config.txt
```

Specify the New Kernel (if different from the default): Add or modify the line:

```
kernel=kernel7.img    # Or appropriate kernel image name
```

3.2.7 Reboot the Raspberry Pi:

Rebooting the Raspberry Pi to load the new kernel and modules.

```
sudo reboot
```

3.2.8 Verifying the Installation:

Verification that the new kernel and modules are in use after rebooting.

Check Kernel Version to Ensure it matches the version of the newly installed kernel.:

```
uname -r
```

Check Loaded Modules:

```
lsmod
```


3.3 Compiling and signing the Kernel Module:

- Compile the Kernel Module: Assuming the module source code file is already built as mentioned before, then from the directory containing this kernel module source code the compiling is done using the kernel headers.
- Sign the Kernel Module: With the use of the existing transferred signed key and certificate the module is signed.

```
# Compiling the Kernel Module:
make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
# Signing the Kernel Module:
/usr/src/linux-headers-$(uname -r)/scripts/sign-file sha256
/home/pi/signing_key.priv /home/pi/signing_key.x509 mymodule.ko
```

3.4 Performing Loading and Unloading Checks for the Signed Module and unsigned one:

After implementing the signed kernel and module on the Raspberry Pi, it is important to verify that the module can be loaded and unloaded correctly. This process involves inserting the module into the kernel, verifying its operation, and then removing it.

Loading the Module

```
# Log into the Raspberry Pi:

ssh pi@raspberrypi

# Insert the Module: Use the insmod command to
load the module into the kernel.

sudo insmod /home/pi/hello.ko

# Verify the Module is Loaded: Check the kernel log
to confirm that the module was loaded successfully.

dmesg | tail

# A message should be appearing indicating that the
module has been inserted, such as:

[timestamp] Hello, World!

# List Loaded Modules: Ensure the module is listed
among the loaded modules.

lsmod | grep hello

This should show an entry for the hello module.
```

Unloading the Module

```
# Remove the Module: A use of the rmmod command to
unload the module from the kernel.

sudo rmmod hello

# Verify the Module is Unloaded: Checking the kernel log
again to confirm that the module was removed successfully.

dmesg | tail

# A message should be appearing indicating that the
module has been removed, such as:

[timestamp] Goodbye, World!

# List Loaded Modules: Ensure the module is no longer
listed among the loaded modules.

lsmod | grep hello

There should be no output, indicating the module has been
successfully unloaded.
```

- **Loading an unsigned module:**

Trying this in a system configured to require signed modules will result in an error. This is because the kernel is set up to verify the signature of each module before allowing it to be loaded. the kernel will check the module's signature and since the module is unsigned, this check will fail. The kernel will reject the module and produce an error message indicating that the module is unsigned or invalid. When the kernel encounters an unsigned module, it marks the kernel as "tainted". This means the kernel is in a state where it is running potentially unsafe or untrusted code. The '- **tainting kernel**' part of the message indicates this status.

Attempt to Load the Unsigned Module:

```
sudo insmod /path/to/unsigned_module.ko
```

By examining the kernel log.

```
dmesg | tail
```

A typical Error Messages: The `dmesg` output will show messages like the following:

```
[timestamp] module: unsigned_module: loading of unsigned
                    module is rejected
[timestamp] module: loading kernel module: module
verification failed: signature and/or required key
                    missing - tainting kernel
```

4. Challenge and Future Proposal

Challenges:

- Due to the limitations of our devices, the kernel had trouble to creating at the beginning, so more upgraded machine used in replace.
- The kernel took three hours to build in the second time, but it was not configured according to the purpose of the project.
- The kernel took eight hours to build at third attempt, but it was a successful build. The difference was that the configuration was done directly upon building as explained earlier and that might be the reason it took that time.
- An attempt was made to implement the project in a raspberry pi emulator as a first step of applying the idea, but it didn't work and it's believed this was due to complex kernel and bootloader configuration, cryptographic operation issues, and security limitations inherent in emulated environments.

- Complexity because the process of building and signing the kernel and modules involves intricate knowledge of kernel development and cryptographic operations, posing a significant challenge.
- Performance Overhead due to that Signature verification introduces computational overhead, potentially degrading system performance.

Future:

- Further study and research in the implementation of this project on an actual raspberry pi and trying to get into more details of the steps and the micro steps of this application.
- More research into developing projects on emulation environments so it's possible to widen the aspects of the application choices.
- Approaching and testing more powerful hardware devices to test the application of this idea in full private compilation or a better performance of cross compilation.
- Familiarization of other compilation methods and the related applications.
- Working on enhancing the level of cryptography and applying more forms of authentication and verification.

5. Conclusion:

To summarize, the process of building a signed kernel and dynamically loading kernel modules signed with a corresponding private key and certificate were expressed. The integrity and authenticity of the modules were ensured through cryptographic verification performed by the kernel, preventing the loading of unsigned or invalidly signed modules. This approach was then applied to an embedded hardware device, specifically a Raspberry Pi board, considering all necessary methods and approaches such as cross-compilation, secure transfer of compiled files, and configuration of the bootloader for secure boot. The findings demonstrate that employing cryptographic signatures significantly enhances system security by ensuring only verified modules are loaded. The implementation on a Raspberry Pi confirmed the feasibility and effectiveness of these methods in an embedded system environment, maintaining system integrity and reliability.

6. References:

1. <https://www.kernel.org>
2. <https://www.kernel.org/doc/html/latest/admin-guide/module-signing.html>
3. <https://tldp.org/LDP/lkmpg/2.6/html/>
4. <https://lkml.org>
5. <https://www.raspberrypi.com/documentation/>
6. https://www.raspberrypi.com/documentation/computers/linux_kernel.html
7. https://elinux.org/Main_Page
8. <https://www.kernel.org/doc/html/v4.15/admin-guide/module-signing.html>