# Trusted Boot Module

User Manual

**white**box

## Contents

# 1 Introduction

This document covers the installation and configuration of the *Trusted Boot Module* (TBM) and the *Read-Only Trusted System* (ROTS). The TBM is an additional board that consists of a *microcontroller unit* or MCU to manage the boot procedure of the host device in a secure fashion by managing keys, logs and other files related to trusted boot management. Furthermore, the host device will be restricted to only boot from one read-only storage device that will contain a trusted image or the *Read-Only Trusted System* (ROTS). Once this image has been booted, the hosted device is in a trusted state from which it will be able to execute a minimal software stack to enumerate the images to boot, to verify these images and to select what image to boot. Once the image has been booted, the host device will enter an untrusted stage and the TBM will only allow for restricted access. This implementation allows the host device to only boot images that are trusted and prevents attackers from tampering with the host device or the TBM to boot untrusted images as long as they don't have physical access and as long as there are no vulnerabilities.

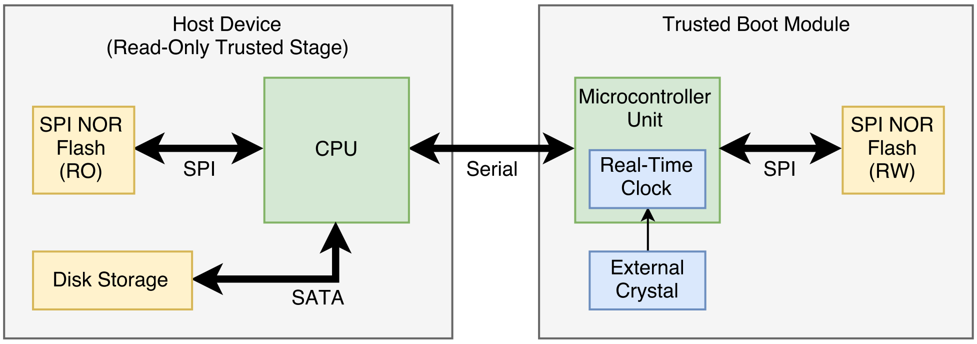

Figure 1: a high-level overview of the interaction between the host device and the Trusted Boot Module

Figure 1 shows a high-level overview of the design. Once the device receives power the *Trusted Boot Module* will boot and at some point the TBM will power on the host device. The host device will then read the trusted image from the SPI NOR flash. Because the device has been configured to be restricted to boot from the SPI NOR flash and because the SPI NOR flash has been configured to be read-only, the host device will be in a trusted state. The image that has been booted is designed to be minimal and only contains the software necessary to perform the boot procedure. Furthermore, the image does not contain a network stack to reduce the amount of possible vulnerabilities and thus to minimise the attack vector. Once the trusted image has been booted, the host device will enumerate the images to boot and co-operate with the TBM to verify images and to select the image to boot. This co-operation happens by means of serial communication with the TBM, where the TBM will grant access to the key storage to the ROTS. Once an image has been selected to boot, the ROTS will inform the TBM that it will boot this image and enter the untrusted stage. From there on the TBM will only allow for restricted access.

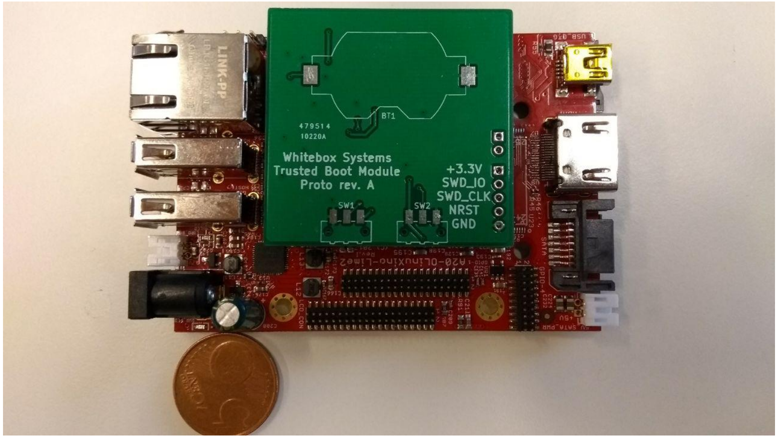

Figure 2: the Trusted Boot Module

# 2 Building ROTS

## 2.1 u-boot

At the moment of writing, the mainline version of u-boot does not have support for SPI NOR flash on Allwinner SoCs such as the Allwinner A10, A20 and the A64. A driver model compatible SPI driver for u-boot is has been worked on and the code can be found at `https://github.com/StephanvanSchaik/u-boot/tree/sunxi-spi`. This driver has been tested on the following boards:

- H2+ Orange Pi Zero with Macronix MX25L1605D 16 Mbit
- A20 OLinuXino LIME 2 with Winbond W25Q128BV 128 Mbit
- A64 Pine64+ with Winbond W25Q128BV 128 Mbit
- A64 OLinuXino with Eon EN25Q64 64 Mbit

To compile u-boot with support for SPI NOR flash:

```
git clone https://github.com/StephanvanSchaik/u-boot -b sunxi-spi
make clean
make A20-OLinuXino-Lime2_defconfig
CROSS_COMPILE=armv7a-hardfloat-linux-gnueabi- make
```

After u-boot-sunxi-with-spl.bin has been built, we can put it on an SD card as follows to test it:

```
dd if=u-boot-sunxi-with-spl.bin of=/dev/sda bs=1024 seek=8
```

While U-boot also supports booting from SPI NOR flash, it has been disabled by default:

```
make menuconfig
```

Enable the `CONFIG_SPL_SPI_SUNXI` option. It is possible that the resulting binary will be too large. In that case, an option like `CONFIG_SPL_MMC_SUPPORT` can be disabled to save some space. After the configuration options have been set up, rebuild the u-boot binary again.

## 2.2 Linux kernel

Make sure that the following options are enabled:

- `CONFIG_BLK_DEV_INITRD`
- `CONFIG_RD_GZIP`
- `CONFIG_RD_BZIP2`
- `CONFIG_RD_LZMA`

- CONFIG_RD_XZ
- CONFIG_RD_LZO
- CONFIG_RD_LZ4
- CONFIG_KEXEC

As the ROTS image will be read-only once it has been flashed to the SPI NOR flash, it is encouraged to build a minimal kernel images to reduce the amount of possible bugs and vulnerabilities. More specifically, it is recommended to build a kernel without any support for networking, graphics and audio.

## 2.3 initramfs

For the initramfs, we will need static binaries of *busybox*, *kexec-tools*, *cpio* and *gzip*.

Run the following command to pack up the directory structure of the initramfs into a gzip compressed cpio archive:

```
find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../initramfs.cpio.gz
```

Because u-boot expects the initramfs to be in the u-boot format, we also have to wrap it up in their format using *mkimage*:

```
mkimage -n 'Ramdisk Image' -A arm -O linux -T ramdisk -C gzip -d initramfs.cpio.gz
 ↪  initramfs.uImage
```

# 3 Flashing ROTS

To write the ROTS image to the SPI NOR flash, we can either use an external programmer or boot the device in FEL mode and use the *sunxi-fel* tool. To write the images, it is strongly recommended to use the *sunxi-fel* tool as it is much faster than using an external programmer. However, to configure the write-protection an external programmer must be used, as there are few tools that support configuring the write-protection of the SPI NOR flash.

## 3.1 Using an External Programmer

In order to be able to program the SPI NOR flash with an external programmer, we will need an external programmer such as the BusPirate v3.6a or the BusPirate v4.0 and SOIC clip. Figure 3 illustrates the pin-out of a Winbond W25Q128.V SPI NOR flash, but any SPI NOR flash chip should be compatible with this pin-out. The SPI NOR flash should have a circular shape at one of the corners, this corner should be bottom-right corner. Once the pins of the SPI NOR flash are aligned with the pin-out in figure 3, we can clip the SPI NOR flash chip between the SOIC clip.

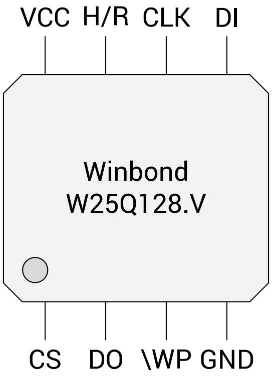

Figure 3: the pin-out of the Winbond W25Q128.V SPI NOR flash

Figure 4 shows how to connect the BusPirate v3.6a with the SPI NOR flash chip. Connect the *Chip Select* (CS) pins using the white cable, the *Master In Slave Out* (MISO) pin with the *Data Out* (DO) pin using the black cable, the *Master Out Slave In* (MOSI) pin with the *Data In* (DI) pin using the grey cable and the *Clock* (CLK) pins using the purple cable. Further, the *Ground* (GND) pins should be connected using the brown cable and the 5V and the VCC pins should be connected with the orange cable. In order for the SPI NOR flash chip to function, the H/R pin of the SPI NOR flash chip should be pulled high, this can be done by connecting the 5V pin with the H/R pin. Finally, to be able to program the chip in case write-protection has been configured before, we have to make sure that the *Write-Protect* (WP) is pulled high to disable write-protection.

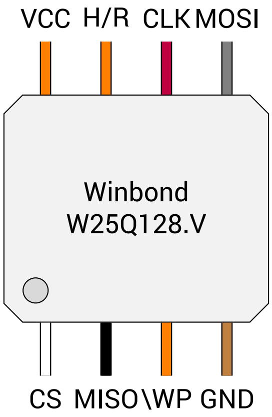

Figure 4: connecting the BusPirate v3.6a with the SPI NOR Flash

Because the configuration of write-protection is vendor-specific, the mainline version of *flashrom* does not support configuring write-protection. Therefore, to be able to configure the write-protection of the SPI NOR flash chip, we have to use Google's fork of *flashrom*.

Unlike the mainline version of flashrom, Google's fork has two flags to get the name and the size of the Flash chip:

```
./flashrom --programmer=buspirate_spi:dev=/dev/buspirate --flash-name
flashrom v0.9.4  : bc6cab1 : Oct 30 2014 07:32:01 UTC on Linux 4.9.4-gentoo (x86_64), built
↪  with libpci 3.1.10, GCC 4.8.x-google 20140307 (prerelease), little endian
vendor="Macronix" name="MX25L6406E"
./flashrom --programmer=buspirate_spi:dev=/dev/buspirate --get-size
flashrom v0.9.4  : bc6cab1 : Oct 30 2014 07:32:01 UTC on Linux 4.9.4-gentoo (x86_64), built
↪  with libpci 3.1.10, GCC 4.8.x-google 20140307 (prerelease), little endian
16777216
```

Further, Google's fork of flashrom allows us to tag regions on the SPI NOR flash chip with a custom name. Assuming that the SPI NOR flash chip is 16 MiB, we will be using the following `layout.txt` file for the ROTS:

```
000000:09ffff uboot
0a0000:5fffff linux
600000:ffffff initramfs
```

We can then write `u-boot.bin`, `bzImage` and `initramfs.cpio.gz` to the SPI NOR flash chip by using the respective names of the regions. To speed up the process of writing these images, we have to disable parsing the fmap and the verification of unmodified regions. Furthermore, to maintain an optimal stability, an SPI speed of no more than 2 MHz is recommended when using the BusPirate v3.6a:

```
./flashrom --programmer=buspirate_spi:spispeed=2M,dev=/dev/buspirate -l layout.txt -i
↪  uboot:u-boot.bin linux:bzImage initramfs:initramfs.cpio.gz -w --ignore-fmap
↪  --fast-verify
```

Now that the images have been written to their respective regions, we can look at the write-protect ranges supported by the chip:

```
./flashrom --programmer=buspirate_spi:dev=/dev/buspirate --wp-list
flashrom v0.9.4  : bc6cab1 : Oct 30 2014 07:32:01 UTC on Linux 4.9.4-gentoo (x86_64), built
 ↪  with libpci 3.1.10, GCC 4.8.x-google 20140307 (prerelease), little endian
Valid write protection ranges:
start: 0x000000, length: 0x000000
start: 0xfc0000, length: 0x040000
start: 0xf80000, length: 0x080000
start: 0xf00000, length: 0x100000
start: 0xe00000, length: 0x200000
start: 0xc00000, length: 0x400000
start: 0x800000, length: 0x800000
start: 0x000000, length: 0x040000
start: 0x000000, length: 0x080000
start: 0x000000, length: 0x100000
start: 0x000000, length: 0x200000
start: 0x000000, length: 0x400000
start: 0x000000, length: 0x800000
start: 0x000000, length: 0x1000000
start: 0xfff000, length: 0x001000
start: 0xffe000, length: 0x002000
start: 0xffc000, length: 0x004000
start: 0xff8000, length: 0x008000
start: 0xff8000, length: 0x008000
start: 0x000000, length: 0x001000
start: 0x000000, length: 0x002000
start: 0x000000, length: 0x004000
start: 0x000000, length: 0x008000
start: 0x000000, length: 0x008000
```

Since we don't want our images to be tampered with, we want to enable write-protection for the full range. We can configure the write-protected range as follows:

```
./flashrom --programmer=buspirate_spi:spispeed=2M,dev=/dev/buspirate --wp-range 0x000000
 ↪  0x1000000
```

After setting the range, we are still able to modify the contents of the entire SPI NOR flash chip. To protect the range, we have to enable write protection as follows:

```
./flashrom --programmer=buspirate_spi:spispeed=2M,dev=/dev/buspirate --wp-enable
```

Upon enabling write-protection, the *Write-Protect* (WP) pin has to be pulled low for the write-protection to be effective. This prevents the user from disabling the write-protection feature, changing the write-protect range and from writing to the write-protected region.

## 3.2 Using sunxi-fel

Download and compile the *sunxi-fel* tool as follows:

```
git clone -b spiflash-a20-test https://github.com/ssvb/sunxi-tools.git
make
```

Connect or reset while holding the recovery or FEL button. Once the board has booted into FEL mode, we can detect the SPI NOR flash chip as follows:

```
./sunxi-fel spiflash-info
Manufacturer: Winbond (EFh), model: 40h, size: 16777216 bytes.
```

Then we can write the `u-boot.bin`, `bzImage` and `initramfs.uImage` images as follows:

```
./sunxi-fel -p spiflash-write 0x000000 u-boot.bin
./sunxi-fel -p spiflash-write 0x0a0000 bzImage
./sunxi-fel -p spiflash-write 0x600000 initramfs.uImage
```

# 4 Booting ROTS

After powering up the board, *u-boot* will be loaded. *u-boot* will then load the Linux kernel image and the initramfs from the SPI NOR flash and boot the Linux kernel with the initramfs as follows:

```
sf probe 0:0 6000000
sf read 0x42000000 0xa0000 5636096
sf read 0x43000000 0x600000 10485760
bootm 0x42000000 0x43000000
```

The ROTS kernel will now boot up and mount the initramfs as the rootfs. At some point the kernel will run the init script in the initramfs. When this happens the ROTS will start communicating with the TBM to fetch the time as well as the certificates. Once these have been retrieved from the TBM, the ROTS will mount the external media such as hard disks and enumerate and verify possible boot images on those media.

# 5 Trusted Boot Module

In this section the process of building, flashing and configuring the firmware for the TBM is described.

## 5.1 Building

A cross-compiler targetting ARMv6 or ARMv7-M such as the GNU ARM Embedded Toolchain is required to build the source code for the Trusted Boot Module. Either install it using your package manager or download the toolchain from https://developer.arm.com/open-source/gnu-toolchain/gnu-rm. For Gentoo users, an ebuild is available in the tbm-overlay repository. Building your own cross-compiler using a tool like crossdev can be quite tricky, and is therefore discouraged.

To build the source code in this repository, you will also need libopencm3. Download the source code for libopencm3 and build it as follows:

```
git submodule init
git submodule update
make -C libopencm3
```

To build the code for the Trusted Boot Module, run:

```
TARGET=stm32f1 make
```

## 5.2 Flashing

Connect your computer with a JTAG device and connect the I/O, CLK, GND and VDD wires with the SWD pin-out of the TBM. Run the following as root:

```
TARGET=stm32f1 make openocd
```

If OpenOCD, the JTAG device and the TBM are functioning correctly, OpenOCD should report the available breakpoint registers. To flash the firmware onto the device, run the following command:

```
TARGET=stm32f1 make run
```

Close OpenOCD and disconnect the power from the TBM, then detach the I/O and CLK wires.

## 5.3 Configuration

Download the source code for gorots and build the admin utility as follows:

```
go build
```

Attach a serial cable to the debugging serial interface of the TBM. Power on the TBM using an external power source such as a JTAG adapter. Run the `prepare.sh` script to perform an initial configuration of the TBM. This script will also perform tests to ensure the hardware is functioning as expected.

# 6 Using gorots

The gorots package consists of two tools: `admin` and `protocol`. The `protocol` utility is used by the ROTS to communicate with the user serial interface. The `admin` utility is used with the admin serial interface of the TBM to configure it.

The following commands are available for the `admin` utility:

- `./admin echo <string>`

  Returns the string passed as an argument to echo.
- `./admin flash_probe`

  Mounts the SPI NOR flash device as a raw device.
- `./admin flash_erase <offset> <length>`

  Erases the region provided by the offset and the length in blocks.
- `./admin ftl_probe`

  Mounts the SPI NOR flash device and initialises the Flash Translation Layer.
- `./admin date`

  Returns the current time formatted as a human-readable date.
- `./admin time`

  Returns the current time in seconds since the UNIX epoch.
- `./admin set-time <seconds>`

  Sets the current time to the given time in seconds since the UNIX epoch.
- `./admin sync-time`

  Synchronizes the current time of the TBM with the current time of the host device
- `./admin set-time <seconds>`

  Sets the current time to the given time in seconds since the UNIX epoch.
- `./admin sync-time`

  Synchronizes the current time of the TBM with the current time of the host device.
- `./admin mount`

  Mounts the filesystem on the flash device.
- `./admin umount`

  Unmounts the currently mounted filesystem.
- `./admin format`

  Formats the filesystem of the flash device.
- `./admin mkdir <path>`

  Creates a directory at the given path, if the path does not yet exist and if the parent is a directory.
- `./admin rmdir <path>`

  Removes the directory at the given path, if the path exists and points to a directory.
- `./admin ls <path>`

  Lists the files in the given path, if the path exists and points to a directory.
- `./admin cat <path>`

  Concatenates the file with the standard output.

- `./admin write <path> <file>`

  Writes the contents of the file to the file at the given path. Creates a new file if the path does not point to an existing file. Otherwise the file will be truncated first.

- `./admin append <path> <file>`

  Appends the contents of the file to the file at the given path. Creates a new file if the path does not point to an existing file.

- `./admin mv <old> <new>`

  Moves the file or directory from the old path to the new path.

- `./admin cp <old> <new>`

  Copies the contents of the file from the old path to the file pointed to by the new path. Creates a file at the new path if it does not exist yet. Otherwise the file is truncated before copying the contents.

- `./admin rm <path>`

  Removes the file at the given path, if the path exists and points to a file.