



Apertis secure boot

1	Contents	
2	Boot sequence	3
3	Secure boot sequence	4
4	Threat models	5
5	offline attacks	5
6	online attacks	5
7	Signing and signing infrastructure	6
8	Apertis secure boot integration	7
9	Apertis secure boot implementation steps	7
10	SabreLite secure boot preparation	7
11	Secure boot in the U-Boot package for Sabrelite	9
12	Sign U-Boot bootloader such that the ROM can verify	10
13	Sign U-Boot bootloader for loading via USB serial downloader	11
14	Sign kernel images for U-Boot to load	11
15	FIT image creation	12
16	Signing the FIT image	13
17	Signing bootloader and kernel from the image build pipeline	14
18	U-Boot signing	15
19	FIT image creation and signing	16
20	As next steps the following could be undertaken:	16
21	For both privacy and security reasons it is important for modern devices to	
22	ensure that the software running on the device hasn't been tampered with. In	
23	particular any tampering with software early in the boot sequence will be hard	
24	to detect later while having a big amount of control over the system. To solve	
25	this issues various vendors and consortiums have created technologies to combat	
26	this, known under names as "secure boot", "highly assured boot"(NXP), "verified	
27	boot"(Google Android/ChromeOS).	
28	While the scope and implementation details of these technologies differs the	
29	approach to provide a trusted boot chain tends to be similar between all of	
30	them. This document discusses how that aspect of the various technologies	
31	works on a high-level and how this can be introduced into Apertis.	

32 Boot sequence

33 To understand how secure boot works first one has to understand how booting
34 works. From a high-level perspective a CPU is a very simple beast, it needs
35 to be pointed at a stream of instructions (code) which it will then be able to
36 execute. Without instructions a CPU cannot do anything. The instructions also
37 need to be in a region of memory which the CPU can access. However when a
38 device is powered on the code that is meant to be run on it (e.g. Linux) will not
39 be in memory yet. To make matters worse on power on main memory (Dynamic
40 RAM) will not even be accessible by the CPU yet! To solve this problem some
41 bootstrapping is required, typically referred to as booting the system.

42 The very first step in the boot process after power on is to get the CPU to start
43 executing some instructions. As the CPU cannot load instructions without
44 running instructions these first instructions are hardwired into the SoC directly
45 with the CPU is hardwired to start executing those when powers comes on. This
46 hardwired piece of code is often referred to as the ROM or romcode.

47 The job of the romcode is to do very basic SoC setup and load further code
48 to execute. To allow the romcode to do its job, it will have access to a small
49 amount of static RAM (SRAM, typically 64 to 128 kilobyte). The locations from
50 where the ROM code can load is system specific. On most modern ARM-based
51 systems this will include at least (SPI-connected) flash (NAND/NOR), eMMC
52 cards, SD cards, serial ports etc. Most systems can only have code loaded over
53 USB initially while some can even load code directly over the network via bootp!.
54 The details of the format the code needs to be in (e.g. specific headers), how
55 the code is presented (e.g. specific offsets on the eMMC) is very system specific.
56 Once romcode managed to load the code from one of its supported location into
57 SRAM execution of that code will start, which will the first time user supplied
58 code is actually ran on the device.

59 This next step is known under various different names such as Boot Loader stage
60 1(BL1), Secondary Program Loader(SPL), Tertiary Program Loader(TPL), etc.
61 The code for this stage must be quite small as only SRAM is available at this
62 stage. The goal for this step is normally to initialize Dynamic RAM (e.g. run
63 DDR memory training) followed by loading the next step into DRAM and ex-
64 ecuting it (which can be far bigger now that DRAM is available). Depending
65 on the system this stage may also provide initial user feedback that the system
66 is booting (e.g. display a first splash image, turning an LED on etc), but that
67 purely depends on the overall system design and available space.

68 What the next step of executed code is more system specific. In some cases it can
69 directly be Linux, in some cases it will be a bootloader with more functionality
70 (as all of main memory is now available) and in some cases it will be multiple
71 loader steps. As an example of the last case for devices using ARM Trusted
72 Firmware there will typically be follow-on steps to load the secure firmware (such

73 as [OP-TEE](#)¹) followed by a non-secure world bootloader which loads Linux. For
74 those interested the various images used in an ATF setup can be found [here](#)².

75 Linux starting up typically is the last phase of the boot process. For Linux to
76 start the previous stage will have loaded a kernel image, optionally a initramfs
77 and optionally a devicetree into main memory. The combination of these will
78 load the root filesystem at which point userspace (e.g. applications) will start
79 running.

80 Note that while the above is a simple view on the basic boot process, the overall
81 flow will be the same on all systems (both ARM and non-ARM devices). For
82 the above we also implicitly assumed that only one CPU is booted, for some
83 more complex systems multiple CPUs (e.g. main application processors and
84 various co-processors) might be booted. It may even be the case that all the
85 early stages are done by a co-processor which takes care of loading the first code
86 and starting the main processor. The overall description is also valid for system
87 with hypervisors, essentially the hypervisor is just another stage in the boot
88 sequence and will load/start the code for each of the cells it runs.

89 For this document we'll only look at securing the booting of the main (Linux
90 running) processor without a hypervisor.

91 Secure boot sequence

92 The main objective for a secure boot process is to ensure all code that gets
93 executed by the processor is trusted. As each of the stages described in the
94 previous section is responsible for loading the code for the next stage the solution
95 for that is relatively straight-forward. Apart from loading the next stage of code,
96 each stage also needs to verify the code it has loaded. Typically this is done by
97 some signature verification mechanism.

98 The ROM step is normally assumed to be fully trusted as it's hard-wired into the
99 SoC and cannot be replaced. How the ROM is configured and how it validates
100 the next stage is highly device specific. Later steps can do the verification either
101 by calling back into ROM code (thus re-using the same mechanisms as the ROM)
102 or by pure software implementation (making it more consistent between different
103 devices).

104 In all cases to support this, apart from device specific configuration, all boot
105 stages need to be appropriately signed. Luckily this is typically based on stan-
106 dard mechanisms such as RSA keys and X.509 Certificates.

107 Once Linux starts the approach has to be different as it's not feasible in most
108 systems to fully verify all of the root filesystem at boot time as this simply

¹<https://www.apertis.org/concepts/distribution/op-tee/>

²https://trustedfirmware-a.readthedocs.io/en/latest/getting_started/image-terminology.html

109 would take far too long. As such the form of protection described thus far only
110 gets applied up to the point the Linux kernel starts loading.

111 **Threat models**

112 To understand what a secure boot system really secures it's important to look
113 at the related threat models. As a first step we can distinguish between offline
114 (device is turned off) and online attacks (device powered on).

115 For these considerations the assumption is made all boot steps work as intended.
116 As with any software security vulnerabilities can invalidate the protection given.
117 While in most cases these can be patches as issues become known, for ROM
118 code this is impossible without a hardware change.

119 **offline attacks**

- 120 • Attack: Replace any of the boot stages on device storage (physical access
121 required)
- 122 • Impact: Depending on the boot stage the attacker can get full control of
123 the device for each following boot.
- 124 • Mitigation: Assuming each stage correctly validates the next boot stage,
125 any tampering with loaded code will be detected and prevented (e.g. de-
126 vice fails to boot).
- 127 • Attack: Trigger the device to load software from external means (e.g. USB
128 or serial) under the attackers control.
- 129 • Impact: Depending on the boot stage the attacker can get full control of
130 the device.
- 131 • Mitigation: The ROM or any stage that loads from an external source
132 should use the same verification as for any on device stages. However for
133 production use, if possible, loading software from external source should
134 be disabled.
- 135 • Attack: Replace or add binaries on the systems root filesystem
- 136 • Impact: Full control of the device as far as the kernel allows.
- 137 • Mitigation: No protection from the above mechanisms.

138 **online attacks**

- 139 • Attack: Gain enough access to replace any of the boot stages on device
140 storage
- 141 • Impact: Depending on the boot stage the attacker can get full control of
142 the device for each following boot.

- 143 • Mitigation: Assuming each stage correctly validates the next boot stage,
144 any tampering with loaded code will be detected and prevented (e.g. de-
145 vice fails to boot).
- 146 • Attack: Replace or add binaries on the systems root filesystem
- 147 • Impact: Full control of the device as far as the kernel allows.
- 148 • Mitigation: No protection from the above mechanisms.

149 Signing and signing infrastructure

150 To securely boot a device it is assumed all the various boot stages have some
151 kind of signature which can be validate by previous stages. Which by extension
152 also means the protection is only as strong as the signature; if an attacker can
153 sign code under their control with a signature that is valid (or seen as valid)
154 for the verifying step all protection is lost. This means that special care has to
155 be taken with respect to key handling to ensure signing keys are kept with the
156 right amount of security depending on their intended use.

157 For development usage and devices a low amount of security is ok in most cases,
158 the intention in the development stage is for developers to be easily able to run
159 their own code and by extension should be able to sign their own builds with
160 minimal effort.

161 For production devices however the requirements should be much more strict
162 as unauthorized of control of a signing key can allow attackers to defeat the
163 intended protection by secure boot. Furthermore production devices should
164 typically not be allowed to run development builds as those tend to enable
165 extra access for debugging and development reasons which tend to be a great
166 attack vector.

167 For these reason it's recommendable to have at least two different sets of signing
168 keys, one for development usage and one for production use. Development keys
169 can be kept with low security or even be publicly available, while production
170 keys should *only* be used to sign final production images and managed by a
171 hardware security module (HSM) for secure storage. To allow the usage of a
172 commercially available HSMs it's recommended for the signing process to be
173 able to support the [PKCS#11 standard](https://en.wikipedia.org/wiki/PKCS_11)³.

174 Note that in case security keys do get lost/stolen/etc it is possible for some
175 devices to revoke or update the valid set of keys. However this can be quite
176 limited e.g. on i.MX6 device one can *one-time* program up to four acceptable
177 keys and each of those can be flagged as revoked, but it's impossible to add
178 more or replace any keys.

³https://en.wikipedia.org/wiki/PKCS_11

179 Apertis secure boot integration

180 Integrating secure boot into Apertis really exists out of two parts. The first
181 part is to ensure all boot stages have the ability to verify. The second part is to
182 be able to sign all the boot stages as part of the Apertis image building process.
183 While the actual implementation details of both will be system/hardware/SoC
184 specific the impact of this is generic for all.

185 As Apertis images are composed out of pre-build binary packages the package
186 delivering the implementation for the various boot stages should either provide
187 a build which will always enforce signature verification *or* the implementation
188 should detect if the device is configured for secure boot and only enforce in that
189 situation. Enforcing on demand has the benefit that it makes it easier to test
190 the same builds on non-secure devices (though care must be taken that secure
191 boot status cannot be faked).

192 For the signing of the various stages this needs to be done at image build time
193 such that the signing key can be chosen based on the target. For example
194 whether it's a final production build or a development build or even a production
195 build to test on development devices. This in turn means that the signing tools
196 and implementation need to support signing outside the build process which is
197 normally supported.

198 Apertis secure boot implementation steps

199 As the whole process is somewhat device specific implementation of a secure
200 boot flow for Apertis should be done on a device per device basis. The best
201 starting point is is most likely the NXP i.MX6 sabrelite reference board as the
202 secure boot process (Highly Assured Boot in NXP terms) is both well-known
203 and well supported by upstream components. Furthermore an initial PoC for
204 the early boot stages was already done for the NXP Sabre Auto boards which
205 are based on the same SoC.

206 SabreLite secure boot preparation

207 The [good introduction into HAB \(High Assurance Boot\)](#)⁴ is prepared by Bound-
208 ary Devices, also there are some [documentation](#)⁵ and examples in U-Boot source
209 tree.

210 The [NXP Code Signing Tool](#)⁶ is needed to create keys, certificates and SRK
211 hashes used during the signing process –please refer to [section 3.1.3 of CST User'](#)

⁴<https://boundarydevices.com/high-assurance-boot-hab-dummies/>

⁵https://github.com/u-boot/u-boot/blob/master/doc/imx/habv4/introduction_habv4.txt

⁶<https://gitlab.apertis.org/pkg/imx-code-signing-tool>

212 s Guide⁷. Apertis reference images use the [public git repository](#)⁸ with all secrets
213 available, so it could be used for signing binaries during development in case if
214 board has been fused with Apertis SRK hash (**irreversible operation!!!**).

215 **Caution:** the SabreLite board can be fused with the SRK (Super Root Key)
216 hash only once!

217 To fuse the [Apertis SRK hash](#)⁹ we have to have the hexadecimal dump of the
218 hash of the key. Command below will produce the output with commands for
219 Apertis SRK hash fusing:

```
220 $ hexdump -e '/4 "0x"' -e '/4 "%X""\n"' SRK_1_2_3_4_fuse.bin | for i in `seq 0 7`; do read h; echo fuse prog -  
221 y 3 $i $h; done
```

222 This command generates the list of commands to be executed in a U-Boot CLI.
223 For Apertis SRK hash fusing they are:

```
224 fuse prog -y 3 0 0xFD415383  
225 fuse prog -y 3 1 0x519690F5  
226 fuse prog -y 3 2 0xE844EB48  
227 fuse prog -y 3 3 0x179B1826  
228 fuse prog -y 3 4 0xEC0F8D7C  
229 fuse prog -y 3 5 0x2F209598  
230 fuse prog -y 3 6 0x9A98BE3  
231 fuse prog -y 3 7 0xAAD9B3D6
```

232 After execution of commands above only [Apertis development keys](#)¹⁰ can be
233 used for signing the U-Boot binary.

234 The i.MX6 ROM does signature verification of the bootloader during startup,
235 and depending on the configured (fused) mode the behaviour is different. The
236 i.MX6 device may work in 2 modes:

- 237 • “open”—the HAB ROM allows the use of unsigned bootloaders or bootload-
238 ers signed with any key, without checking its validity. In case of errors, it
239 will only generate HAB secure events on boot without halting the process.
240 This mode is useful for development.
- 241 • “closed”—only signed with correct key U-Boot may be started, any incor-
242 rectly signed bootloader will not be started. This mode should be used
243 only for final product.

244 **It is highly recommended not to use “closed” mode for development**
245 **boards!**

⁷https://gitlab.apertis.org/pkg/imx-code-signing-tool/-/blob/apertis/v2021dev2/docs/CST_UG.pdf

⁸<https://gitlab.apertis.org/infrastructure/apertis-imx-srk>

⁹https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/SRK_1_2_3_4_fuse.bin

¹⁰<https://gitlab.apertis.org/infrastructure/apertis-imx-srk/>

246 To check if your device is booted with correctly signed bootloader, and SRK key
247 is fused, just type this in the U-Boot CLI:

```
248 => hab_status
249
250 Secure boot enabled
251
252 HAB Configuration: 0xcc, HAB State: 0x99
253 No HAB Events Found!
```

254 The output shows if the device is in “closed”mode (secure boot enabled) and
255 booted without any security errors.

256 In case of errors in “open”mode the same command will show the list of HAB
257 events similar to:

```
258 ----- HAB Event 5 -----
259 event data:
260     0xdb 0x00 0x14 0x41 0x33 0x21 0xc0 0x00
261     0xbe 0x00 0x0c 0x00 0x03 0x17 0x00 0x00
262     0x00 0x00 0x00 0x50
263
264 STS = HAB_FAILURE (0x33)
265 RSN = HAB_INV_CERTIFICATE (0x21)
266 CTX = HAB_CTX_COMMAND (0xc0)
267 ENG = HAB_ENG_ANY (0x00)
```

268 During Linux kernel verification it is possible to emulate the “closed”mode with
269 fuse override command and proceed with the boot:

```
270 => fuse override 0 6 0x2
271 => run bootcmd
```

272 *Note:* the only issue with closed mode emulation –the device will accept kernel
273 signed with any key, but HAB events will be generated and shown in that case.

274 To close a device you need to fuse the same values used for overriding.

275 **Caution:** the board can only use bootloaders signed with the Apertis develop-
276 ment key after the step below! This is irreversible operation:

```
277 => fuse prog 0 6 0x2
```

278 Secure boot in the U-Boot package for Sabrelite

279 The U-Boot bootloader must be configured with the option `CONFIG_SECURE_BOOT`
280 to enable support of HAB (High Assurance Boot) support on i.MX6 platform.

281 Upstream U-Boot has no protection based on the HAB engine to prevent exe-
282 cuting unsigned binaries. Verified boot with the usage of HAB ROM is enabled

in U-Boot for Apertis only for [FIT \(Flattened uImage Tree\)](#)¹¹ format since it allows to embed Linux kernel, initramfs and DTB into a single image. Hence the support of FIT images must be enabled in U-Boot configuration by option `CONFIG_FIT`.

The [patch series](#)¹² enables verification of FIT image prior to execution of the Linux kernel. Patched U-Boot do verification of the whole FIT binary prior to extraction kernel and initramfs images, and this ensures that only verified initial system will be started.

All other format types like zImage, as well as other boot methods are prohibited on fully secured device when “closed” mode is enabled or emulated.

Sign U-Boot bootloader such that the ROM can verify

To sign the U-Boot for SabreLite we need `cst` tool installed in the system and the [Apertis development keys repository](#)¹³ need to be checked out. Please use the [csf/csf_uboot.txt](#)¹⁴ file as a template for your U-Boot binary.

U-Boot for SabreLite board doesn’t use SPL, hence the whole `u-boot.imx` binary must be signed. With enabled `CONFIG_SECURE_BOOT` option the build log will contain following output (and file `u-boot.imx.log` as well):

```
Image Type:   Freescale IMX Boot Image
Image Ver:    2 (i.MX53/6/7 compatible)
Mode:         DCD
Data Size:    606208 Bytes = 592.00 KiB = 0.58 MiB
Load Address: 177ff420
Entry Point:  17800000
HAB Blocks:   0x177ff400 0x00000000 0x00091c00
DCD Blocks:   0x00910000 0x0000002c 0x00000310
```

we need values from the string started with “HAB Blocks:”. Those values must be used in “[Authenticate Data]” section of [template](#)¹⁵:

```
[Authenticate Data]
    Verification index = 2
    Blocks = 0x177ff400 0x00000000 0x00091C00 "u-boot.imx"
```

To sign the U-Boot with `cst` tool simply call:

¹¹https://github.com/u-boot/u-boot/blob/master/doc/usage/fit/source_file_format.rst

¹²https://gitlab.apertis.org/pkg/u-boot/-/merge_requests/4

¹³<https://gitlab.apertis.org/infrastructure/apertis-imx-srk>

¹⁴https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/csf/csf_uboot.txt

¹⁵https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/csf/csf_uboot.txt

```
315  cst -i csf_uboot.txt -o csf_uboot.bin
```

316 File `csf_uboot.bin` will contain signatures which should be appended to original
317 `u-boot.imx` binary:

```
318  cat u-boot.imx csf_uboot.bin > u-boot.imx.signed
```

319 Sign U-Boot bootloader for loading via USB serial down- 320 loader

321 In case if something goes wrong and the system does not boot anymore it
322 is still possible to boot with the help of [USB serial downloaders](#)¹⁶, such as
323 `imx_usb_loader` OR `uuu`.

324 However the U-Boot must be signed in a slightly different way since some
325 changes are done by ROM in runtime while loading binary. Please refer to
326 section “What about `imx_usb_loader`?” of [High Assurance Boot \(HAB\) for dummies](#)¹⁷
327 document.

328 The template `csf_uboot.txt`¹⁸ for signing U-Boot to be loaded over serial down-
329 loader protocol should contain additional block in “[Authenticate Data]” section:

```
330  [Authenticate Data]
331      Verification index = 2
332      Blocks = 0x177ff400 0x00000000 0x00091C00 "u-boot.imx", \
333              0x00910000 0x0000002c 0x00000310 "u-boot.imx"
```

334 With the help of `mod_4_mfgtool.sh`¹⁹ script we need to store and restore DCD
335 address from original `u-boot.imx` in addition to signing:

```
336  sh mod_4_mfgtool.sh clear_dcd_addr u-boot.imx
337  cst -i csf_uboot.txt -o csf_uboot.bin
338  sh mod_4_mfgtool.sh set_dcd_addr u-boot.imx
339  cat u-boot.imx csf_uboot.bin > u-boot.imx.signed_usb
```

340 Sign kernel images for U-Boot to load

341 After the successful startup of U-Boot we need to load the Linux kernel,
342 `initramfs` and `DTB` file into the memory. All these bits must be verified before
343 transferring control to the kernel. With [FIT \(Flattened uImage Tree\)](#)²⁰ format
344 we can use single signed image with kernel, `initramfs` and `DTB` embedded, and

¹⁶<https://community.nxp.com/docs/DOC-95604>

¹⁷<https://boundarydevices.com/high-assurance-boot-hab-dummies/>

¹⁸https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/csf/csf_uboot.txt

¹⁹https://storage.googleapis.com/boundarydevices.com/mod_4_mfgtool.sh

²⁰https://github.com/u-boot/u-boot/blob/master/doc/uImage.FIT/source_file_format.txt

345 this allows to avoid “mix and match”attacks with mixed versions of kernel,
346 initramfs, DTB and configuration.

347 The signing procedure for kernel images is split into 2 parts:

- 348 • preparation of the kernel image in FIT format
- 349 • sign FIT image

350 FIT image creation

351 [U-Boot documentation](#)²¹ contains a lot of details and examples how to create
352 FIT images for different purposes.

353 To embed all bits into the single FIT image we need to prepare file in image tree
354 source format, for Apertis we use simple [template](#)²² containing configuration
355 with 3 entries for kernel, initramfs and DTB respectively. So values `{{kernel}}`,
356 `{{ramdisk}}` and `{{dtb}}` should be substituted with absolute or relative path to
357 corresponding files.

358 Please pay attention to addresses in `load` fields, since the whole FIT image is
359 loaded into the memory by address `0x12000000` (check the value of `kernel_addr_r`
360 in U-Boot environment), it is important to avoid intersections with embedded
361 binaries since they will be copied to configured memory regions after successful
362 verification.

363 To create the FIT image you need to have `mkimage` command from the package
364 `u-boot-tools` compiled with FIT support. With FIT source file prepared just
365 run `mkimage` and generate the FIT binary:

```
366 $ mkimage -f vmlinuz.its vmlinuz.itb
367 FIT description: Apertis armhf kernel with dtb and initramfs
368 Created:      Fri Mar 13 02:23:33 2020
369 Image 0 (kernel-0)
370   Description: Linux Kernel
371   Created:     Fri Mar 13 02:23:33 2020
372   Type:        Kernel Image
373   Compression: uncompressed
374   Data Size:   4526592 Bytes = 4420.50 KiB = 4.32 MiB
375   Architecture: ARM
376   OS:          Linux
377   Load Address: 0x10800000
378   Entry Point:  0x10800000
379   Hash algo:    sha1
380   Hash value:   8a64994bdab06d01450560ea229c9f44f1f0af14
381 Image 1 (ramdisk-0)
382   Description:  ramdisk
```

²¹<https://github.com/u-boot/u-boot/tree/master/doc/usage/fit>

²²https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/si gn/imx6/fit_image.template

```

383 Created:      Fri Mar 13 02:23:33 2020
384 Type:        RAMDisk Image
385 Compression: uncompressed
386 Data Size:    20285185 Bytes = 19809.75 KiB = 19.35 MiB
387 Architecture: ARM
388 OS:          Linux
389 Load Address: 0x15000000
390 Entry Point:  unavailable
391 Hash algo:    sha1
392 Hash value:   c12652573d1b301b191cf3e2a318913afc1ae4b7
393 Image 2 (fdt-0)
394 Description:  Flattened Device Tree blob
395 Created:      Fri Mar 13 02:23:33 2020
396 Type:        Flat Device Tree
397 Compression: uncompressed
398 Data Size:    42366 Bytes = 41.37 KiB = 0.04 MiB
399 Architecture: ARM
400 Hash algo:    sha1
401 Hash value:   ace0dd1dea00568b1c4e6df3fb0420c912e3e091
402 Default Configuration: 'conf-0'
403 Configuration 0 (conf-0)
404 Description:  Boot Apertis
405 Kernel:      kernel-0
406 Init Ramdisk: ramdisk-0
407 FDT:         fdt-0
408 Hash algo:    sha1
409 Hash value:   unavailable
410 CSF Processed successfully and signed data available in vmlinuz.itb

```

411 Signing the FIT image

412 Now it is time to sign the produced image. The procedure is similar to signing
413 U-Boot with additional step –we need to add the **IVT** (Image Vector Table) for
414 the kernel image. We skip this step for U-Boot since it is prepared automatically
415 during the build of the bootloader.

416 The IVT is needed for the HAB ROM and must be the part of the binary, it
417 should be aligned to 0x1000 boundary. For instance, if the produced binary is:

```

418 $ stat -c "%s" vmlinuz.itb
419 25555173

```

420 we need to pad the file to nearest aligned value, which is 25559040:

```

421 $ objcopy -I binary -O binary --pad-to=25559040 --gap-fill=0x00 vmlinuz-
422 pad.itb

```

423 The next step is IVT generation for the FIT image and the easiest method is

424 to use the [genIVT script](#)²³ provided by Boundary Devices with adaptation for
425 padded FIT image:

- 426 • Jump Location -0x12000000 Here we expect the image will be loaded by
427 U-Boot
- 428 • Self Pointer -0x13860000 (Jump Location + size of padded image) Pointer
429 to the IVT table itself, which will place after padded image
- 430 • CSF Pointer -0x13860020 (Jump Location + size of padded image + size
431 of IVT) Pointer to signature data, which we will add after IVT

432 So, the IVT generation is pretty simple:

```
433 $ perl genIVT
```

434 it will generate the binary named `ivt.bin` to be added to the image:

```
435 $ cat vmlinuz-pad.itb ivt.bin > vmlinuz-pad-ivt.itb
```

436 We need to prepare the config file for signing the padded FIT image with IVT.
437 This step is absolutely the same as for [U-Boot signing](#).

438 Configuration file for FIT image is created from template [csf_uboot.txt](#)²⁴, and
439 values in [Authenticate Data] section must be the same as used for IVT calcula-
440 tion -Jump Location and the size of generated file:

```
441 [Authenticate Data]
442     Verification index = 2
443     # Authenticate Start Address, Offset, Length and file
444     Blocks = 0x12000000 0x00000000 0x1860020 "vmlinuz-pad-ivt.itb"
```

445 At last we are able to sign the prepared FIT image:

```
446 $ cst -i vmlinuz-pad-ivt.csf -o vmlinuz-pad-ivt.bin
447 CSF Processed successfully and signed data available in vmlinuz-pad-ivt.bin
```

448 Signing bootloader and kernel from the image 449 build pipeline

450 Starting with v2021dev1 Apertis uses single signed FIT kernel image for OSTree-
451 based systems. The signed version of U-Boot is a part of U-Boot installer.

452 For signing binaries with the `cst` tool we need some files from the [Apertis devel-](#)
453 [opment keys](#)²⁵ git repository. The minimal working setup should include only
454 6 files:

- 455 • `SRK_1_2_3_4_table.bin` -Super Root Keys table
- 456 • `key_pass.txt` -file with password

²³<https://storage.googleapis.com/boundarydevices.com/genIVT>

²⁴[https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/csf/csf_uboot.t](https://gitlab.apertis.org/infrastructure/apertis-imx-srk/-/blob/master/csf/csf_uboot.txt)
xt




²⁵<https://gitlab.apertis.org/infrastructure/apertis-imx-srk>

- CSF certificate and key in PEM format
- IMG certificate and key in PEM format

In addition we need a template for the FIT source file and CSF template suitable for signing U-Boot and FIT kernel.

All files listed above are added into the git repository inside [sign/imx6](#)²⁶ subdirectory. Since all secrets for Apertis are public we are able to use them directly from the repo. However this is not acceptable for production.

Fortunately the most of CI tools have possibility to add files as secrets available only on several steps. Hence we add “private”keys and password file as “Secret file”global credentials to demonstrate the integration into the Jenkins pipeline:

	Jenkins	(global)	671e0fea-31df-4b05-9668-41014145e87c	CSF1_1_sha256_2048_65537_v3_usr_key.pem (HAB CSF key)
	Jenkins	(global)	0916d363-efe7-4973-9336-5cdcc05a2f89	IMG1_1_sha256_2048_65537_v3_usr_key.pem (HAB img key)
	Jenkins	(global)	dbdc5b91-6ea7-4f9c-b809-306a9696efbd	key_pass.txt (HAB CSF password file)

For keys usage they should be available during the call of `cst` tool, so we have to add into the Jenkins pipeline copying of these secret files with the same names as used in [CSF template](#)²⁷ and remove them after the usage.

For instance the simple secrets copying for Jenkins:

```
withCredentials([ file(credentialsId: csf_csf_key, variable: 'CSF_CSFKEY'),
  file(credentialsId: csf_img_key, variable: 'CSF_IMGKEY'),
  file(credentialsId: csf_key_pass, variable: 'CSF_PASSWD')]) {
  // Setup keys for cst tool from Jenkins secrets
  // Have to keep keys and password file near certificates
  sh(script: """
    cd ${WORKSPACE}/sign/imx6
    cp -af $CSF_CSFKEY ./
    cp -af $CSF_IMGKEY ./
    cp -af $CSF_PASSWD ./""")
}
```

U-Boot signing

To sign the U-Boot the script [scripts/sign-u-boot.sh](#)²⁸ has been added. It automatically generates the CSF configuration from the template [sign/imx6/fit_image_csf.template](#)²⁹ and call the `cst` tool to sign the U-

²⁶<https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/tree/apertis/v2023/sign/imx6>

²⁷https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/sign/imx6/fit_image_csf.template

²⁸<https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/scripts/sign-u-boot.sh>

²⁹https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/sign/imx6/fit_image_csf.template

487 Boot binary.

488 The script is called by the [Debops recipe for the SabreLite U-Boot installer im-](#)
489 [age](#)³⁰:

```
490   - action: run
491     description: Sign U-Boot
492       script: scripts/sign-u-boot.sh "${ROOTDIR}/deb-binaries/usr/lib/u-
493 boot/{{ $target }}/u-boot.imx"
```

494 FIT image creation and signing

495 The FIT image is more complex. So for Apertis we use 2 scripts:

- 496 • the [scripts/generate_signed_fit_image.py](#) [script](#)³¹ is used for generation
497 FIT image, padding, IVT calculation and signing. This script can be used
498 standalone to automate all steps described in the section “[Sign kernel](#)
499 [images for U-Boot to load](#)”
- 500 • the [scripts/generate_fit_image.sh](#) [script](#)³² is a wrapper for the former pro-
501 viding it the paths for kernel, initramfs and DTB to include them in the
502 signed FIT image.

503 The integration with the build pipeline happens **after** the kernel is installed by
504 the [OSTree commit recipe](#)³³ by adding the step below:

```
505   - action: run
506     description: Generate FIT image
507     script: scripts/generate_fit_image.sh
```

508 **NB:** this action must be done prior to ostree commit action to add the signed
509 FIT kernel into OSTree repository for OTA upgrades.

510 As next steps the following could be undertaken:

- 511 • Integration of PKCS#11 support in the signing process to support HSM
512 devices
- 513 • Automated testing of secure boot if possible

³⁰<https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/mx6qsabrelite-uboot-installer.yaml>

³¹https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/scripts/generate_signed_fit_image.py

³²https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/scripts/generate_fit_image.sh

³³<https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2023/ostree-commit.yaml>