wolfBoot Documentation

2024-03-13

Contents

1 Introduction

wolfBoot is a portable, OS-agnostic, secure bootloader solution for 32-bit microcontrollers, relying on wolfCrypt for firmware authentication, providing firmware update mechanisms.

Due to the minimalist design of the bootloader and the tiny HAL API, wolfBoot is completely independent from any OS or bare-metal application, and can be easily ported and integrated in existing embedded software projects to provide a secure firmware update mechanism.

Features

- Multi-slot partitioning of the flash device
- Integrity verification of the firmware image(s)
- Authenticity verification of the firmware image(s) using wolfCrypt's Digital Signature Algorithms (DSA)
- Minimalist hardware abstraction layer (HAL) interface to facilitate portability across different vendors/MCUs
- Copy/swap images from secondary slots into the primary slots to consent firmware update operations
- In-place chain-loading of the firmware image in the primary slot
- Support of Trusted Platform Module(TPM)
- Measured boot support, storing of the firmware image hash into a TPM Platform Configuration Register(PCR)

Components

The wolfBoot Github repository contains the following components:

- the wolfBoot bootloader
- key generator and image signing tools (requires python 3.x and wolfcrypt-py https://github.com/wolfSSL/wolfcry
  py)
- Baremetal test applications

2 Compiling wolfBoot

WolfBoot is portable across different types of embedded systems. The platform-specific code is contained in a single file under the `hal` directory, and implements the hardware-specific functions.

To enable specific compile options, use environment variables while calling make, e.g.

```
make CORTEX_M0=1
```

As an alternative, you can provide a .config file in the root directory of wolfBoot. Command line options have priority on `.config` options, as long as .config options are defined using the `?=` operator, e.g.:

```
WOLFBOOT_PARTITION_BOOT_ADDRESS?=0x14000
```

2.1 Generate a new configuration

A new `.config` file with a set of default parameters can be generated by running `make config`. The build script will ask to enter a default value for each configuration parameter. Enter confirm the current value, indicated in between `[]`.

Once a .config file is in place, it will change the default compile-time options when running `make` without parameters.

.config can be modified with a text editor to alter the default options later on.

2.2 Platform selection

If supported natively, the target platform can be specified using the `TARGET` variable. Make will automatically select the correct compile option, and include the corresponding HAL for the selected target.

For a list of the platforms currently supported, see the chapter on HAL.

To add a new platform, simply create the corresponding HAL driver and linker script file in the `hal` directory.

Default option if none specified: `TARGET=stm32f4`

Some platforms will require extra options, specific for the architecture. By default, wolfBoot is compiled for ARM Cortex-M3/4/7. To compile for Cortex-M0, use:

```
CORTEX_M0=1
```

2.2.1 Flash partitions

The file `include/target.h` is generated according to the configured flash geometry, partitions size and offset of the target system. The following values must be set to provide the desired flash configuration, either via the command line, or using the .config file:

- `WOLFBOOT_SECTOR_SIZE`

This variable determines the size of the physical sector on the flash memory. If areas with different block sizes are used for the two partitions (e.g. update partition on an external flash), this variable should indicate the size of the biggest sector shared between the two partitions.

WolfBoot uses this value as minimum unit when swapping the firmware images in place. For this reason, this value is also used to set the size of the SWAP partition.

- `WOLFBOOT_PARTITION_BOOT_ADDRESS`

This is the start address of the boot partition, aligned to the beginning of a new flash sector. The application code starts after a further offset, equal to the partition header size (256B for Ed25519 and ECC signature headers).

- `WOLFBOOT_PARTITION_UPDATE_ADDRESS`

This is the start address of the update partition. If an external memory is used via the `EXT_FLASH` option, this variable contains the offset of the update partition from the beginning of the external memory addressable space.

- `WOLFBOOT_PARTITION_SWAP_ADDRESS`

The address for the swap spaced used by wolfBoot to swap the two firmware images in place, in order to perform a reversable update. The size of the SWAP partition is exactly one sector on the flash. If an external memory is used, the variable contains the offset of the SWAP area from the beginning of its addressable space.

- `WOLFBOOT_PARTITION_SIZE`

The size of the BOOT and UPDATE partition. The size is the same for both partitions.

2.3 Bootloader features

A number of characteristics can be turned on/off during wolfBoot compilation. Bootloader size, performance and activated features are affected by compile-time flags.

2.3.1 Change DSA algorithm

By default, wolfBoot is compiled to use Ed25519 DSA. The implementation of ed25519 is smaller, while giving a good compromise in terms of boot-up time.

Better performance can be achieved using ECDSA with curve p-256. To activate ECC256 support, use

`SIGN=ECC256`

when invoking `make`.

RSA is also supported, with different key length. To activate RSA2048 or RSA4096, use:

`SIGN=RSA2048`

or

`SIGN=RSA4096`

respectively.

Ed448 is also supported via `SIGN=ED448`.

The default option, if no value is provided for the `SIGN` variable, is

`SIGN=ED25519`

Changing the DSA algorithm will also result in compiling a different set of tools for key generation and firmware signature.

Find the corresponding key generation and firmware signing tools in the `tools` directory.

It's possible to disable authentication of the firmware image by explicitly using:

`SIGN=NONE`

in the Makefile commandline. This will compile a minimal bootloader with no support for public-key authenticated secure boot.

2.3.2 Incremental updates

wolfBoot support incremental updates. To enable this feature, compile with `DELTA_UPDATES=1`.

An additional file is generated when the sign tool is invoked with the `--delta` option, containing only the differences between the old firmware to replace, currently running on the target, and the new version.

For more information and examples, see the firmware update section.

2.3.3 Enable debug symbols

To debug the bootloader, simply compile with `DEBUG=1`. The size of the bootloade will increase consistently, so ensure that you have enough space at the beginning of the flash before `WOLFBOOT_PARTITION_BOOT_ADDRESS`.

2.3.4 Disable interrupt vector relocation

On some platforms, it might be convenient to avoid the interrupt vector relocation before boot-up. This is required when a component on the system already manages the interrupt relocation at a different stage, or on these platform that do not support interrupt vector relocation.

To disable interrupt vector table relocation, compile with `VTOR=0`. By default, wolfBoot will relocate the interrupt vector by setting the offset in the vector relocation offset register (VTOR).

2.3.5 Limit stack usage

By default, wolfBoot does not require any memory allocation. It does this by performing all the operations using the stack. Although the stack space used by the algorithms can be predicted at compile time, the amount of stack space be relatively big, depending on the algorithm selected.

Some targets offer limited amount of RAM to use as stack space, either in general, or in a configuration dedicated for the bootloader stage.

In these cases, it might be useful to activate `WOLFBOOT_SMALL_STACK=1`. With this option, a fixed-size pool is created at compile time to assist the allocation of the object needed by the cryptography implementation. When compiled with `WOLFBOOT_SMALL_STACK=1`, wolfBoot reduces the stack usage considerably, and simulates dynamic memory allocations by assigning dedicated, statically allocated, pre-sized memory areas.

2.3.6 Disable Backup of current running firmware

Optionally, it is possible to disable the backup copy of the current running firmware upon the installation of the update. This implies that no fall-back mechanism is protecting the target from a faulty firmware installation, but may be useful in some cases where it is not possible to write on the update partition from the bootloader. The associated compile-time option is

```
DISABLE_BACKUP=1
```

2.3.7 Enable workaround for 'write once' flash memories

On some microcontrollers, the internal flash memory does not allow subsequent writes (adding zeroes) to a sector, after the entire sector has been erased. WolfBoot relies on the mechanism of adding zeroes to the 'flags' fields at the end of both partitions to provide a fail-safe swap mechanism.

To enable the workaround for 'write once' internal flash, compile with

```
NVM_FLASH_WRITEONCE=1
```


warning When this option is enabled, the fail-safe swap is not guaranteed, i.e. the microcontroller cannot be safely powered down or restarted during a swap operation.

2.3.8 Allow version roll-back

WolfBoot will not allow updates to a firmware with a version number smaller than the current one. To allow downgrades, compile with `ALLOW_DOWNGRADE=1`.

Warning: this option will disable version checking before the updates, thus exposing the system to potential forced downgrade attacks.

2.3.9 Enable optional support for external flash memory

WolfBoot can be compiled with the makefile option `EXT_FLASH=1`. When the external flash support is enabled, update and swap partitions can be associated to an external memory, and will use alternative HAL function for read/write/erase access. To associate the update or the swap partition to an external memory, define `PART_UPDATE_EXT` and/or `PART_SWAP_EXT`, respectively. By default, the makefile assumes that if an external memory is present, both `PART_UPDATE_EXT` and `PART_SWAP_EXT` are defined.

If the `NO_XIP=1` makefile option is present, `PART_BOOT_EXT` is assumed too, as no execute-in-place is available on the system. This is typically the case of MMU system (e.g. Cortex-A) where the operating system image(s) are position-independent ELF images stored in a non-executable non-volatile memory, and must be copied in RAM to boot after verification.

When external memory is used, the HAL API must be extended to define methods to access the custom memory. Refer to the HAL chapter for the description of the `ext_flash_*` API.

2.3.9.1 SPI devices In combination with the `EXT_FLASH=1` configuration parameter, it is possible to use a platform-specific SPI drivers, e.g. to access an external SPI flash memory. By compiling wolfBoot with the makefile option `SPI_FLASH=1`, the external memory is directly mapped to the additional SPI layer, so the user does not have to define the `ext_flash_*` functions.

SPI functions, instead, must be defined. Example SPI drivers are available for multiple platforms in the `hal/spi` directory.

2.3.9.2 UART bridge towards neighbor systems Another alternative available to map external devices consists in enabling a UART bridge towards a neighbor system. The neighbor system must expose a service through the UART interface that is compatible with the wolfBoot protocol.

In the same way as for SPI devices, the `ext_flash_*` API is automatically defined by wolfBoot when the option `UART_FLASH=1` is used.

For more details, see the section Remote External flash memory support via UART

2.3.9.3 Encryption support for external partitions When update and swap partitions are mapped to an external device using `EXT_FLASH=1`, either in combination with `SPI_FLASH`, `UART_FLASH`, or any custom external mapping, it is possible to enable ChaCha20, Aes128 or Aes256 encryption when accessing those partition from the bootloader. The update images must be pre-encrypted at the source using the key tools, and wolfBoot should be instructed to use a temporary ChaCha20 symmetric key to access the content of the updates.

For more details about this optional feature, please refer to the Encrypted external partitions section.

2.3.10 Executing flash access code from RAM

On some platform, flash access code requires to be executed from RAM, to avoid conflict e.g. when writing to the same device where wolfBoot is executing, or when changing the configuration of the flash itself.

To move all the code accessing the internal flash for writing, into a section in RAM, use the compile time option `RAM_CODE=1` (on some hardware configurations this is required for the bootloader to access the flash for writing).

2.3.11 Enable Dual-bank hardware-assisted swapping

When supported by the target platform, hardware-assisted dual-bank swapping can be used to perform updates. To enable this functionality, use `DUALBANK_SWAP=1`. Currently, only STM32F76x and F77x support this feature.

2.3.12 Store UPDATE partition flags in a sector in the BOOT partition

By default, wolfBoot keeps track of the status of the update procedure to the single sectors in a specific area at the end of each partition, dedicated to store and retrieve a set of flags associated to the partition itself.

In some cases it might be helpful to store the status flags related to the UPDATE partition and its sectors in the internal flash, alongside with the same set of flags used for the BOOT partition. By compiling wolfBoot with the `FLAGS_HOME=1` makefile option, the flags associated to the UPDATE partition are stored in the BOOT partition itself.

While on one hand this option slightly reduces the space available in the BOOT partition to store the firmware image, it keeps all the flags in the BOOT partition.

2.3.13 Invert logic of flags

By default, most NVMs set the content of erased pages to `0xFF` (all ones). Some FLASH memory models use inverted logic for erased page, setting the content to `0x00` (all zeroes) after erase. For these special cases, the option `FLAGS_INVERT = 1` can be used to modify the logic of the partition/sector flags used in wolfBoot.

Note: if you are using an external FLASH (e.g. SPI) in combination with a flash with inverted logic, ensure that you store all the flags in one partition, by using the `FLAGS_HOME=1` option described above.

2.3.14 Using Mac OS/X

If you see 0xC3 0xBF (C3BF) repeated in your factory.bin then your OS is using Unicode characters.

The “tr” command for assembling the 0xFF padding between `"bootloader" ... 0xFF ... "application" = factory.bin`, which requires the “C” locale.

Set this in your terminal

```
LANG=
LC_COLLATE="C"
LC_CTYPE="C"
LC_MESSAGES="C"
LC_MONETARY="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_ALL=
```

Then run the normal `make` steps.

2.3.15 Enabling mitigations against glitches and fault injections

One type of attacks against secure boot mechanisms consists in skipping the execution of authentication and validation steps by injecting faults into the CPU through forced voltage or clock anomalies, or electromagnetic interferences at close range.

Extra protection from specific attacks aimed to skip CPU instructions can be enabled using `ARMOR=1`. This feature is currently only available for ARM Cortex-M targets.

3 Targets

This chapter describes configuration of supported targets.

3.1 Supported Targets

- Cortex-A53 / Raspberry PI 3
- Cypress PSoC-6
- Nordic nRF52840
- NXP LPC54xxx
- NXP iMX-RT
- NXP Kinetis
- NXP T2080 PPC
- SiFive HiFive1 RISC-V
- STM32F4
- STM32L4
- STM32F7
- STM32G0
- STM32H7
- STM32L5
- STM32U5
- STM32L0
- STM32WB55
- TI Hercules TMS570LC435
- Xilinx Zynq UltraScale
- Qemu x86_64 UEFI

3.2 STM32F4

Example 512KB partitioning on STM32-F407

The example firmware provided in the `test-app` is configured to boot from the primary partition starting at address 0x20000. The flash layout is provided by the default example using the following configuration in `target.h`:

```
#define WOLFBOOT_SECTOR_SIZE                0x20000
#define WOLFBOOT_PARTITION_SIZE             0x20000

#define WOLFBOOT_PARTITION_BOOT_ADDRESS   0x20000
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS 0x40000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS   0x60000
```

This results in the following partition configuration:

This configuration demonstrates one of the possible layouts, with the slots aligned to the beginning of the physical sector on the flash.

The entry point for all the runnable firmware images on this target will be `0x20100`, 256 Bytes after the beginning of the first flash partition. This is due to the presence of the firmware image header at the beginning of the partition, as explained more in details in Firmware image

In this particular case, due to the flash geometry, the swap space must be as big as 128KB, to account for proper sector swapping between the two images.

On other systems, the SWAP space can be as small as 512B, if multiple smaller flash blocks are used.

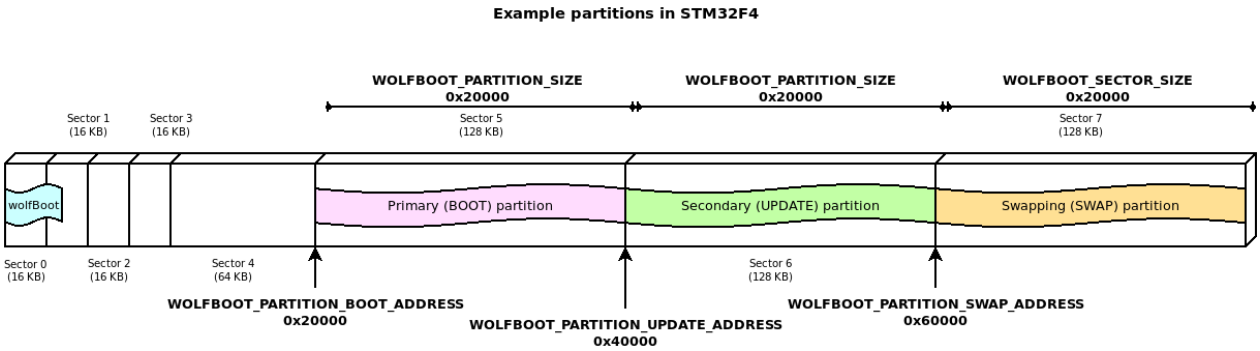

Figure 1: example partitions

More information about the geometry of the flash and in-application programming (IAP) can be found in the manufacturer manual of each target device.

3.2.1 STM32F4 Programming

```
st-flash write factory.bin 0x08000000
```

3.2.2 STM32F4 Debugging

1. Start GDB server

OpenOCD: `openocd --file ./config/openocd/openocd_stm32f4.cfg` OR ST-Link: `st-util -p 3333`

2. Start GDB Client

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf 0x20100
mon reset init
b main
c
```

3.3 STM32L4

Example 1MB partitioning on STM32L4

- Sector size: 4KB
- Wolfboot partition size: 40 KB
- Application partition size: 488 KB

```
#define WOLFBOOT_SECTOR_SIZE                0x1000   /* 4 KB */
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x0800A000
#define WOLFBOOT_PARTITION_SIZE             0x7A000  /* 488 KB */
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x08084000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x080FE000
```

3.4 STM32L5

3.4.1 Scenario 1: TrustZone Enabled

3.4.1.1 Example Description The implementation shows how to switch from secure application to non-secure application, thanks to the system isolation performed, which splits the internal Flash and

internal SRAM memories into two halves: - the first half for secure application - the second half for non-secure application

3.4.1.2 Hardware and Software environment

- This example runs on STM32L562QEIxQ devices with security enabled (TZEN=1).
- This example has been tested with STMicroelectronics STM32L562E-DK (MB1373)
- User Option Bytes requirement (with STM32CubeProgrammer tool - see below for instructions)

```
TZEN = 1                              System with TrustZone-M enabled
DBANK = 1                             Dual bank mode
SECWM1_PSTRT=0x0  SECWM1_PEND=0x7F  All 128 pages of internal Flash Bank1 set
   as secure
SECWM2_PSTRT=0x1  SECWM2_PEND=0x0   No page of internal Flash Bank2 set as
   secure, hence Bank2 non-secure
```

- NOTE: STM32CubeProgrammer V2.3.0 is required (v2.4.0 has a known bug for STM32L5)

3.4.1.3 How to use it

1. `cp ./config/examples/stm32l5.config .config`
2. `make TZEN=1`
3. Prepare board with option bytes configuration reported above
   - `STM32_Programmer_CLI -c port=swd mode=hotplug -ob TZEN=1 DBANK=1`
   - `STM32_Programmer_CLI -c port=swd mode=hotplug -ob SECWM1_PSTRT=0x0 SECWM1_PEND=0x7F SECWM2_PSTRT=0x1 SECWM2_PEND=0x0`
4. flash `wolfBoot.bin` to 0x0c00 0000
   - `STM32_Programmer_CLI -c port=swd -d ./wolfboot.bin 0x0C000000`
5. flash `.\test-app\image_v1_signed.bin` to 0x0804 0000
   - `STM32_Programmer_CLI -c port=swd -d ./test-app/image_v1_signed.bin 0x08040000`
6. RED LD9 will be on

- NOTE: STM32_Programmer_CLI Default Locations
- Windows: `C:\Program Files\STMicroelectronics\STM32Cube\STM32CubeProgrammer\bin\STM32_Pro`
- Linux: `/usr/local/STMicroelectronics/STM32Cube/STM32CubeProgrammer/bin/STM32_Programmer_`
- Mac OS/X: `/Applications/STMicroelectronics/STM32Cube/STM32CubeProgrammer/STM32CubeProgra`

3.4.2 Scenario 2: Trustzone Disabled

3.4.2.1 Example Description The implementation shows how to use STM32L5xx in DUAL_BANK mode, with TrustZone disabled. The DUAL_BANK option is only available on this target when TrustZone is disabled (TZEN = 0).

The flash memory is segmented into two different banks:

- Bank 0: (0x08000000)
- Bank 1: (0x08040000)

Bank 0 contains the bootloader at address 0x08000000, and the application at address 0x08040000. When a valid image is available at the same offset in Bank 1, a candidate is selected for booting between the two valid images. A firmware update can be uploaded at address 0x08048000.

The example configuration is available in `/config/examples/stm32l5-nonsecure-dualbank.config`.

To run flash `./test-app/image.bin` to `0x08000000`. - `STM32_Programmer_CLI -c port=swd -d ./test-app/image.bin 0x08000000`

Or program each partition using: 1. flash `wolfboot.bin` to 0x08000000: - `STM32_Programmer_CLI -c port=swd -d ./wolfboot.elf` 2. flash wolfBoot.bin to 0x0c00 0000 - `STM32_Programmer_CLI -c port=swd -d ./test-app/image_v1_signed.bin 0x08008000`

RED LD9 will be on indicating successful boot ()

3.4.3 Debugging

Use `make DEBUG=1` and reload firmware.

- STM32CubeIDE v.1.3.0 required
- Run the debugger via:

Linux:

```
ST-LINK_gdbserver -d -cp /opt/st/stm32cubeide_1.3.0/plugins/com.st.stm32cube.
    ide.mcu.externaltools.cubeprogrammer.linux64_1.3.0.202002181050/tools/bin -
    e -r 1 -p 3333`
```

Max OS/X:

```
sudo ln -s /Application-
 ↪  s/STM32CubeIDE.app/Contents/Eclipse/plugins/com.st.stm32cube.ide.mcu.externaltools.stli
 ↪  gdb-
 ↪  server.macos64_1.6.0.202101291314/tools/bin/native/mac_x64/libSTLinkUSBDriver.dylib
 ↪  /usr/local/lib/libSTLinkUSBDriver.dylib

/Applications/STM32CubeIDE.app/Contents/Eclipse/plugins/com.st.stm32cube.ide.mcu.externalto
 ↪  gdb-server.macos64_1.6.0.202101291314/tools/bin/ST-LINK_gdbserver -d -cp
 ↪  ./Contents/Eclipse/plugins/-
 ↪  com.st.stm32cube.ide.mcu.externaltools.cubeprogrammer.macos64_1.6.0.202101291314/tools/
 ↪  -e -r 1 -p 3333
```

- Connect with arm-none-eabi-gdb

wolfBoot has a .gdbinit to configure

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf
mon reset init
```

3.5 STM32U5

3.5.1 Scenario 1: TrustZone Enabled

3.5.1.1 Example Description The implementation shows how to switch from secure application to non-secure application, thanks to the system isolation performed, which splits the internal Flash and internal SRAM memories into two halves: - the first half for secure application - the second half for non-secure application

3.5.1.2 Hardware and Software environment

- This example runs on STM32U585AII6Q devices with security enabled (TZEN=1).
- This example has been tested with STMicroelectronics B-U585I-IOT02A (MB1551)
- User Option Bytes requirement (with STM32CubeProgrammer tool - see below for instructions)

```
TZEN = 1                              System with TrustZone-M enabled
DBANK = 1                             Dual bank mode
SECWM1_PSTRT=0x0  SECWM1_PEND=0x7F  All 128 pages of internal Flash Bank1 set
    as secure
SECWM2_PSTRT=0x1  SECWM2_PEND=0x0   No page of internal Flash Bank2 set as
    secure, hence Bank2 non-secure
```

- NOTE: STM32CubeProgrammer V2.8.0 or newer is required

3.5.1.3 How to use it

1. `cp ./config/examples/stm32u5.config .config`
2. `make TZEN=1`
3. Prepare board with option bytes configuration reported above
   - `STM32_Programmer_CLI -c port=swd mode=hotplug -ob TZEN=1 DBANK=1`
   - `STM32_Programmer_CLI -c port=swd mode=hotplug -ob SECWM1_PSTRT=0x0 SECWM1_PEND=0x7F SECWM2_PSTRT=0x1 SECWM2_PEND=0x0`
4. flash wolfBoot.bin to 0x0c00 0000
   - `STM32_Programmer_CLI -c port=swd -d ./wolfboot.bin 0x0C000000`
5. flash `.\test-app\image_v1_signed.bin` to 0x0804 0000
   - `STM32_Programmer_CLI -c port=swd -d./test-app/image_v1_signed.bin 0x08100000‘`
6. RED LD9 will be on

- NOTE: STM32_Programmer_CLI Default Locations
- Windows: `C:\Program Files\STMicroelectronics\STM32Cube\STM32CubeProgrammer\bin\STM32_Pro`
- Linux: `/usr/local/STMicroelectronics/STM32Cube/STM32CubeProgrammer/bin/STM32_Programmer_`
- Mac OS/X: `/Applications/STMicroelectronics/STM32Cube/STM32CubeProgrammer/STM32CubeProgra`

3.5.2 Scenario 2: TrustZone Disabled

3.5.2.1 Example Description The implementation shows how to use STM32U5xx in DUAL_BANK mode, with TrustZone disabled. The DUAL_BANK option is only available on this target when TrustZone is disabled (TZEN = 0).

The flash memory is segmented into two different banks:

- Bank 0: (0x08000000)
- Bank 1: (0x08100000)

Bank 0 contains the bootloader at address 0x08000000, and the application at address 0x08100000. When a valid image is available at the same offset in Bank 1, a candidate is selected for booting between the two valid images. A firmware update can be uploaded at address 0x08108000.

The example configuration is available in `config/examples/stm32u5-nonsecure-dualbank.config`.

To run flash `./test-app/image.bin` to `0x08000000`. - `STM32_Programmer_CLI -c port=swd -d ./test-app/image.bin 0x08000000`

Or program each partition using: 1. flash `wolfboot.bin` to 0x08000000: - `STM32_Programmer_CLI -c port=swd -d ./wolfboot.elf` 2. flash `image_v1_signed.bin` to 0x08008000 - `STM32_Programmer_CLI -c port=swd -d ./test-app/image_v1_signed.bin 0x08008000`

RED LD9 will be on indicating successful boot ()

3.5.3 Debugging

Use `make DEBUG=1` and reload firmware.

- STM32CubeIDE v.1.7.0 required

- Run the debugger via:

Linux:

```
ST-LINK_gdbserver -d -cp /opt/st/stm32cubeide_1.3.0/plugins/com.st.stm32cube.
    ide.mcu.externaltools.cubeprogrammer.linux64_1.3.0.202002181050/tools/bin -
    e -r 1 -p 3333`
```

Max OS/X:

```
sudo ln -s /Application-
 ↪  s/STM32CubeIDE.app/Contents/Eclipse/plugins/com.st.stm32cube.ide.mcu.externaltools.stli
 ↪  gdb-
 ↪  server.macos64_1.6.0.202101291314/tools/bin/native/mac_x64/libSTLinkUSBDriver.dylib
 ↪  /usr/local/lib/libSTLinkUSBDriver.dylib

/Applications/STM32CubeIDE.app/Contents/Eclipse/plugins/com.st.stm32cube.ide.mcu.externalto
 ↪  gdb-server.macos64_1.6.0.202101291314/tools/bin/ST-LINK_gdbserver -d -cp
 ↪  ./Contents/Eclipse/plugins/-
 ↪  com.st.stm32cube.ide.mcu.externaltools.cubeprogrammer.macos64_1.6.0.202101291314/tools/
 ↪  -e -r 1 -p 3333
```

Win:

```
ST-LINK_gdbserver -d -cp C:\ST\STM32CubeIDE_1.7.0\STM32CubeIDE\plugins\com.st.
    stm32cube.ide.mcu.externaltools.cubeprogrammer.win32_2.0.0.202105311346\
    tools\bin -e -r 1 -p 3333`
```

- Connect with arm-none-eabi-gdb

wolfBoot has a .gdbinit to configure

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf
mon reset init
```

3.6 STM32L0

Example 192KB partitioning on STM32-L073

This device is capable of erasing single flash pages (256B each).

However, we choose to use a logic sector size of 4KB for the swaps, to limit the amount of writes to the swap partition.

The proposed geometry in this example `target.h` uses 32KB for wolfBoot, and two partitions of 64KB each, leaving room for up to 8KB to use for swap (4K are being used here).

```
#define WOLFBOOT_SECTOR_SIZE                0x1000   /* 4 KB */
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x8000
#define WOLFBOOT_PARTITION_SIZE             0x10000 /* 64 KB */
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x18000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x28000
```

3.6.1 STM32L0 Building

Use `make TARGET=stm32l0`. The option `CORTEX_M0` is automatically selected for this target.

3.7 STM32G0

Supports STM32G0x0x0/STM32G0x1.

Example 128KB partitioning on STM32-G070:

- Sector size: 2KB
- Wolfboot partition size: 32KB
- Application partition size: 44 KB

```
#define WOLFBOOT_SECTOR_SIZE                0x800   /* 2 KB */
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x08008000
#define WOLFBOOT_PARTITION_SIZE             0xB000  /* 44 KB */
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x08013000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x0801E000
```

3.7.1 Building STM32G0

Reference configuration (see `/config/examples/stm32g0.config`. You can copy this to wolfBoot root as `.config`: `cp ./config/examples/stm32g0.config .config`. To build you can use `make`.

The TARGET for this is `stm32g0`: `make TARGET=stm32g0`. The option `CORTEX_M0` is automatically selected for this target. The option `NVM_FLASH_WRITEONCE=1` is mandatory on this target, since the IAP driver does not support multiple writes after each erase operation.

This target also supports secure memory protection on the bootloader region using the `FLASH_CR:SEC_PROT` and `FLASH_SECT:SEC_SIZE` registers. This is the number of 2KB pages to block access to from the 0x8000000 base address.

```
STM32_Programmer_CLI -c port=swd mode=hotplug -ob SEC_SIZE=0x10
```

For RAMFUNCTION support (required for SEC_PROT) make sure `RAM_CODE=1`.

Compile requirements: `make TARGET=stm32g0 NVM_FLASH_WRITEONCE=1`

3.7.2 Debugging STM32G0

The output is a single `factory.bin` that includes `wolfboot.bin` and `test-app/image_v1_signed.bin` combined together. This should be programmed to the flash start address `0x08000000`.

Flash using the STM32CubeProgrammer CLI:

```
STM32_Programmer_CLI -c port=swd -d factory.bin 0x08000000
```

3.7.3 STM32G0 Debugging

Use `make DEBUG=1` and program firmware again.

Start GDB server on port 3333:

```
ST-LINK_gdbserver -d -e -r 1 -p 3333
OR
st-util -p 3333
```

wolfBoot has a .gdbinit to configure GDB

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf 0x08008100
mon reset init
```

3.8 STM32WB55

Example partitioning on Nucleo-68 board:

- Sector size: 4KB
- Wolfboot partition size: 32 KB
- Application partition size: 128 KB

```
#define WOLFBOOT_SECTOR_SIZE                0x1000   /* 4 KB */
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x8000
#define WOLFBOOT_PARTITION_SIZE             0x20000 /* 128 KB */
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x28000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x48000
```

3.8.1 STM32WB55 Building

Use `make TARGET=stm32wb`.

The option `NVM_FLASH_WRITEONCE=1` is mandatory on this target, since the IAP driver does not support multiple writes after each erase operation.

Compile with:

```
make TARGET=stm32wb NVM_FLASH_WRITEONCE=1
```

3.8.2 STM32WB55 with OpenOCD

```
openocd --file ./config/openocd/openocd_stm32wbx.cfg
```

```
telnet localhost 4444
reset halt
flash write_image unlock erase factory.bin 0x08000000
flash verify_bank 0 factory.bin
reset
```

3.8.3 STM32WB55 with ST-Link

```
git clone https://github.com/stlink-org/stlink.git
cd stlink
cmake .
make
sudo make install

st-flash write factory.bin 0x08000000

# Start GDB server
st-util -p 3333
```

3.8.4 STM32WB55 Debugging

Use `make DEBUG=1` and reload firmware.

wolfBoot has a .gdbinit to configure

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf 0x08008100
mon reset init
```

3.9 SiFive HiFive1 RISC-V

3.9.1 Features

- E31 RISC-V 320MHz 32-bit processor
- Onboard 16KB scratchpad RAM
- External 4MB QSPI Flash

3.9.2 Default Linker Settings

- FLASH: Address 0x20000000, Len 0x6a120 (424 KB)
- RAM: Address 0x80000000, Len 0x4000 (16 KB)

3.9.3 Stock bootloader

Start Address: 0x20000000 is 64KB. Provides a "double tap" reset feature to halt boot and allow debugger to attach for reprogramming. Press reset button, when green light comes on press reset button again, then board will flash red.

3.9.4 Application Code

Start Address: 0x20010000

3.9.5 wolfBoot configuration

The default wolfBoot configuration will add a second stage bootloader, leaving the stock "double tap" bootloader as a fallback for recovery. Your production implementation should replace this and partition addresses in `target.h` will need updated, so they are `0x10000` less.

To set the Freedom SDK location use `FREEDOM_E_SDK=~/src/freedom-e-sdk`.

For testing wolfBoot here are the changes required:

1. Makefile arguments:
   - ARCH=RISCV
   - TARGET=hifive1

   ```
   make ARCH=RISCV TARGET=hifive1 RAM_CODE=1 clean
   make ARCH=RISCV TARGET=hifive1 RAM_CODE=1
   ```

   If using the `riscv64-unknown-elf-` cross compiler you can add `CROSS_COMPILE=riscv64-unknown-elf-` to your `make` or modify `arch.mk` as follows:

   ```
    ifeq ($(ARCH),RISCV)
   -  CROSS_COMPILE:=riscv32-unknown-elf-
   +  CROSS_COMPILE:=riscv64-unknown-elf-
   ```

2. `include/target.h`

Bootloader Size: 0x10000 (64KB) Application Size 0x40000 (256KB) Swap Sector Size: 0x1000 (4KB)

```
#define WOLFBOOT_SECTOR_SIZE                0x1000
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x20020000

#define WOLFBOOT_PARTITION_SIZE             0x40000
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x20060000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x200A0000
```

3.9.6 Build Options

- To use ECC instead of ED25519 use make argument `SIGN=ECC256`
- To output wolfboot as hex for loading with JLink use make argument `wolfboot.hex`

3.9.7 Loading

Loading with JLink:

```
JLinkExe -device FE310 -if JTAG -speed 4000 -jtagconf -1,-1 -autoconnect 1
loadbin factory.bin 0x20010000
rnh
```

3.9.8 Debugging

Debugging with JLink:

In one terminal: `JLinkGDBServer -device FE310 -port 3333`

In another terminal:

```
riscv64-unknown-elf-gdb wolfboot.elf -ex "set remotetimeout 240" -ex "target
    extended-remote localhost:3333"
add-symbol-file test-app/image.elf 0x20020100
```

3.10 STM32F7

The STM32-F76x and F77x offer dual-bank hardware-assisted swapping. The flash geometry must be defined beforehand, and wolfBoot can be compiled to use hardware assisted bank-swapping to perform updates.

Example 2MB partitioning on STM32-F769:

- Dual-bank configuration

BANK A: 0x08000000 to 0x080FFFFFF (1MB) BANK B: 0x08100000 to 0x081FFFFFF (1MB)

- WolfBoot executes from BANK A after reboot (address: 0x08000000)
- Boot partition @ BANK A + 0x20000 = 0x08020000
- Update partition @ BANK B + 0x20000 = 0x08120000
- Application entry point: 0x08020100

```
#define WOLFBOOT_SECTOR_SIZE              0x20000
#define WOLFBOOT_PARTITION_SIZE           0x40000

#define WOLFBOOT_PARTITION_BOOT_ADDRESS   0x08020000
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS 0x08120000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS   0x0   /* Unused, swap is hw-assisted
 ↪  */
```

3.10.1 Build Options

To activate the dual-bank hardware-assisted swap feature on STM32F76x/77x, use the `DUALBANK_SWAP=1` compile time option. Some code requires to run in RAM during the swapping of the images, so the compile-time option `RAMCODE=1` is also required in this case.

Dual-bank STM32F7 build can be built using:

```
make TARGET=stm32f7 DUALBANK_SWAP=1 RAM_CODE=1
```


3.10.2 Loading the firmware

To switch between single-bank (1x2MB) and dual-bank (2 x 1MB) mode mapping, this stm32f7-dualbank-tool can be used. Before starting openocd, switch the flash mode to dualbank (e.g. via `make dualbank` using the dualbank tool).

OpenOCD configuration for flashing/debugging, can be copied into `openocd.cfg` in your working directory:

```
source [find interface/stlink.cfg]
source [find board/stm32f7discovery.cfg]
$_TARGETNAME configure -event reset-init {
    mmw 0xe0042004 0x7 0x0
}
init
reset
halt
```

OpenOCD can be either run in background (to allow remote GDB and monitor terminal connections), or directly from command line, to execute terminal scripts.

If OpenOCD is running, local TCP port 4444 can be used to access an interactive terminal prompt. `telnet localhost 4444`

Using the following openocd commands, the initial images for wolfBoot and the test application are loaded to flash in bank 0:

```
flash write_image unlock erase wolfboot.bin 0x08000000
flash verify_bank 0 wolfboot.bin
flash write_image unlock erase test-app/image_v1_signed.bin 0x08020000
flash verify_bank 0 test-app/image_v1_signed.bin 0x20000
reset
resume 0x0000001
```

To sign the same application image as new version (2), use the python script `sign.py` provided:

```
tools/keytools/sign.py test-app/image.bin wolfboot_signing_private_key.der 2
```

From OpenOCD, the updated image (version 2) can be flashed to the second bank:

```
flash write_image unlock erase test-app/image_v2_signed.bin 0x08120000
flash verify_bank 0 test-app/image_v1_signed.bin 0x20000
```

Upon reboot, wolfboot will elect the best candidate (version 2 in this case) and authenticate the image. If the accepted candidate image resides on BANK B (like in this case), wolfBoot will perform one bank swap before booting.

The bank-swap operation is immediate and a SWAP image is not required in this case. Fallback mechanism can rely on a second choice (older firmware) in the other bank.

3.10.3 STM32F7 Debugging

Debugging with OpenOCD:

Use the OpenOCD configuration from the previous section to run OpenOCD.

From another console, connect using gdb, e.g.:

```
arm-none-eabi-gdb
(gdb) target remote:3333
```

3.11 STM32H7

The STM32H7 flash geometry must be defined beforehand.

Use the “make config” operation to generate a .config file or copy the template using `cp ./config/examples/stm32h7.config .config`.

Example 2MB partitioning on STM32-H753:

```
WOLFBOOT_SECTOR_SIZE?=0x20000
WOLFBOOT_PARTITION_SIZE?=0xD0000
WOLFBOOT_PARTITION_BOOT_ADDRESS?=0x8020000
WOLFBOOT_PARTITION_UPDATE_ADDRESS?=0x80F0000
WOLFBOOT_PARTITION_SWAP_ADDRESS?=0x81C0000
```

3.11.1 Build Options

The STM32H7 build can be built using:

```
make TARGET=stm32h7 SIGN=ECC256
```

3.11.2 STM32H7 Programming

ST-Link Flash Tools:

```
st-flash write factory.bin 0x08000000
```

OR

```
st-flash write wolfboot.bin 0x08000000
st-flash write test-app/image_v1_signed.bin 0x08020000
```

3.11.3 STM32H7 Testing

To sign the same application image as new version (2), use the sign tools

Python: `tools/keytools/sign.py --ecc256 --sha256 test-app/image.bin wolfboot_signing_private_key.der 2` C Tool: `tools/keytools/sign --ecc256 --sha256 test-app/image.bin wolfboot_signing_private_key.der 2`

Flash the updated version 2 image: `st-flash write test-app/image_v2_signed.bin 0x08120000`

Upon reboot, wolfboot will elect the best candidate (version 2 in this case) and authenticate the image. If the accepted candidate image resides on BANK B (like in this case), wolfBoot will perform one bank swap before booting.

3.11.4 STM32H7 Debugging

1. Start GDB server

ST-Link: `st-util -p 3333`

2. Start GDB Client from wolfBoot root:

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf 0x08020000
mon reset init
b main
c
```

3.12 NXP LPC54xxx

3.12.1 Build Options

The LPC54xxx build can be obtained by specifying the CPU type and the MCUXpresso SDK path at compile time.

The following configuration has been tested against LPC54606J512BD208:

```
make TARGET=lpc SIGN=ECC256 MCUXPRESSO?=/path/to/LPC54606J512/SDK
    MCUXPRESSO_CPU?=LPC54606J512BD208 \
    MCUXPRESSO_DRIVERS?=$(MCUXPRESSO)/devices/LPC54606 \
    MCUXPRESSO_CMSIS?=$(MCUXPRESSO)/CMSIS
```

3.12.2 Loading the firmware

Loading with JLink (example: LPC54606J512)

```
JLinkExe -device LPC606J512 -if SWD -speed 4000
erase
loadbin factory.bin 0
r
h
```

3.12.3 Debugging with JLink

```
JLinkGDBServer -device LPC606J512 -if SWD -speed 4000 -port 3333
```

Then, from another console:

```
arm-none-eabi-gdb wolfboot.elf -ex "target remote localhost:3333"
(gdb) add-symbol-file test-app/image.elf 0x0000a100
```

3.13 Cortex-A53 / Raspberry PI 3 (experimental)

Tested using `https://github.com/raspberrypi/linux` on Ubuntu 20

Prerequsites: `sudo apt install gcc-aarch64-linux-gnu qemu-system-aarch64`

3.13.1 Compiling the kernel

- Get raspberry-pi linux kernel:

```
git clone https://github.com/raspberrypi/linux linux-rpi -b rpi-4.19.y --depth
    =1
```

- Build kernel image:

```
export wolfboot_dir=`pwd`
cd linux-rpi
patch -p1 < $wolfboot_dir/tools/wolfboot-rpi-devicetree.diff
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcmrpi3_defconfig
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
```

- Copy Image and .dtb to the wolfboot directory

```
cp ./arch/arm64/boot/Image arch/arm64/boot/dts/broadcom/bcm2710-rpi-3-b.dtb
    $wolfboot_dir
cd $wolfboot_dir
```

3.13.2 Testing with qemu-system-aarch64

- Build wolfboot using the example configuration (RSA4096, SHA3)

```
cp config/examples/raspi3.config .config
make clean
make wolfboot.bin CROSS_COMPILE=aarch64-linux-gnu-
```

- Sign Linux kernel image

```
make keytools
./tools/keytools/sign --rsa4096 --sha3 Image wolfboot_signing_private_key.der
    1
```

- Compose the image

```
tools/bin-assemble/bin-assemble wolfboot_linux_raspi.bin 0x0 wolfboot.bin \
                                0xc0000 Image_v1_signed.bin
dd if=bcm2710-rpi-3-b.dtb of=wolfboot_linux_raspi.bin bs=1 seek=128K conv=
    notrunc
```

- Test boot using qemu

```
qemu-system-aarch64 -M raspi3b -m 1024 -serial stdio -kernel
    wolfboot_linux_raspi.bin -cpu cortex-a53
```

3.14 Xilinx Zynq UltraScale

Xilinx UltraScale+ ZCU102 (Aarch64)

Build configuration options (`.config`):

```
TARGET=zynq
ARCH=AARCH64
SIGN=RSA4096
HASH=SHA3
```

3.14.1 QNX

```
cd ~
source qnx700/qnxsdp-env.sh
cd wolfBoot
cp ./config/examples/zynqmp.config .config
make clean
make CROSS_COMPILE=aarch64-unknown-nto-qnx7.0.0-
```

3.14.1.1 Debugging `qemu-system-aarch64 -M raspi3 -kernel /path/to/wolfboot/fac-tory.bin -serial stdio -gdb tcp::3333 -S`

3.14.1.2 Signing `tools/keytools/sign.py --rsa4096 --sha3 /srv/linux-rpi4/vmlinux.bin wolfboot_signing_private_key.der 1`

3.15 Cypress PSoC-6

The Cypress PSoC 62S2 is a dual-core Cortex-M4 & Cortex-M0+ MCU. The secure boot process is managed by the M0+. WolfBoot can be compiled as second stage flash bootloader to manage application verification and firmware updates.

3.15.1 Building

The following configuration has been tested using PSoC 62S2 Wi-Fi BT Pioneer Kit (CY8CKIT-052S2-43012).

3.15.1.1 Target specific requirements wolfBoot uses the following components to access peripherals on the PSoC:

- Cypress Core Library
- PSoC 6 Peripheral Driver Library
- CY8CKIT-062S2-43012 BSP

Cypress provides a customized OpenOCD for programming the flash and debugging.

3.15.2 Clock settings

wolfBoot configures PLL1 to run at 100 MHz and is driving `CLK_FAST`, `CLK_PERI`, and `CLK_SLOW` at that frequency.

3.15.2.1 Build configuration The following configuration has been tested on the PSoC CY8CKIT-62S2-43012:

```
make TARGET=psoc6 \
    NVM_FLASH_WRITEONCE=1 \
    CYPRESS_PDL=./lib/psoc6pdl \
    CYPRESS_TARGET_LIB=./lib/TARGET_CY8CKIT-062S2-43012 \
    CYPRESS_CORE_LIB=./lib/core-lib \
    WOLFBOOT_SECTOR_SIZE=4096
```

Note: A reference `.config` can be found in `/config/examples/cypsoc6.config`.

Hardware acceleration is enable by default using psoc6 crypto hw support.

To compile with hardware acceleration disabled, use the option

`PSOC6_CRYPTO=0`

in your wolfBoot configuration.

3.15.2.2 OpenOCD installation Compile and install the customized OpenOCD.

Use the following configuration file when running `openocd` to connect to the PSoC6 board:

```
### openocd.cfg for PSoC-62S2

source [find interface/kitprog3.cfg]
transport select swd
adapter speed 1000
source [find target/psoc6_2m.cfg]
init
reset init
```

3.15.3 Loading the firmware

To upload `factory.bin` to the device with OpenOCD, connect the device, run OpenOCD with the configuration from the previous section, then connect to the local openOCD server running on TCP port 4444 using `telnet localhost 4444`.

From the telnet console, type:

```
program factory.bin 0x10000000
```

When the transfer is finished, you can either close openOCD or start a debugging session.

3.15.4 Debugging

Debugging with OpenOCD:

Use the OpenOCD configuration from the previous sections to run OpenOCD.

From another console, connect using gdb, e.g.:

```
arm-none-eabi-gdb
(gdb) target remote:3333
```

To reset the board to start from the M0+ flash bootloader position (wolfBoot reset handler), use the monitor command sequence below:

```
(gdb) mon init
(gdb) mon reset init
(gdb) mon psoc6 reset_halt
```

3.16 NXP iMX-RT

NXP RT1060/1062 and RT1050

The NXP iMX-RT1060 is a Cortex-M7 with a DCP coprocessor for SHA256 acceleration. Example configuration for this target is provided in `/config/examples/imx-rt1060.config`.

3.16.1 Building wolfBoot

MCUXpresso SDK is required by wolfBoot to access device drivers on this platform. A package can be obtained from the MCUXpresso SDK Builder, by selecting a target and keeping the default choice of components.

- For the RT1060 use `EVKB-IMXRT1060`. See configuration example in `config/examples/imx-rt1060.config`.
- For the RT1050 use `EVKB-IMXRT1050`. See configuration example in `config/examples/imx-rt1050.config`.

Set the wolfBoot `MCUXPRESSO` configuration variable to the path where the SDK package is extracted, then build wolfBoot normally by running `make`.

wolfBoot support for iMX-RT1060/iMX-RT1050 has been tested using MCUXpresso SDK version 2.11.1.

DCP support (hardware acceleration for SHA256 operations) can be enabled by using PKA=1 in the configuration file. Firmware can be directly uploaded to the target by copying `factory.bin` to the virtual USB drive associated to the device.

3.17 NXP Kinetis

Supports K64 and K82 with crypto hardware acceleration.

3.17.1 Buld options

See `/config/examples/kinetis-k82f.config` for example configuration.

The TARGET is `kinetis`. For LTC PKA support set PKA=.

Set `MCUXPRESSO`, `MCUXPRESSO_CPU`, `MCUXPRESSO_DRIVERS` and `MCUXPRESSO_CMSIS` for MCUXpresso configuration.

3.17.2 Example partioning for K82

```
WOLFBOOT_PARTITION_SIZE?=0x7A000
WOLFBOOT_SECTOR_SIZE?=0x1000
WOLFBOOT_PARTITION_BOOT_ADDRESS?=0xA000
WOLFBOOT_PARTITION_UPDATE_ADDRESS?=0x84000
WOLFBOOT_PARTITION_SWAP_ADDRESS?=0xff000
```

3.18 NXP T2080 PPC

The T2080 is a PPC e6500 based processor.

Example configuration for this target is provided in `/config/examples/t2080.config`.

3.18.1 Building wolfBoot

wolfBoot can be built with gcc powerpc tools. For example, `apt install gcc-powerpc-linux-gnu`. Then make will use the correct tools to compile.

3.19 TI Hercules TMS570LC435

See `/config/examples/ti-tms570lc435.config` for example configuration.

3.20 Qemu x86-64 UEFI

x86-64bit machine with UEFI bios can run wolfBoot as EFI application.

3.20.1 Prerequisites:

- qemu-system-x86_64
- [GNU-EFI] (https://sourceforge.net/projects/gnu-efi/)
- Open Virtual Machine firmware bios images (OVMF) by Tianocore

On a debian-like system it is sufficient to install the packages as follows:

```
# for wolfBoot and others
apt install git make gcc

# for test scripts
apt install sudo dosfstools curl
apt install qemu qemu-system-x86 ovmf gnu-efi

# for buildroot
apt install file bzip2 g++ wget cpio unzip rsync bc
```

3.20.2 Configuration

An example configuration is provided in `config/examples/x86_64_efi.config`

3.20.3 Building and running on qemu

The bootloader and the initialization script `startup.nsh` for execution in the EFI environment are stored in a loopback FAT partition.

The script `tools/efi/prepare_uefi_partition.sh` creates a new empty FAT loopback partitions and adds `startup.nsh`.

A kernel with an embedded rootfs partition can be now created and added to the image, via the script `tools/efi/compile_efi_linux.sh`. The script actually adds two instances of the target systems: `kernel.img` and `update.img`, both signed for authentication, and tagged with version 1 and 2 respectively.

Compiling with `make` will produce the bootloader image in `wolfboot.efi`.

The script `tools/efi/run_efi.sh` will add `wolfboot.efi` to the bootloader loopback partition, and run the system on qemu. If both kernel images are present and valid, wolfBoot will choose the image with the higher version number, so `update.img` will be staged as it's tagged with version 2.

The sequence is summarized below:

```
cp config/examples/x86_64_efi.config .config
tools/efi/prepare_efi_partition.sh
make
tools/efi/compile_efi_linux.sh
tools/efi/run_efi.sh
```

```
EFI v2.70 (EDK II, 0x00010000)
[700/1832]
Mapping table
      FS0: Alias(s):F0a:;BLK0:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
     BLK1: Alias(s):
               PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
Press ESC in 1 seconds to skip startup.nsh or any other key to continue.
Starting wolfBoot EFI...
Image base: 0xE3C6000
Opening file: kernel.img, size: 6658272
Opening file: update.img, size: 6658272
Active Part 1
Firmware Valid
Booting at 0D630000
Staging kernel at address D630100, size: 6658016
```

You can `Ctrl-C` or login as `root` and power off qemu with `poweroff`

3.21 Nordic nRF52840

We have full Nordic nRF5280 examples for Contiki and RIOT-OS in our wolfBoot-examples repo

Examples for nRF52: * RIOT-OS: https://github.com/wolfSSL/wolfBoot-examples/tree/master/riotOS-nrf52840dk-ble * Contiki-OS: https://github.com/wolfSSL/wolfBoot-examples/tree/master/contiki-nrf52

Example of flash memory layout and configuration on the nRF52:

- 0x000000 - 0x01efff : Reserved for Nordic SoftDevice binary
- 0x01f000 - 0x02efff : Bootloader partition for wolfBoot
- 0x02f000 - 0x056fff : Active (boot) partition
- 0x057000 - 0x057fff : Unused
- 0x058000 - 0x07ffff : Upgrade partition

```
#define WOLFBOOT_SECTOR_SIZE                4096
#define WOLFBOOT_PARTITION_SIZE             0x28000

#define WOLFBOOT_PARTITION_BOOT_ADDRESS   0x2f000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS   0x57000
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS 0x58000
```

3.22 Simulated

You can create a simulated target that uses files to mimic an internal and optionally an external flash. The build will produce an executable ELF file `wolfBoot.elf`. You can provide another executable ELF as firmware image and it will be executed. The command-line arguments of `wolfBoot.elf` are forwarded to the application. The example application `test-app\app_sim.c` uses the arguments to interact with `libwolfboot.c` and automatize functional testing. You can find an example configuration in `config/examples/sim.config`.

An example of using the `test-app/sim.c` to test firmware update:

```
cp ./config/examples/sim.config .config
make

# create the file internal_flash.dd with firmware v1 on the boot partition and
# firmware v2 on the update partition
make test-sim-internal-flash-with-update
# it should print 1
./wolfboot.elf success get_version
# trigger an update
./wolfboot.elf update_trigger
# it should print 2
./wolfboot.elf success get_version
# it should print 2
./wolfboot.elf success get_version
```

4 Hardware abstraction layer

In order to run wolfBoot on a target microcontroller, an implementation of the HAL must be provided.

The HAL's purpose is to allow write/erase operations from the bootloader and the application initiating the firmware upgrade through the application library, and ensuring that the MCU is running at full speed during boot (to optimize the verification of the signatures).

The implementation of the hardware-specific calls for each platform are grouped in a single c file in the `hal` directory.

The directory also contains a platform-specific linker script for each supported MCU, with the same name and the `.ld` extension. This is used to link the bootloader's firmware on the specific hardware, exporting all the necessary symbols for flash and RAM boundaries.

4.1 Supported platforms

The following platforms are supported in the current version: - STM32F4, STM32L5, STM32L0, STM32F7, STM32H7, STM32G0 - nRF52 - Atmel samR21 - TI cc26x2 - Kinetis - SiFive HiFive1 RISC-V

4.2 API

The Hardware Abstraction Layer (HAL) consists of six function calls be implemented for each supported target:

`void hal_init(void)`

This function is called by the bootloader at the very beginning of the execution. Ideally, the implementation provided configures the clock settings for the target microcontroller, to ensure that it runs at at the required speed to shorten the time required for the cryptography primitives to verify the firmware images.

`void hal_flash_unlock(void)`

If the IAP interface of the flash memory of the target requires it, this function is called before every write and erase operations to unlock write access to the flash. On some targets, this function may be empty.

`int hal_flash_write(uint32_t address, const uint8_t *data, int len)`

This function provides an implementation of the flash write function, using the target's IAP interface. `address` is the offset from the beginning of the flash area, `data` is the payload to be stored in the flash using the IAP interface, and `len` is the size of the payload. `hal_flash_write` should return 0 upon success, or a negative value in case of failure.

`void hal_flash_lock(void)`

If the IAP interface of the flash memory requires locking/unlocking, this function restores the flash write protection by excluding write accesses. This function is called by the bootloader at the end of every write and erase operations.

`int hal_flash_erase(uint32_t address, int len)`

Called by the bootloader to erase part of the flash memory to allow subsequent boots. Erase operations must be performed via the specific IAP interface of the target microcontroller. `address` marks the start of the area that the bootloader wants to erase, and `len` specifies the size of the area to be erased. This function must take into account the geometry of the flash sectors, and erase all the sectors in between.

`void hal_prepare_boot(void)`

This function is called by the bootloader at a very late stage, before chain-loading the firmware in the next stage. This can be used to revert all the changes made to the clock settings, to ensure that the state of the microcontroller is restored to its original settings.

4.2.1 Optional support for external flash memory

WolfBoot can be compiled with the makefile option EXT_FLASH=1. When the external flash support is enabled, update and swap partitions can be associated to an external memory, and will use alternative HAL function for read/write/erase access. To associate the update or the swap partition to an external memory, define PART_UPDATE_EXT and/or PART_SWAP_EXT, respectively.

The following functions are used to access the external memory, and must be defined when EXT_FLASH is on:

```
int  ext_flash_write(uintptr_t address, const uint8_t *data, int len)
```

This function provides an implementation of the flash write function, using the external memory's specific interface. `address` is the offset from the beginning of the addressable space in the device, `data` is the payload to be stored, and `len` is the size of the payload. `ext_flash_write` should return 0 upon success, or a negative value in case of failure.

```
int  ext_flash_read(uintptr_t address, uint8_t *data, int len)
```

This function provides an indirect read of the external memory, using the driver's specific interface. `address` is the offset from the beginning of the addressable space in the device, `data` is a pointer where payload is stored upon a successful call, and `len` is the maximum size allowed for the payload. `ext_flash_read` should return 0 upon success, or a negative value in case of failure.

```
int  ext_flash_erase(uintptr_t address, int len)
```

Called by the bootloader to erase part of the external memory. Erase operations must be performed via the specific interface of the target driver (e.g. SPI flash). `address` marks the start of the area relative to the device, that the bootloader wants to erase, and `len` specifies the size of the area to be erased. This function must take into account the geometry of the sectors, and erase all the sectors in between.

```
void ext_flash_lock(void)
```

If the interface of the external flash memory requires locking/unlocking, this function may be used to restore the flash write protection or exclude write accesses. This function is called by the bootloader at the end of every write and erase operations on the external device.

```
void ext_flash_unlock(void)
```

If the IAP interface of the external memory requires it, this function is called before every write and erase operations to unlock write access to the device. On some drivers, this function may be empty.

5 Flash partitions

5.1 Flash memory partitions

To integrate wolfBoot you need to partition the flash into separate areas (partitions), taking into account the geometry of the flash memory.

Images boundaries **must** be aligned to physical sectors, because the bootloader erases all the flash sectors before storing a new firmware image, and swaps the content of the two partitions, one sector at a time.

For this reason, before proceeding with partitioning on a target system, the following aspects must be considered:

- BOOT partition and UPDATE partition must have the same size, and be able to contain the running system
- SWAP partition must be as big as the largest sector in both BOOT and UPDATE partition.

The flash memory of the target is partitioned into the following areas:

- Bootloader partition, at the beginning of the flash, generally very small (16-32KB)
- Primary slot (BOOT partition) starting at address `WOLFBOOT_PARTITION_BOOT_ADDRESS`
- Secondary slot (UPDATE partition) starting at address `WOLFBOOT_PARTITION_UPDATE_ADDRESS`
  - both partitions share the same size, defined as `WOLFBOOT_PARTITION_SIZE`
- Swapping space (SWAP partition) starting at address `WOLFBOOT_PARTITION_SWAP_ADDRESS`
  - the swap space size is defined as `WOLFBOOT_SECTOR_SIZE` and must be as big as the largest sector used in either BOOT/UPDATE partitions.

A proper partitioning configuration must be set up for the specific use, by setting the values for offsets and sizes in `include/target.h`.

5.1.1 Bootloader partition

This partition is usually very small, and only contains the bootloader code and data. Public keys pre-authorized during factory image creations are automatically stored as part of the firmware image.

5.1.2 BOOT partition

This is the only partition from where it is possible to chain-load and execute a firmware image. The firmware image must be linked so that its entry-point is at address `WOLFBOOT_PARTITION_BOOT_ADDRESS + 256`.

5.1.3 UPDATE partition

The running firmware is responsible for transferring a new firmware image through a secure channel, and store it in the secondary slot. If an update is initiated, the bootloader will replace or swap the firmware in the boot partition at the next reboot.

5.2 Partition status and sector flags

Partitions are used to store firmware images currently in use (BOOT) or ready to swap in (UPDATE). In order to track the status of the firmware in each partition, a 1-Byte state field is stored at the end of each partition space. This byte is initialized when the partition is erased and accessed for the first time.

Possible states are: - STATE_NEW (0xFF): The image was never staged for boot, or triggered for an update. If an image is present, no flags are active. - STATE_UPDATING (0x70): Only valid in the UPDATE partition. The image is marked for update and should replace the current image in BOOT. - STATE_TESTING (0x10): Only valid in the BOOT partition. The image has been just updated, and never completed its boot. If present after reboot, it means that the updated image failed to boot, despite being correctly verified. This particular situation triggers a rollback. - STATE_SUCCESS (0x00): Only valid in the BOOT partition. The image stored in BOOT has been successfully staged at least once, and the update is now complete.

Starting from the State byte and growing backwards, the bootloader keeps track of the state of each sector, using 4 bits per sector at the end of the UPDATE partition. Whenever an update is initiated, the firmware is transferred from UPDATE to BOOT one sector at a time, and storing a backup of the original firmware from BOOT to UPDATE. Each flash access operation correspond to a different value of the flags for the sector in the sector flags area, so that if the operation is interrupted, it can be resumed upon reboot.

5.3 Overview of the content of the FLASH partitions

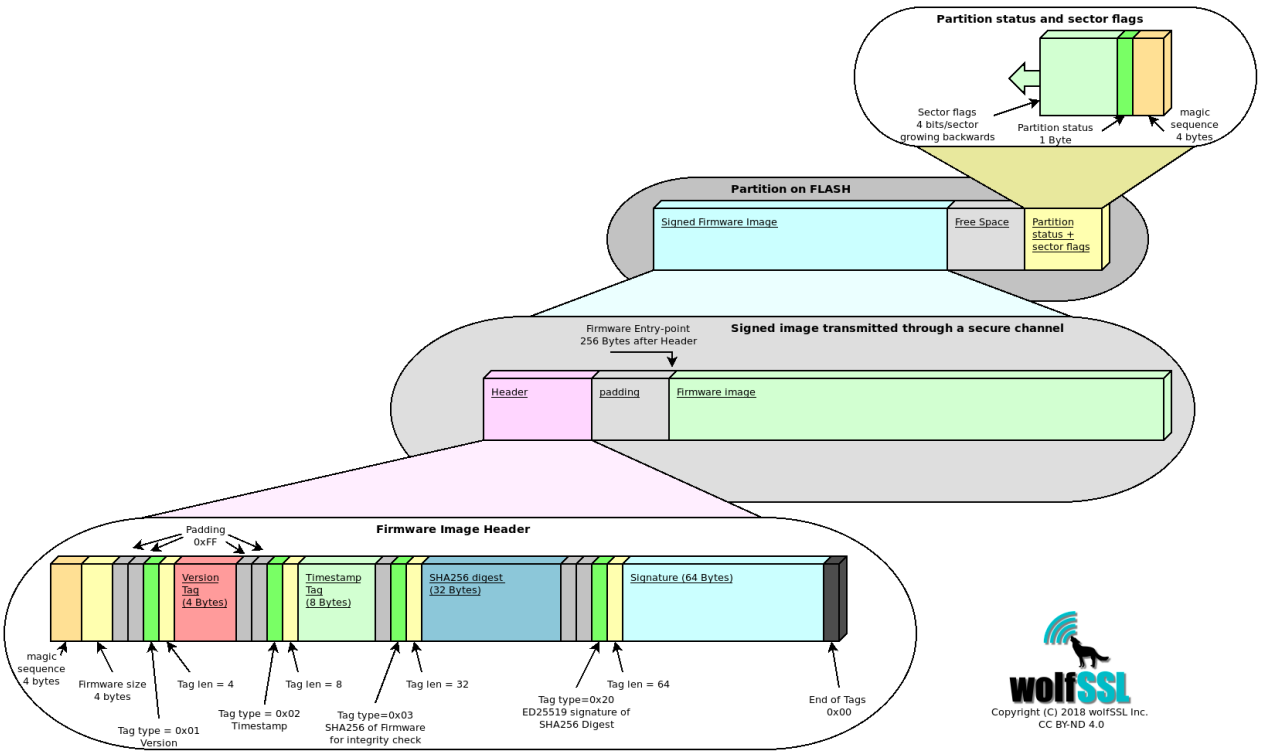

Figure 2: wolfBoot partition

6 wolfBoot Features

6.1 Signing

6.1.1 wolfBoot key tools installation

Instructions for setting up Python, wolfCrypt-py module and wolfBoot for firmware signing and key generation.

Note: There is a pure C version of the key tool available as well. See C Key Tools below.

6.1.2 Install Python3

1. Download latest Python 3.x and run installer: https://www.python.org/downloads
2. Check the box that says Add Python 3.x to PATH

6.1.3 Install wolfCrypt

```
git clone https://github.com/wolfSSL/wolfssl.git
cd wolfssl
./configure --enable-keygen --enable-rsa --enable-ecc --enable-ed25519 --
   enable-ed448 --enable-des3 CFLAGS="-DWOLFSSL_PUBLIC_MP"
make
sudo make install
```

6.1.4 Install wolfcrypt-py

```
git clone https://github.com/wolfSSL/wolfcrypt-py.git
cd wolfcrypt-py
sudo USE_LOCAL_WOLFSSL=/usr/local pip3 install .
```

6.1.5 Install wolfBoot

```
git clone https://github.com/wolfSSL/wolfBoot.git
cd wolfBoot
git submodule update --init
## Setup configuration (or copy template from ./config/examples)
make config
## Build the wolfBoot binary and sign an example test application
make
```

6.1.6 C Key Tools

A standalone C version of the keygen tools is available in: `./tools/keytools`.

These can be built in `tools/keytools` using `make` or from the wolfBoot root using `make keytools`.

If the C version of the key tools exists they will be used by wolfBoot (the default is the Python scripts).

6.1.6.1 Windows Visual Studio Use the `wolfBootSignTool.vcxproj` Visual Studio project to build the `sign.exe` and `keygen.exe` tools for use on Windows.

6.1.7 Command Line Usage

6.1.7.1 Keygen tool Usage: `keygen[.py] [OPTIONS] [-g new-keypair.der] [-i existing-pubkey.der] [...]`

keygen is used to populate a keystore with existing and new public keys. Two options are supported:

- `-g privkey.der` to generate a new keypair, add the public key to the keystore and save the private key in a new file `privkey.der`
- `-i existing.der` to import an existing public key from `existing.der`

Arguments are not exclusive, and can be repeated more than once to populate a keystore with multiple keys.

One option must be specified to select the algorithm enabled in the keystore (e.g. `--ed25519` or `--rsa3072`. See the section "Public key signature options" for the sign tool for the available options.

The files generate by the keygen tool is the following:

- A C file `src/keystore.c`, which is normally linked with the wolfBoot image, when the keys are provisioned through generated C code.
- A binary file `keystore.img` that can be used to provision the public keys through an alternative storage
- The private key, for each `-g` option provided from command line

6.1.7.2 Sign tool `sign` and `sign.py` produce a signed firmware image by creating a manifest header in the format supported by wolfBoot.

Usage: `sign[.py] [OPTIONS] IMAGE.BIN KEY.DER VERSION`

`IMAGE.BIN`: A file containing the binary firmware/software to sign `KEY.DER`: Private key file, in DER format, to sign the binary image `VERSION`: The version associated with this signed software `OPTIONS`: Zero or more options, described below

6.1.7.3 Public key signature options If none of the following arguments is given, the tool will try to guess the key size from the format and key length detected in KEY.DER.

- `--ed25519` Use ED25519 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--ed448` Use ED448 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--ecc256` Use ecc256 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--ecc384` Use ecc384 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--rsa2048` Use rsa2048 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--rsa3072` Use rsa3072 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--rsa4096` Use rsa4096 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--no-sign` Disable secure boot signature verification. No signature verification is performed in the bootloader, and the KEY.DER argument is ignored.

6.1.8 Key generation and management

KeyStore is the name of the mechanism used by wolfBoot to store all the public keys used for authenticating the signature of current firmware and updates.

wolfBoot's key generation tool can be used to generate one or more keys. By default, when running `make` for the first time, a single key `wolfboot_signing_private_key.der` is created, and added to the keystore module. This key should be used to sign any firmware running on the target, as well as firmware update binaries.

Additionally, the keygen tool creates additional files with different representations of the keystore - A .c file (src/keystore.c) which can be used to deploy public keys as part of the bootloader itself, by linking the keystore in `wolfboot.elf` - A .bin file (keystore.bin) which contains the keystore that can be hosted on a custom memory support. In order to access the keystore, a small driver is required (see section "Interface API" below).

By default, the keystore object in `src/keystore.c` is accessed by wolfboot by including its symbols in the build. Once generated, this file contains an array of structures describing each public key that will be available to wolfBoot on the target system. Additionally, there are a few functions that connect to the wolfBoot keystore API to access the details and the content of the public key slots.

The public key is described by the following structure:

```
struct keystore_slot {
    uint32_t slot_id;
    uint32_t key_type;
    uint32_t part_id_mask;
    uint32_t pubkey_size;
    uint8_t  pubkey[KEYSTORE_PUBKEY_SIZE];
};
```

- `slot_id` is the incremental identifier for the key slot, starting from 0.
- `key_type` describes the algorithm of the key, e.g. `AUTH_KEY_ECC256` or `AUTH_KEY_RSA3072`
- `mask` describes the permissions for the key. It's a bitmap of the partition ids for which this key can be used for verification
- `pubkey_size` the size of the public key buffer
- `pubkey` the actual buffer containing the public key in its raw format

When booting, wolfBoot will automatically select the public key associated to the signed firmware image, check that it matches the permission mask for the partition id where the verification is running and then attempts to authenticate the signature of the image using the selected public key slot.

6.1.8.1 Creating multiple keys

keygen accepts multiple filenames for private keys.

Two arguments:

- `-g priv.der` generate new keypair, store the private key in priv.der, add the public key to the keystore
- `-i pub.der` import an existing public key and add it to the keystore

Example of creation of a keystore with two ED25519 keys:

```
./tools/keytools/keygen.py --ed25519 -g first.der -g second.der
```

will create the following files:

- `first.der` first private key
- `second.der` second private key
- `src/keystore.c` C keystore containing both public keys associated with `first.der` and `second.der`.

The `keystore.c` generated should look similar to this:

```
#define NUM_PUBKEYS 2
const struct keystore_slot PubKeys[NUM_PUBKEYS] = {

     /* Key associated to private key 'first.der' */
    {
        .slot_id = 0,
        .key_type = AUTH_KEY_ED25519,
        .part_id_mask = KEY_VERIFY_ALL,
        .pubkey_size = KEYSTORE_PUBKEY_SIZE_ED25519,
        .pubkey = {
            0x21, 0x7B, 0x8E, 0x64, 0x4A, 0xB7, 0xF2, 0x2F,
            0x22, 0x5E, 0x9A, 0xC9, 0x86, 0xDF, 0x42, 0x14,
            0xA0, 0x40, 0x2C, 0x52, 0x32, 0x2C, 0xF8, 0x9C,
            0x6E, 0xB8, 0xC8, 0x74, 0xFA, 0xA5, 0x24, 0x84
        },
    },

     /* Key associated to private key 'second.der' */
    {
        .slot_id = 1,
        .key_type = AUTH_KEY_ED25519,
        .part_id_mask = KEY_VERIFY_ALL,
        .pubkey_size = KEYSTORE_PUBKEY_SIZE_ED25519,
        .pubkey = {
            0x41, 0xC8, 0xB6, 0x6C, 0xB5, 0x4C, 0x8E, 0xA4,
            0xA7, 0x15, 0x40, 0x99, 0x8E, 0x6F, 0xD9, 0xCF,
            0x00, 0xD0, 0x86, 0xB0, 0x0F, 0xF4, 0xA8, 0xAB,
            0xA3, 0x35, 0x40, 0x26, 0xAB, 0xA0, 0x2A, 0xD5
        },
    },


};
```

6.1.8.2 Public keys and permissions By default, when a new keystore is created, the permissions mask is set to `KEY_VERIFY_ALL`, which means that the key can be used to verify a firmware targeting any partition id.

To restrict the permissions for single keys, it would be sufficient to change the value of their `part_id_mask` attributes.

The `part_id_mask` value is a bitmask, where each bit represent a different partition. The bit '0' is reserved for wolfBoot self-update, while typically the main firmware partition is associated to id 1, so it requires a key with the bit '1' set. In other words, signing a partition with `--id 3` would require turning on bit '3' in the mask, i.e. adding (1U « 3) to it.

Beside `KEY_VERIFY_ALL`, pre-defined mask values can also be used here:

- `KEY_VERIFY_APP_ONLY` only verifies the main application, with partition id 1
- `KEY_VERIFY_SELF_ONLY` this key can only be used to authenticate wolfBoot self-updates (id = 0)
- `KEY_VERIFY_ONLY_ID(N)` macro that can be used to restrict the usage of the key to a specific partition id `N`

6.1.9 Signing Firmware

1. Load the private key to use for signing into `./wolfboot_signing_private_key.der`
2. Run the signing tool with asymmetric algorithm, hash algorithm, file to sign, key and version.

```
./tools/keytools/sign --rsa2048 --sha256 test-app/image.bin
 ↪  wolfboot_signing_private_key.der 1
# OR
python3 ./tools/keytools/sign.py --rsa2048 --sha256 test-app/image.bin
 ↪  wolfboot_signing_private_key.der 1
```

Note: The last argument is the "version" number.

6.1.10 Signing Firmware with External Private Key (HSM)

Steps for manually signing firmware using an external key source.

```
# Create file with Public Key
openssl rsa -inform DER -outform DER -in my_key.der -out rsa2048_pub.der
 ↪  -pubout

# Add the public key to the wolfBoot keystore using `keygen -i`
./tools/keytools/keygen --rsa2048 -i rsa2048_pub.der
# OR
python3 ./tools/keytools/keygen.py --rsa2048 -i rsa4096_pub.der

# Generate Hash to Sign
./tools/keytools/sign            --rsa2048 --sha-only --sha256
 ↪  test-app/image.bin rsa2048_pub.der 1
# OR
python3 ./tools/keytools/sign.py --rsa2048 --sha-only --sha256
 ↪  test-app/image.bin rsa4096_pub.der 1

# Sign hash Example (here is where you would use an HSM)
openssl pkeyutl -sign -keyform der -inkey my_key.der -in
 ↪  test-app/image_v1_digest.bin > test-app/image_v1.sig

# Generate final signed binary
./tools/keytools/sign            --rsa2048 --sha256 --manual-sign
 ↪  test-app/image.bin rsa2048_pub.der 1 test-app/image_v1.sig
# OR
python3 ./tools/keytools/sign.py --rsa2048 --sha256 --manual-sign
 ↪  test-app/image.bin rsa4096_pub.der 1 test-app/image_v1.sig

# Combine into factory image (0xc0000 is the WOLFBOOT_PARTITION_BOOT_ADDRESS)
tools/bin-assemble/bin-assemble factory.bin 0x0 wolfboot.bin \
                                0xc0000 test-app/image_v1_signed.bin
```

6.2 Measured Boot using wolfBoot

wolfBoot offers a simplified measured boot implementation, a way to record and track the state of the system boot process using a Trusted Platform Module(TPM).

This record is tamper-proofed by special registers in the TPM called Platform Configuration Register. Then, the firmware application, RTOS or rich OS(Linux), can access that log of information by reading the PCRs of the TPM.

wolfBoot can interact with TPM2.0 chips thanks to its integration with wolfTPM. wolfTPM has native support for Microsoft Windows and Linux, and can be used standalone or together with wolfBoot. The combination of wolfBoot with wolfTPM gives the developer a tamper-proof secure storage for protecting the system during and after boot.

6.2.1 Concept

Typically, systems use Secure Boot to guarantee that the correct and geniune firmware is booted by verifying its signature. Afterwards, this knowledge is unknown to the sytem. The application does not know if the system started in a good known state. Sometimes, this guarantee is needed by the firmware itself. To provide such mechanism the concept of Measured Boot exist.

Measured Boot can be used to check every start-up component, including settings and user information(user partition). The result of the checks is then stored into special registers called PCR. This process is called PCR Extend and is refered to as a TPM measurement. PCR registers can be reset only on TPM power-on.

Having TPM measurements provide a way for the firmware or Operating System(OS), like Windows or Linux, to know that the software loaded before it gained control over system, is trustworthy and not modified.

In wolfBoot the concept is simplified to measuring a single component, the main firmware image. However, this can easily be extended by using more PCR registers.

6.2.2 Configuration

To enable measured boot add `MEASURED_BOOT=1` setting in your wolfBoot config.

It is also necessary to select the PCR (index) where the measurement will be stored.

Selection is made using the `MEASURED_BOOT_PCR_A=[index]` setting. Add this setting in your wolfBoot config and replace `[index]` with a number between 0 and 23. Below you will find guidelines for selecting a PCR index.

Any TPM has a minimum of 24 PCR registers. Their typical use is as follows:

Index	Typical use	Recommended to use with
0	Core Root of Trust and/or BIOS measurement	bare-metal, RTOS
1	measurement of Platform Configuration Data	bare-metal, RTOS
2-3	Option ROM Code measurement	bare-metal, RTOS
4-5	Master Boot Record measurement	bare-metal, RTOS
6	State Transitions	bare-metal, RTOS
7	Vendor specific	bare-metal, RTOS
8-9	Partition measurements	bera-metal, RTOS
10	measurement of the Boot Manager	bare-metal, RTOS
11	Typically used by Microsoft Bitlocker	bare-metal, RTOS
12-15	Available for any use	bare-metal, RTOS, Linux, Windows
16	DEBUG	Use only for test purposes
17	DRTM	Trusted Bootloader
18-22	Trusted OS	Trusted Execution Environment(TEE)
23	Application	Use only for temporary measurements

Recommendations for choosing a PCR index:

- During development it is recommended to use PCR16 that is intented for testing.

- In production, if you are running a bare-metal firmware or RTOS, you could use almost all PCRs(PCR0-15), except the one for DRTM and Trusted OS(PCR17-23).
- If you are running Linux or Windows, PCR12-15 can be chosen for production ready firmware, in order to avoid conflict with other software that might be using PCRs from within Linux, like the Linux IMA or Microsoft Bitlocker.

Here is an example part of a wolfBoot .config during development:

```
MEASURED_BOOT?=1
MEASURED_PCR_A?=16
```

6.2.2.1 Code wolfBoot offers out-of-the-box solution. There is zero need of the developer to touch wolfBoot code in order to use measured boot. If you would want to check the code, then look in `src/image.c` and more specifically the `measure_boot()` function. There you would find several TPM2 native API calls to wolfTPM. For more information about wolfTPM you can check its GitHub repository.

6.3 Firmware image

6.3.1 Firmware entry point

WolfBoot can only chain-load and execute firmware images from a specific entry point in memory, which must be specified as the origin of the FLASH memory in the linker script of the embedded application. This corresponds to the first partition in the flash memory.

Multiple firmware images can be created this way, and stored in two different partitions. The bootloader will take care of moving the selected firmware to the first (BOOT) partition before chain-loading the image.

Due to the presence of an image header, the entry point of the application has a fixed additional offset of 256B from the beginning of the flash partition.

6.3.2 Firmware image header

Each (signed) firmware image is pre-pended with a fixed-size **image header**, containing useful information about the firmware. The **image header** is padded to fit in 256B, in order to guarantee that the entry point of the actual firmware is stored on the flash starting from a 256-Bytes aligned address. This ensures that the bootloader can relocate the vector table before chain-loading the firmware the interrupt continue to work properly after the boot is complete.

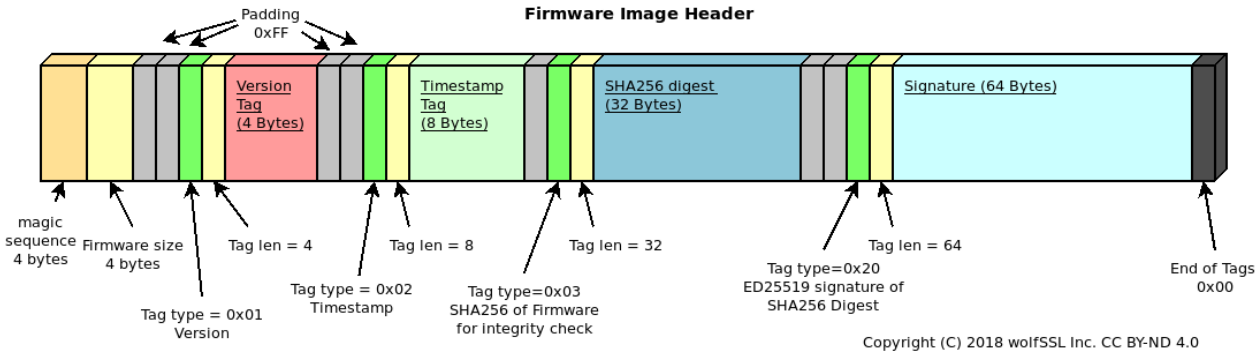

Figure 3: Image header

The image header is stored at the beginning of the slot and the actual firmware image starts 256 Bytes after it

6.3.2.1 Image header: Tags The **image header** is prepended with a single 4-byte magic number, followed by a 4-byte field indicating the firmware image (excluding the header). All numbers in the header are stored in Little-endian format.

The two fixed fields are followed by one or more tags. Each TAG is structured as follows:

- 2 bytes indicating the **Type**
- 2 bytes indicating the **size** of the tag, excluding the type and size bytes
- ***N*** bytes of tag content

With the following exception: - A '0xFF' in the Type field indicate a simple padding byte. The 'padding' byte has no **size** field, and the next byte should be processed as **Type** again. Each **Type** has a different meaning, and integrate information about the firmware. The following Tags are mandatory for validating the firmware image: - A 'version' Tag (type: 0x0001, size: 4 Bytes) indicating the version number for the firmware stored in the image - A 'timestamp' Tag (type: 0x0002, size 8 Bytes) indicating the timestamp in unix seconds for the creation of the firmware - A 'sha256 digest' Tag (type: 0x0003, size: 32 Bytes) used for integrity check of the firmware - A 'firmware signature' Tag (type: 0x0020, size: 64 Bytes) used to validate the signature stored with the firmware against a known public key - A 'firmware type' Tag (type: 0x0030, size: 2 Bytes) used to identify the type of firmware, and the authentication mechanism in use.

Optionally, a 'public key hint digest' Tag can be transmitted in the header (type: 0x10, size:32 Bytes). This Tag contains the SHA256 digest of the public key used by the signing tool. The bootloader may use this field to locate the correct public key in case of multiple keys available.

wolfBoot will, in all cases, refuse to boot an image that cannot be verified and authenticated using the built-in digital signature authentication mechanism.

6.3.2.2 Image signing tool The image signing tool generates the header with all the required Tags for the compiled image, and add them to the output file that can be then stored on the primary slot on the device, or transmitted later to the device through a secure channel to initiate an update.

6.3.2.3 Storing firmware image Firmware images are stored with their full header at the beginning of any of the partitions on the system. wolfBoot can only boot images from the BOOT partition, while keeping a second firmware image in the UPDATE partition.

In order to boot a different image, wolfBoot will have to swap the content of the two images.

For more information on how firmware images are stored and managed within the two partitions, see Flash partitions

6.4 Firmware update

This section documents the complete firmware update procedure, enabling secure boot for an existing embedded application.

6.4.1 Updating Microcontroller FLASH

The steps to complete a firmware update with wolfBoot are: - Compile the firmware with the correct entry point - Sign the firmware - Transfer the image using a secure connection, and store it to the secondary firmware slot - Trigger the image swap - Reboot to let the bootloader begin the image swap

At any given time, an application or OS running on a wolfBoot system can receive an updated version of itself, and store the updated image in the second partition in the FLASH memory.

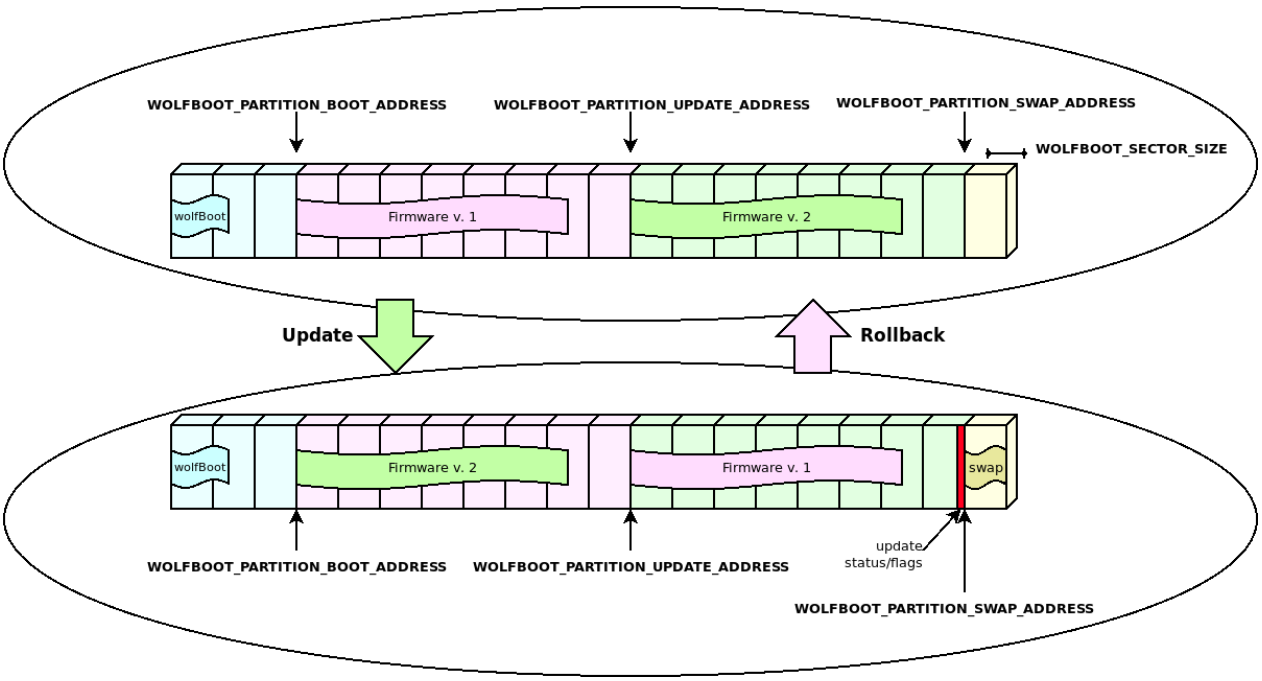

Figure 4: Update and Rollback

Applications or OS threads can be linked to the libwolfboot library, which exports the API to trigger the update at the next reboot, and some helper functions to access the flash partition for erase/write through the target specific HAL.

6.4.2 Update procedure description

Using the API provided to the application, wolfBoot offers the possibility to initiate, confirm or rollback an update.

After storing the new firmware image in the UPDATE partition, the application should initiate the update by calling `wolfBoot_update_trigger()`. By doing so, the UPDATE partition is marked for update. Upon the next reboot, wolfBoot will: - Validate the new firmware image stored in the UPDATE partition - Verify the signature attached against a known public key stored in the bootloader image - Swap the content of the BOOT and the UPDATE partitions - Mark the new firmware in the BOOT partition as in state `STATE_TESTING` - Boot into the newly received firmware

If the system is interrupted during the swap operation and reboots, wolfBoot will pick up where it left off and continue the update procedure.

6.4.2.1 Successful boot Upon a successful boot, the application should inform the bootloader by calling `wolfBoot_success()`, after verifying that the system is up and running again. This operation confirms the update to a new firmware.

Failing to set the BOOT partition to STATE_SUCCESS before the next reboot triggers a roll-back operation. Roll-back is initiated by the bootloader by triggering a new update, this time starting from the backup copy of the original (pre-update) firmware, which is now stored in the UPDATE partition due to the swap occurring earlier.

6.4.2.2 Building a new firmware image Firmware images are position-dependent, and can only boot from the origin of the **BOOT** partition in FLASH. This design constraint implies that the chosen

firmware is always stored in the **BOOT** partition, and wolfBoot is responsible for pre-validating an update image and copy it to the correct address.

All the firmware images must therefore have their entry point set to the address corresponding to the beginning of the **BOOT** partition, plus an offset of 256 Bytes to account for the image header.

Once the firmware is compiled and linked, it must be signed using the `sign` tool. The tool produces a signed image that can be transferred to the target using a secure connection, using the same key corresponding to the public key currently used for verification.

The tool also adds all the required Tags to the image header, containing the signatures and the SHA256 hash of the firmware.

6.4.2.3 Self-update wolfBoot can update itself if `RAM_CODE` is set. This procedure operates almost the same as firmware update with a few key differences. The header of the update is marked as a bootloader update (use `--wolfboot-update` for the sign tools).

The new signed wolfBoot image is loaded into the UPDATE parition and triggered the same as a firmware update. Instead of performing a swap, after the image is validated and signature verified, the bootloader is erased and the new image is written to flash. This operation is *not* safe from interruption. Interruption will prevent the device from rebooting.

wolfBoot can be used to deploy new bootloader versions as well as update keys.

6.4.2.4 Incremental updates (aka: 'delta' updates) wolfBoot supports incremental updates, based on a specific older version. The sign tool can create a small "patch" that only contains the binary difference between the version currently running on the target and the update package. This reduces the size of the image to be transferred to the target, while keeping the same level of security through public key verification, and integrity due to the repeated check (on the patch and the resulting image).

The format of the patch is based on the mechanism suggested by Bentley/McIlroy, which is particularly effective to generate small binary patches. This is useful to minimize time and resources needed to transfer, authenticate and install updates.

6.4.2.4.1 How it works As an alternative to transferring the entire firmware image, the key tools create a binary diff between a base version previously uploaded and the new updated image.

The resulting bundle (delta update) contains the information to derive the content of version '2' of the firmware, starting from the base version, that is currently running on the target (version '1' in this example), and the reverse patch to downgrade version '2' back to version '1' if something goes wrong running the new version.

On the device side, wolfboot will recognize and verify the authenticity of the delta update before applying the patch to the current firmware. The new firmware is rebuilt in place, replacing the content of the BOOT partition according to the indication in the (authenticated) 'delta update' bundle.

6.4.2.4.2 Two-steps verification Binary patches are created by comparing signed firmware images. wolfBoot verifies that the patch is applied correctly by checking for the integrity and the authenticity of the resulting image after the patch.

The delta update bundle itself, containing the patches, is prefixed with a manifest header describing the details for the patch, and signed like a normal full update bundle.

This means that wolfBoot will apply two levels of authentication: the first one when the delta bundle is processed (e.g. when an update is triggered), and the second one every time a patch is applied, or reversed, to validate the firmware image before booting.

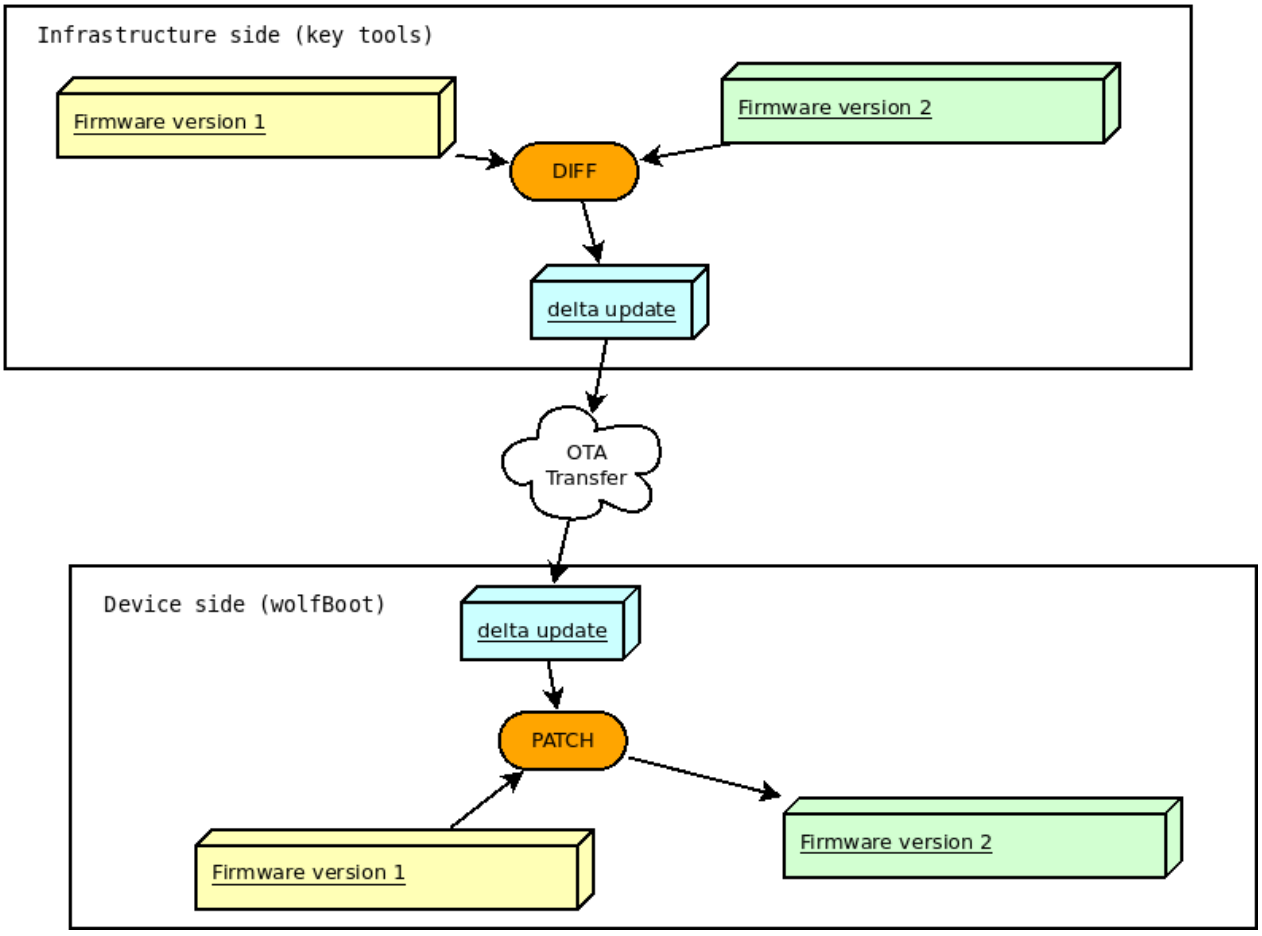

Figure 5: Delta update

These steps are performed automatically by the key tools when using the `--delta` option, as described in the example.

6.4.2.4.3 Confirming the update From the application perspective, nothing changes from the normal, 'full' update case. Application must still call `wolfBoot_success()` on the first boot with the updated version to ensure that the update is confirmed.

Failing to confirm the success of the update will cause wolfBoot to revert the patch applied during the update. The 'delta update' bundle also contains a reverse patch, which can revert the update and restore the base version of the firmware.

The diagram below shows the authentication steps and the diff/patch process in both directions (update and roll-back for missed confirmation).

6.4.2.4.4 Incremental update: example Requirement: wolfBoot is compiled with `DELTA_UPDATES=1`

Version "1" is signed as usual, as a standalone image:

```
tools/keytools/sign.py --ecc256 --sha256 test-app/image.bin wolfboot_signing_private_key.der
1
```

When updating from version 1 to version 2, you can invoke the sign tool as:

```
tools/keytools/sign.py --delta test-app/image_v1_signed.bin --ecc256 --sha256
test-app/image.bin wolfboot_signing_private_key.der 2
```

Besides the usual output file `image_v2_signed.bin`, the sign tool creates an additional `image_v2_signed_diff.bin` which should be noticeably smaller in size as long as the two binary files contain overlapping areas.

This is the delta update bundle, a signed package containing the patches for updating version 1 to version 2, and to roll back to version 1 if needed, after the first patch has been applied.

The delta bundle `image_v2_signed_diff.bin` can be now transferred to the update partition on the target like a full update image.

At next reboot, wolfBoot recognizes the incremental update, checks the integrity, the authenticity and the versions of the patch. If all checks succeed, the new version is installed by applying the patch on the current firmware image.

If the update is not confirmed, at the next reboot wolfBoot will restore the original base `image_v1_signed.bin`, using the reverse patch contained in the delta update bundle.

6.5 Remote External flash memory support via UART

wolfBoot can emulate external partitions using UART communication with a neighbor system. This feature is particularly useful in those asynchronous multi-process architectures, where updates can be stored with the assistance of an external processing unit.

6.5.1 Bootloader setup

The option to activate this feature is `UART_FLASH=1`. This configuration option depends on the external flash API, which means that the option `EXT_FLASH=1` is also mandatory to compile the bootloader.

The HAL of the target system must be expanded to include a simple UART driver, that will be used by the bootloader to access the content of the remote flash using one of the UART controllers on board.

Example UART drivers for a few of the supported platforms can be found in the `hal/uart` directory.

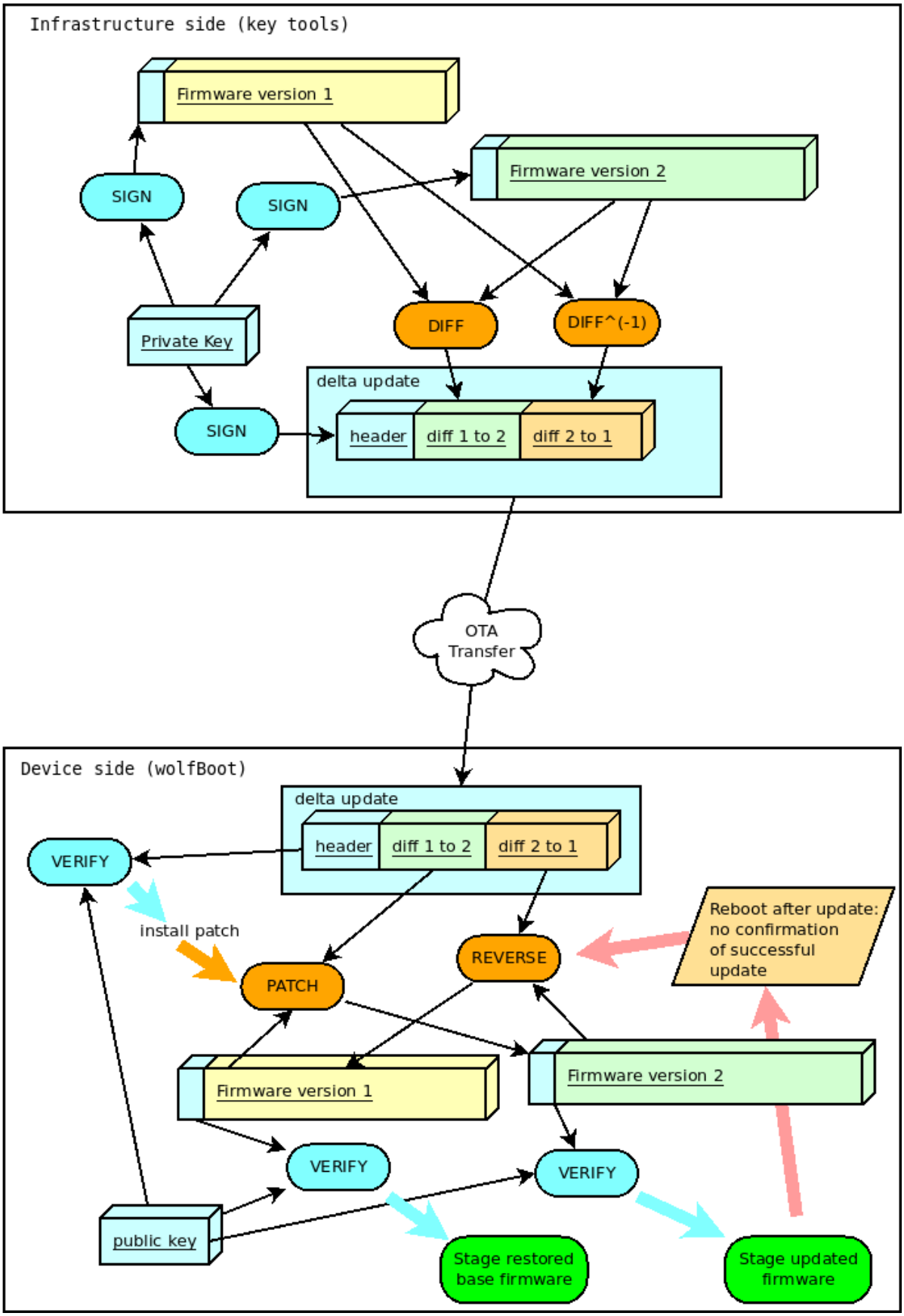

Figure 6: Delta update: details

The API exposed by the UART HAL extension for the supported targets is composed by the following functions:

```
int uart_init(uint32_t bitrate, uint8_t data, char parity, uint8_t stop);
int uart_tx(const uint8_t c);
int uart_rx(uint8_t *c);
```

Consider implementing these three functions based on the provided examples if you want to use external flash memory support on your platform, if not officially supported yet.

6.5.2 Host side: UART flash server

On the remote system hosting the external partition image for the target, a simple protocol can be implemented on top of UART messages to serve flash-access specific calls.

An example uart-flash-server daemon, designed to run on a GNU/Linux host and emulate the external partition with a local file on the filesystem, is available in tools/uart-flash-server.

6.5.3 External flash update mechanism

wolfBoot treats external UPDATE and SWAP partitions in the same way as when they are mapped on a local SPI flash. Read and write operations are simply translated into remote procedure calls via UART, that can be interpreted by the remote application and provide read and write access to actual storage elements which would only be accessible by the host.

This means that after a successful update, a copy of the previous firmware will be stored in the remote partition to provide exactly the same update mechanism that is available in all the other use cases. The only difference consist in the way of accessing the physical storage area, but all the mechanisms at a higher level stay the same.

6.6 Encrypted external partitions

wolfBoot offers the possibility to encrypt the content of the entire UPDATE partition, by using a pre-shared symmetric key which can be temporarily stored in a safer non-volatile memory area.

SWAP partition is also temporarily encrypted using the same key, so a dump of the external flash won't reveal any content of the firmware update packages.

6.6.1 Rationale

Encryption of external partition works at the level of the external flash interface.

All write calls to external partitions from the bootloader perform an additional encryption step to hide the actual content of the external non-volatile memory.

Viceversa, all read operations will decrypt the data stored when the feature is enabled.

An extra option is provided to the `sign.py` sign tool to encrypt the firmware update after signing it, so that it can be stored as is in the external memory by the application, and will be decrypted by the bootloader in order to verify the update and begin the installation.

6.6.2 Temporary key storage

By default, wolfBoot will store the pre-shared symmetric key used for encryption in a temporary area on the internal flash. This allows read-out protections to be used to hide the temporary key.

Alternatively, more secure mechanisms are available to store the temporary key in a different key storage (e.g. using a hardware security module or a TPM device).

The temporary key can be set at run time by the application, and will be used exactly once by the bootloader to verify and install the next update. The key can be for example received from a back-end during the update process using secure communication, and set by the application, using `libwolfboot` API, to be used by wolfBoot upon next boot.

Aside from setting the temporary key, the update mechanism remains the same for distributing, uploading and installing firmware updates through wolfBoot.

6.6.3 Libwolfboot API

The API to communicate with the bootloader from the application is expanded when this feature is enabled, to allow setting a temporary key to process the next update.

The functions

```
int wolfBoot_set_encrypt_key(const uint8_t *key, const uint8_t *nonce);
int wolfBoot_erase_encrypt_key(void);
```

can be used to set a temporary encryption key for the external partition, or erase a key previously set, respectively.

Moreover, using `libwolfboot` to access the external flash with wolfboot hal from the application will not use encryption. This way the received update, already encrypted at origin, can be stored in the external memory unchanged, and retrieved in its encrypted format, e.g. to verify that the transfer has been successful before reboot.

6.6.4 Symmetric encryption algorithms

Encryption can be enabled in wolfBoot using `ENCRYPT=1`.

The default algorithm used to encrypt and decrypt data in external partitions is Chacha20-256. AES-128 and AES-256 options are also available and can be selected using `ENCRYPT_WITH_AES128=1` or `ENCRYPT_WITH_AES256=1`

6.6.5 Chacha20-256

When ChaCha20 is selected:

- The key provided to `wolfBoot_set_encrypt_key()` must be exactly 32 Bytes long.
- The nonce argument must be a 96-bit (12 Bytes) randomly generated buffer, to be used as IV for encryption and decryption.

6.6.5.1 Example usage with ChaCha20-256 The `sign.py` tool can sign and encrypt the image with a single command. The encryption secret is provided in a binary file that should contain a concatenation of a 32B ChaCha-256 key and a 12B nonce.

In the examples provided, the test application uses the following parameters:

```
key = "0123456789abcdef0123456789abcdef"
nonce = "0123456789ab"
```

So it is easy to prepare the encryption secret in the test scripts or from the command line using:

```
echo -n "0123456789abcdef0123456789abcdef0123456789ab" > enc_key.der
```

The `sign.py` script can now be invoked to produce a signed+encrypted image, by using the extra argument `--encrypt` followed by the secret file:

```
./tools/keytools/sign.py --encrypt enc_key.der test-app/image.bin
   wolfboot_signing_private_key.der 24
```

which will produce as output the file `test-app/image_v24_signed_and_encrypted.bin`, that can be transferred to the target's external device.

6.6.6 AES-CTR

AES is used in CTR mode. When AES is selected: - The key provided to `wolfBoot_set_encrypt_key()` must be 16 Bytes (AES128) or 32 Bytes (AES256) long. - The nonce argument is a 128-bit (16 Byyes) randomly generated buffer, used as initial counter for encryption and decryption.

6.6.6.1 Example usage with AES-256 In case of AES-256, the encryption secret is provided in a binary file that should contain a concatenation of a 32B key and a 16B IV.

In the examples provided, the test application uses the following parameters:

```
key = "0123456789abcdef0123456789abcdef"
iv = "0123456789abcdef"
```

So it is easy to prepare the encryption secret in the test scripts or from the command line using:

```
echo -n "0123456789abcdef0123456789abcdef0123456789abcdef" > enc_key.der
```

The `sign.py` script can now be invoked to produce a signed+encrypted image, by using the extra argument `--encrypt` followed by the secret file. To select AES-256, use the `--aes256` option.

```
./tools/keytools/sign.py --aes256 --encrypt enc_key.der test-app/image.bin
   wolfboot_signing_private_key.der 24
```

which will produce as output the file `test-app/image_v24_signed_and_encrypted.bin`, that can be transferred to the target's external device.

6.6.7 API usage in the application

When transferring the image, the application can still use the libwolfboot API functions to store the encrypted firmware. When called from the application, the function `ext_flash_write` will store the payload unencrypted.

In order to trigger an update, before calling `wolfBoot_update_trigger` it is necessary to set the temporary key used by the bootloader by calling `wolfBoot_set_encrypt_key`.

An example of encrypted update trigger can be found in the stm32wb test application source code (in `../test-app/app_stm32wb.c`).

6.7 Application interface for interactions with the bootloader

wolfBoot offers a small interface to interact with the images stored in the partition, explicitly initiate an update and confirm the success of a previously scheduled update.

6.7.1 Compiling and linking with libwolfboot

An application that requires interactions with wolfBoot must include the header file:

```
#include <wolfboot/wolfboot.h>
```

This exports the API function declarations, and the predefined values for the flags and tags stored together with the firmware images in the two partitions.

For more information about flash partitions, flags and states see Flash partitions.

6.7.2 API

libwolfboot provides low-level access interface to flash partition states. The state of each partition can be retrieved and altered by the application.

Basic interaction from the application is provided via the following high-level function calls:

```
uint32_t wolfBoot_get_image_version(uint8_t part)
```

```
void wolfBoot_update_trigger(void)
```

```
void wolfBoot_success(void)
```

6.7.2.1 Firmware version Current (boot) firmware and update firmware versions can be retrieved from the application using:

```
uint32_t wolfBoot_get_image_version(uint8_t part)
```

Or via the shortcut macros:

```
wolfBoot_current_firmware_version()
```

and

```
wolfBoot_update_firmware_version()
```

6.7.2.2 Trigger an update

- `wolfBoot_update_trigger()` is used to trigger an update upon the next reboot, and it is normally used by an update application that has retrieved a new version of the running firmware, and has stored it in the UPDATE partition on the flash. This function will set the state of the UPDATE partition to `STATE_UPDATING`, instructing the bootloader to perform the update upon the next execution (after reboot).

wolfBoot update process swaps the contents of the UPDATE and the BOOT partitions, using a temporary single-block SWAP space.

6.7.2.3 Confirm current image

- `wolfBoot_success()` indicates a successful boot of a new firmware. This can be called by the application at any time, but it will only be effective to mark the current firmware (in the BOOT partition) with the state `STATE_SUCCESS`, indicating that no roll-back is required. An application should typically call `wolfBoot_success()` only after verifying that the basic system features are up and running, including the possibility to retrieve a new firmware for the next upgrade.

If after an upgrade and reboot wolfBoot detects that the active firmware is still in `STATE_TESTING` state, it means that a successful boot has not been confirmed for the application, and will attempt to revert the update by swapping the two images again.

For more information about the update process, see Firmware Update

For the image format, see Firmware Image

7 Integrating wolfBoot in an existing project

7.1 Required steps

- See the Targets chapter for reference implementation examples.
- Provide a HAL implementation for the target platform (see Hardware Abstraction Layer)
- Decide a flash partition strategy and modify `include/target.h` accordingly (see Flash partitions)
- Change the entry point of the firmware image to account for bootloader presence
- Equip the application with the wolfBoot library to interact with the bootloader
- Configure and compile a bootable image with a single "make" command
- For help signing firmware see wolfBoot Signing
- For enabling measured boot see wolfBoot measured boot

7.2 Examples provided

Additional examples available on our GitHub wolfBoot-examples repository here.

The following steps are automated in the default `Makefile` target, using the baremetal test application as an example to create the factory image. By running `make`, the build system will:

- Create a Ed25519 Key-pair using the keygen tool
- Compile the bootloader. The public key generated in the step above is included in the build
- Compile the firmware image from the test application in the 'test_app' directory
- Re-link the firmware to change the entry-point to the start address of the primary partition
- Sign the firmware image using the `sign` tool
- Create a factory image by concatenating the bootloader and the firmware image

The factory image can be flashed to the target device. It contains the bootloader and the signed initial firmware at the specified address on the flash.

The `sign.py` tool transforms a bootable firmware image to comply with the firmware image format required by the bootloader.

For detailed information about the firmware image format, see Firmware image

For detailed information about the configuration options for the target system, see Compiling wolfBoot

7.3 Upgrading the firmware

- Compile the new firmware image, and link it so that its entry point is at the start address of the primary partition
- Sign the firmware using the `sign.py` tool and the private key generated for the factory image
- Transfer the image using a secure connection, and store it to the secondary firmware slot
- Trigger the image swap using libwolfboot `wolfBoot_update_trigger()` function. See wolfBoot library API for a description of the operation
- Reboot to let the bootloader begin the image swap
- Confirm the success of the update using libwolfboot `wolfBoot_success()` function. See wolfBoot library API for a description of the operation

For more detailed information about firmware update implementation, see Firmware Update

8 Troubleshooting

8.1 Python errors when signing a key

```
Traceback (most recent call last):
  File "tools/keytools/keygen.py", line 135, in <module>
    rsa = ciphers.RsaPrivate.make_key(2048)
AttributeError: type object 'RsaPrivate' has no attribute 'make_key'

Traceback (most recent call last):
  File "tools/keytools/sign.py", line 189, in <module>
    r, s = ecc.sign_raw(digest)
AttributeError: 'EccPrivate' object has no attribute 'sign_raw'
```

You need to install the latest wolfcrypt-py here: https://github.com/wolfSSL/wolfcrypt-py

Use `pip3 install wolfcrypt`.

Or to install based on a local wolfSSL installation use:

```
cd wolfssl
./configure --enable-keygen --enable-rsa --enable-ecc --enable-ed25519 --
    enable-des3 CFLAGS="-DFP_MAX_BITS=8192 -DWOLFSSL_PUBLIC_MP"
make
sudo make install
cd wolfcrypt-py
USE_LOCAL_WOLFSSL=/usr/local pip3 install .
```

8.2 Python errors in command line parser running keygen.py

```
Traceback (most recent call last):
  File "tools/keytools/keygen.py", line 173, in <module>
    parser.add_argument('-i', dest='pubfile', nargs='+', action='extend')
  File "/usr/lib/python3.7/argparse.py", line 1361, in add_argument
    raise ValueError('unknown action "%s"' % (action_class,))
ValueError: unknown action "extend"
```

The version of the python interpreter installed on the system is too old. To run `keygen.py` you need to upgrade python to v.3.8 or greater.

8.3 Contact support

If you run into problems and need help, contact us at *support@wolfssl.com*