Android Verified Boot

Broadcom Proprietary and Confidential

Release Note: a pdf version of this document is embedded inside the avb package released.

avb: Android Verified Boot

All of the actions you may take in this path are irreversible - it is important to read through this page and understand its content before taking those actions; failure to do so may lead to your device being unable to perform the expected feature and|or being left in a state that is difficult to recover from easily.

- Once a device emmc rpmb block has been programmed with a key, you can no longer change that key (OTP programming in effect).
  - if the key is invalid for some reason, or if the key has been assigned before by other means than the one described here, then your device will not be able to provide "avb".
  - similarly, until the rpmb key is programmed and valid, the device will always be considered "avb unlocked".
- Once a device has been otp'ed for secure boot, you will need to always use a signed bolt image for it from that point on regardless of the android (avb) device state and regardless of the image you run on this device: android, linux, linux with nexus.
- As a corollary to the above statement, we also mandate that once you run a signed bolt image, you also run a signed bsu image. This is to ensure bolt+bsu are properly in sync (they will be delivered as a packaged bootloader.img) and it will ensure that you always have a way to recover the "avb" state no matter its current value. Matching a random bsu to a signed bolt is simply asking for problems to happen, so we deny that luxury on purpose.

CONTENTS

Current Supported Version

Unless otherwise specified, we support two versions for each target integration. A stable version and a "pre" version which is typically an increment over the stable version to patch up things needed or introduce new behavior (such as re-sync to a new baseline bolt version).

- android N: **avb-1.0.5**
  - no active development on this branch, but bug fixes can lead to increment.
- android O: **avb-2.0.2.2, avb-2.0.5**
  - **note: avb-2.x.y is the package version provided by broadcom; it has no relation to Android Verified Boot version 2, which is currently not integrated with any reference platforms.**

Link to avb Bootloader Images

This is the internal link to the bootloader.img repository.

- This is where you can find all the device specific bootloader.img images containing signed bolt, signed bsu and signed bl31 (when applicable).
- **The bootloader.img itself is not signed (nor encrypted).**
  - it is simply a container which helps reconcile the various android boot stage(s) image(s) together for a given device.
  - the container format is understood by android bsu alone (fastboot) and it is parsed during the "**fastboot flash bootloader**" command execution.

        - the command execution would parse the container format, split up the various boot stage parts and flash them to their corresponding location.
        - the command (currently) would ***not*** validate that the signed image(s) it contains is(are) correctly signed before flashing them, therefore if you are passing in a bogus bootloader.img content, you may brick your secure boot device.[SWANDROID-3818]
    - all the images within the avb aware bootloader.img container must be signed and are assumed so by the device, there is one exception to this, stated below.
- If a particular product variant or software revision is missing from the repository that you believe should be available on the page, report it.
- To use those images, you should always flash them through fastboot using the '**fastboot flash bootloader**' command.
    - **reminder: on a "avb[locked]" device, you will not be able to flash a bootloader.img until you "avb[unlock]" it first.**

On "secure" Enabled Devices

In some cases, one may be using a device which is enabled for secure boot by default by virtue of it being required for the particular part installed. In this case, you may not be needing to run android verified boot (avb) infrastructure necessarily, but you will need to ensure the bootloader.img image contains a signed bolt since the device will be secure boot opt'ed.

Since the bootloader.img is not signed itself and is a simple container, it is therefore possible to package up a signed bolt with a standard (unsigned) bsu and run this on your secure part. This would not constitute an integration of android verified boot, and therefore is not being discussed further here, it is simply mentioned as a convenience.

To package a bootloader.img for an android device using a signed bolt and unsigned bsu, the following command can be used:

```
bootloaderimg.py bolt.signed.bin bsu.UNSIGNED.elf bootloader.img
```

Target Supported

The avb package is platform agnostic and the code can be built for any platform, However the package is tested and validated only on the following platforms:

- "avko", i.e. 7439SSFFG_NOAVS device.
    - **baseline device for all avb versions.**
- "cypress", i.e. 7271T device.
    - **starting with avb version 1.0.2**.

Before You Begin

Terminology

1. "**secure bootloader**"
    - this is an android bootloader (BOLT+BSU[+BL31]) which is enabled for security support; it can be signed or unsigned, either ways, it contains the necessary support for security actions.
    - in practice, non-signed secure bootloaders images are a very rare species, but may exists for testing purposes or debugging, you will not find them on the linked images however, you would most likely get them personally, should you need one.
2. "**signed bootloader**"
    - this is a "**secure bootloader**" (BOLT+BSU[+BL31]) which has been signed. **All images that form the bootloader have been signed**. After enabling a device in secure mode enforcement through OTP, this is the only way to run bootloader onward for that device.
3. "**secure otp**"
    - this is the OTP bit which controls secure boot from BOLT onward; once programmed, you must use "**signed bootloader**" for the rest of the life of this device.

Definition

avb is the stb integration of the android's verified boot feature described here (https://source.android.com/security/verifiedboot/verified-boot.html).

Full avb integration consists in three main parts:

1. Configure the emmc|rpmb block with a valid key and seed its content (the "avb" state).
    - this action of seeding the rpmb block key is a **one time only**.
        - the action will fail if for some reason the key has been programmed already prior by any other means than the one provided with the secure bootloader.
        - there is no recovery possible from a miss-programmed rpmb key, in particular, it will not be possible to write data to the rpmb block.
    - the rpmb block, in the context of avb, is used to keep the avb run time update-able boot state; while the rpmb in general could be used also for other usage by other parts of the device (trustedOS, sage, etc..) which are out of scope of this document, although need to be aware of the use made by the bootloader.
        - **the bootloader reserves the first four (4) sectors of the rpmb block**.
    - without this functionality, the device **is not** avb capable.
    - to avoid any miss-handling, a single command is provided to generate the rpmb key for a device and program it on such device without having to manipulate the key at any point in time.
        - while the command can be run many times over, its actual action would only 'apply' once at most.
        - once programmed, an OTP is set to prevent key leakage, so you will not be able to determine the key used for the rpmb block, which in practice is not necessary to have, nor should it be possible to find for obvious security reasons.

        - the above implies that once a particular soc is associated to a emmc|rpmb, **this association is unique and cannot be broken** (and incidentally cannot be tampered with)... in particular, if for some reason either the soc or the emmc need to be replaced on a device, the following should be considered:
            - replacing a soc means you must also replace the emmc as the current emmc|rpmb would have been associated with the prior soc and the rpmb key can no longer be changed to associate the current emmc|rpmb with the new soc.
            - replacing the emmc means you must also replace the soc as the new emmc|rpmb would not be able to be properly programmed by the existing soc since the key can no longer be extracted out of the soc bsp.
2. Secure boot otp, which is the traditional way to force bootloader (bolt) validation.
    - this setting is non reversible once applied, unlike the avb state.
    - without this setting, the device **is not** avb capable.
    - if by inadvertence you flash a bolt image which is either not signed, or not signed properly for the device (as example flashing an image signed for a ZB part on a ZS|ZD part), the device will fail to boot bolt,
        - **to recover** you will need to use broadband studio to flash a known good bolt (details further down this page).
3. Android device state, which is the android specific state of the device.
    - **locked** (production device) or **unlocked** (default).
    - device state is toggled through android fastboot command and the state is saved on the emmc|rpmb block.
        - thus if the rpmb block is unavailable for any reason (such as not yet programmed with a valid key), avb state will always be considered 'unlocked'.
        - the reason to choose 'unlocked' is such that the device can still be used reasonably well, however it would never be considered a '**green**' verified boot device, that is not production worthy.
    - "**locked**" device can only process and launch signed|validated bootloaders and boot.img and will fail to boot if any of those is invalid for any reason (as example not signed or not signed correctly for the part).
    - "**unlocked**" device can boot any custom (unsigned) image from boot.img onward, that's the 'development' mode essentially; however beware: you will still need to flash signed bootloader.img on those unlocked devices following secure otp enabling.

Relation to Android Verified Boot State:

1. avb **locked** device will advertise to the kernel that the android verified bootstate is **GREEN** (assuming successful boot chain validation).
2. avb **unlocked** device will advertise to the kernel that the android verified bootstate is **ORANGE**.
3. avb **un-capable** (i.e. not secure boot otp) device do not report any android verified bootstate, this ensures that the information is only populated by a signed bsu and therefore any non secure otp'ed device reporting such information can be considered bogus.

Determine the State of Your Device

It is important to understand the state of the device you are using before you start pushing new images to it. The following will help you make the correct decision to avoid getting in an non recoverable (or non easily recoverable) state.

Default

By default, or if you cannot determine the state of the device, you should always assume the most constricted environment applies: the device is secure boot otp'ed and avb locked.

Making this assumption is easy because flashing a signed bolt image on the device even if secure boot otp is not enabled has no consequences; the image is simply not validated on boot (but it still has the sufficient knowledge to boot at least up to the bolt command prompt from which you can recover bsu onward if need be).

However the opposite is not true: flashing a non signed bolt image on a secure otp device will result in the device being unable to boot up (and need to be revived through broadband studio download).

Check From Bolt

If your device is running some bolt version, the simplest way to check for the state is to look at the command prompt offering. Secure boot otp'ed bolt images and avb aware bootloader would offer a set of commands that only such packages contain; not finding those commands is a good indication that you are not running any secure otp nor avb aware system.

```
BOLT> help

verifyboot ........ Load, verify signature and on success launch
a
                    program from <device> using <loader>
```

If you see the "verifyboot" command exists under bolt, you are running a bolt aware avb image.

Otherwise you are not running any avb aware bootloader and you are also most likely not running a signed bolt to begin with.

Command Restrictions Under avb Locked State

When the device is in avb locked state, the following BOLT commands are being denied execution on the BOLT command line:

- batch
- boot
- erase
- flash
- go
- load
- rpmb android
- save

The following commands are denied execution on the fastboot host command

- fastboot flash [...]

The blacklisting is to enforce that only fastboot command can be used for flashing device from bootloader as well as to ensure that the only way to load and launch an image is through the explicit loading commands which include image signature validation.

Consequently, when in avb locked state, the only way you can flash new images is through android OTA [classic or A|B system] update, just like it would be the case for a production device. **There is no exception nor back-door loophole to this enforcement**.

Those blacklisted commands can be made to work again after the device is "avb unlocked". Follow instructions below to unlock a device.

Making the Device avb Aware

There are two actions to follow. It is strongly recommended to ensure the steps are followed in this order. Both actions require you are running a secure bootloader; although it does not necessarily require you are in secure mode (i.e. secure otp is not programmed).

First Action: Setup the rpmb key (steps [1.a] and [1.b])

Your device may actually already have a rpmb emmc setup done; this is possible because rpmb configuration and secure boot are orthogonal, However both are needed for the avb use case.

In any cases, trying to re-configure an already configured rpmb key is just a no-op (i.e. failure).

Step 1.a

```
BOLT> rpmb gen-key <device>
```

This action will generate and program the rpmb key if none was setup before. If one is setup, programming will be ignored and rpmb won't work properly for this device (unless the key was by chance programmed using the same secure bootloader method you are now running).

- **<device>** is the emmc's rpmb block, which typically is identified as "**flash3**" by bolt unless the bootloader has been customized and|or the emmc device is prefixed (may be the case if you have multiple flash devices installed on the system).
    - you can confirm this by issuing a "show devices" on the bolt prompt.

Example: this is how you can identify the rpmb device from a device list

```
BOLT> show devices
Device Name          Description
-------------------  ------------------------------------------------------
---
               uart0  16550 DUART at 0xf040a900 channel 0
                mem0  Memory
              flash0  EMMC flash Data : 0x000000000-0x370000000 (14080MB)
       flash0.macadr  EMMC flash Data : 0x000004400-0x000004600 (512B)
        flash0.nvram  EMMC flash Data : 0x000004600-0x000014600 (64KB)
          flash0.bsu  EMMC flash Data : 0x000014600-0x000054600 (256KB)
         flash0.misc  EMMC flash Data : 0x000100000-0x000200000 (1024KB)
        flash0.hwcfg  EMMC flash Data : 0x000200000-0x000300000 (1024KB)
         flash0.boot  EMMC flash Data : 0x000300000-0x002300000 (32MB)
     flash0.recovery  EMMC flash Data : 0x002300000-0x004300000 (32MB)
        flash0.cache  EMMC flash Data : 0x004300000-0x014300000 (256MB)
       flash0.splash  EMMC flash Data : 0x014300000-0x014F00000 (12MB)
     flash0.metadata  EMMC flash Data : 0x014F00000-0x016300000 (20MB)
       flash0.system  EMMC flash Data : 0x016300000-0x062B00000 (1224MB)
     flash0.userdata  EMMC flash Data : 0x062B00000-0x1D1FFBE00 (5877MB)
              flash1  EMMC flash Boot1: 0x000000000-0x000400000 (4MB)
              flash2  EMMC flash Boot2: 0x000000000-0x000400000 (4MB)
              flash3  EMMC flash RPMB : 0x000000000-0x000020000 (128KB)
                eth0  GENET Internal Ethernet at 0xf0b00000
               mdio0  GENET MDIO at 0xf0b00800
               sata0  SATA3 AHCI Device
            usbdev0  USB BDC at 0xf0472000
         tcpfastboot0  TCP Fastboot (port 1234)
          tcpconsole0  TCP Console (port 23)
*** command status = 0
```

Step 1.b

Note: if for some reason your device is already "secure opt'ed", the command being described in this step is not going to be available (blacklisted), instead you would need to seed the state using the "fastboot flashing unlock" command (described further in this document).

In step [1.a] above, we seeded the rpmb key. in this second step [1.b], we initialize the block content to be in "unlocked" state. The reason we initialize in unlocked state is to prevent bad user experience with bricked devices as soon as you enable the support. By being unlocked, we give a chance for the user to fastboot properly signed image and re-lock the device if wanted; that said, it is possible to initialize the state as being "locked".

```
BOLT> rpmb android <device> unlock
```

- **<device>** is the same rpmb device as in step 1.

By issuing the command to seed the rpmb content, you would implicitly verify whether the key was properly setup as well: if this step fails, it means the previous step must have failed as well, which is likely to be if the rpmb was previously seeded. **In that case, your device is not going to be able to provide avb support**.

Second Action: Enable the otp secure boot (step [2])

This section illustrates how to one time program your device for secure boot otp and avb support.

```
to read the current value of the secure boot otp.
BOLT> secboot

to enable secure boot otp.
BOLT> secboot -program
```

... done.

Conclusion

Once you have run through steps [1.a], [1.b] and [2]; it is recommended to mark the device with some sticker information to help identify the state it is in to make potential users aware of what to expect.

We recommend a simple marking with the following information:

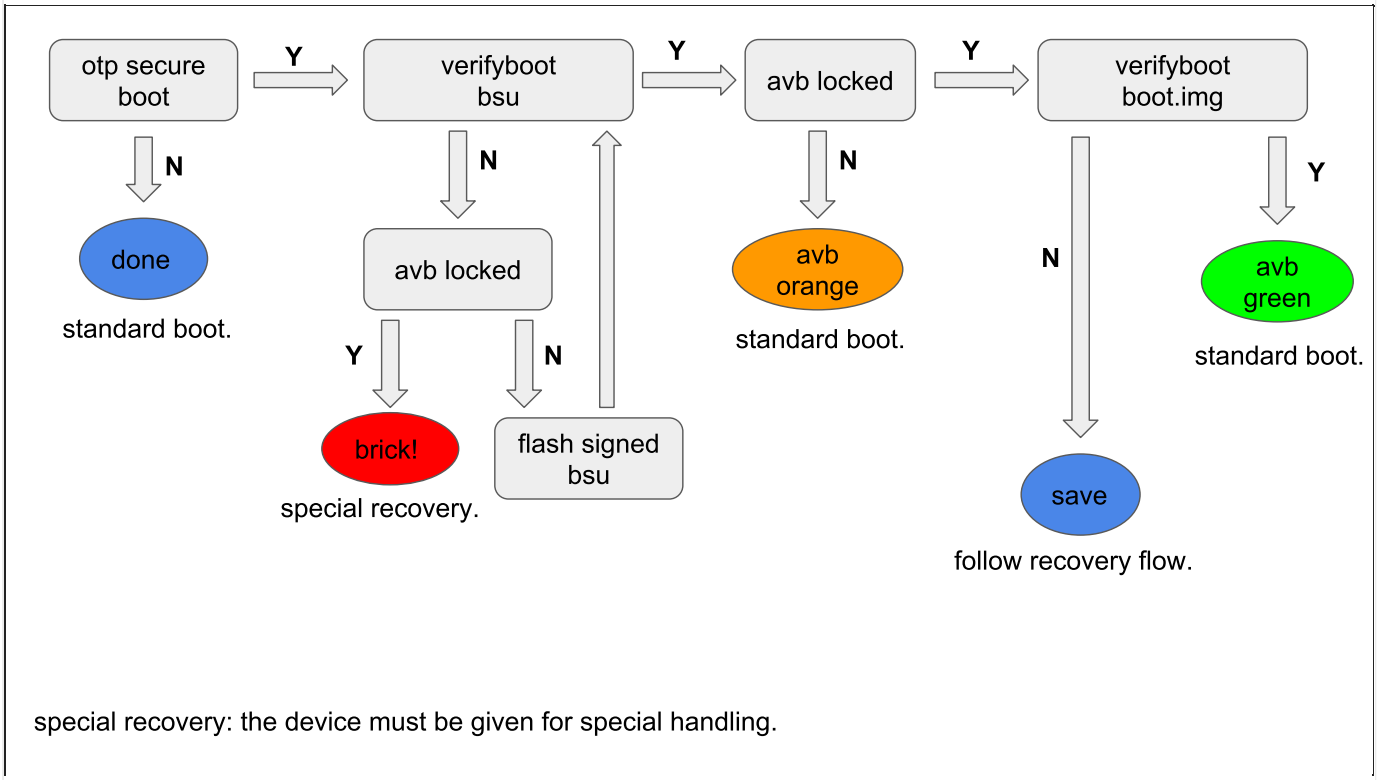
- "RPMB: GOOD" or "RPMB: BAD"; to signify the fact that rpmb block is functional and has been programmed according to the expectations of this avb integration.
- "SECURE BOOT: YES" or "SECURE BOOT: NO"; to signify the fact the OTP for secure boot has been programmed successfully or not.

avb Boot Flow and avb Unlocking

Boot Flow

Let's take a look at the typical expected boot flow for the avb aware (or not) device.

AVB BOOT FLOW



Open *avb boot flow*

Splash Feedback

When splash screen is enabled for the device, it is a good idea to push the preset "android-vboot-bolt-splash" splash content to the device "splash" partition. This will help provide a feedback to the user about the device state (the default bolt splash support is enhanced to enable those feedback for avb state), in particular:

- if the device is unlocked, a **ORANGE** splash will flash, failure to see it would mean the device is locked (thus cannot be updated)
    - this helps give a simple indication that the device can be flashed from bootloader for development purposes as example.
- if the device fails to verify boot, a **RED** splash will flash, indicating that the boot failed.
    - this helps dtermine that either the android bsu or the boot.img are invalid for the device (i.e. signed improperly, or not signed at all).

Unlocking avb State

Unlocking the avb state can be useful for several purposes:

1. use it for recovery of a device which has an incorrect boot.img (as example a non signed image was flashed on the device).
2. use it for jumping into the **ORANGE** verified boot state which allows you to load you own (unsigned) images freely, that is the most common development mode expected.

To unlock, you must put the device in fastboot mode and then issue the command

```
HOST> fastboot flashing unlock
```

```
[confirm by repeating the same command]
```

```
HOTST> fastboot flashing unlock
```

You will be asked to confirm the choice by issuing the same command again. **Unlocking the device causes the userdata (and cache if applicable) partition(s) to be wiped out** - be patient.

- **Important**: because we do not have ext4 support in the bootloader (due to gpl conflicts) you will need to flash a userdata.img following the unlocking stage in order for the device to properly recognize the data partition as ext4 format.
- The wiping of the userdata/cache is removing all content but not recreating any file system.

After a device has been unlocked, you can choose to lock the device again in order to run an avb flow (as example) - this transition from unlocked to locked does not trigger a wipe out of the userdata, But you may need to do so possibly to get into a proper boot up state depending on what was running on the device while unlocked, so we do recommend that you flash a clean userdata image as well following this step.

Because you are running already a secure otp'ed mode at that point and so a validated bolt+bsu images, there is no problem with locking/unlocking the device, You will always be able to recover from a bad boot.

avb Boot Commands

When running the device in "**avb locked state**", **there are only two commands which can be used to boot the device properly**; any other combination is bound to fail.

- This restriction is purposefully built into the bootloader blacklist of commands to ensure the user is aware of the state it is in, as well as to avoid support burden with users who try to be creative about booting the device.

When running the device in "**avb unlocked state**", the standard set of bolt and (android) bsu commands can be used without restriction - those are not repeated here.

1) Booting the bsu

```
BOLT> help verifyboot

  SUMMARY

     Load, verify signature and on success launch a program from <device>
using <loader>

  USAGE

     verifyboot <loader (-img|-bsu)> [-noclose] <device>


  OPTIONS

     -noclose ......... Don't close network connection
     -img ............. Android boot.img loader
     -bsu ............. Android secondary bootloader

*** command status = 0
```

issuing the command:

```
BOLT> verifyboot -bsu flash0.bsu
```

2) Booting the boot.img

```
BOLT> help android vboot

  SUMMARY

     Load an Android boot image into memory, verify its signature and boot
it if signed properly

  USAGE

     android vboot


*** command status = 0
```

issuing the command:

```
BOLT> android vboot
```

3) Practical Example

Assuming a otp'ed device in avb locked state, the following bolt command line should boot up android in the **GREEN** state upon successful validations of images in the boot chain

```
BOLT> verifyboot -bsu flash0.bsu && android vboot
```

The above command can, as example, be programmed into the STARTUP enevironment variable of bolt as a mean to automatically launch into this boot sequence from power up.

Integration with a|b Update

If you are running a device which enabled a|b update support instead of legacy boot (refer to this page for details), your boot command would look exactly the same as above,
The difference is you do not need to specify any of the bootloader variables (such as ANDROID_BOOT_IMG) nor pass in additional parameters (such as [-i=*]) since the boot selection is made from the commander block content.

avb Version and Signing

A couple notes on the signing procedure and version supported for avb.

- avb is supported starting from the URSR 16.4 onward.
- any prior versions of the android integration are not "avb" capable.
    - however, such versions can be "secure boot" aware, limited to the bolt image.

avb Version, Bolt Version and imagetool Version Correspondence

- avb version
    - the package version of the avb integration.
- ursr version
    - the version of the ursr release baseline.
- aosp version
    - the baseline version of AOSP this verified boot integration is compliant with.
- bolt version
    - the version of bolt necessary to support the integration; unless otherwise indicated, this is the last version and all prior versions would also be supported within the same scope.
- imgtool
    - the version of the signing tool required to properly sign the expected parts of the bootloader.img.
    - note that the bsu (and bl31 if applicable); as well as the android boot.img are all signed as a "kernel" blob from imagetool standpoint, there is no specific profile other than the kernel one to be used.
- reference
    - the reference platform validated against the avb delivery.

avb-1.x.y: android N

AVB	URSR	AOSP	BOLT	IMGTOOL	REFERENCE
0.9.0	16.4	N (7.0)	v1.23	1.2.2	avko

no longer supported.			BFW: 4.2.3 [1]		
1.0.0 **no longer supported.**	17.1 **(take 1)**	N (7.0|7.1)	v1.26 BFW: 4.2.3 [1]	1.2.4 [2]	avko
1.0.1 **no longer supported.**	17.1 **(final)**	N (7.0|7.1)	v1.27 BFW: 4.2.3 [1]	1.2.4 [2]	avko
1.0.2 **no longer supported.**	17.1 **(final)**	N (7.0|7.1)	v1.28 BFW: 4.2.3 [1]	1.2.4 [2]	avko cypress
1.0.3 **no longer supported.**	17.1 **(final)**	N (7.0|7.1)	v1.28 BFW: 4.2.3 [1]	1.2.4 [2]	avko cypress
1.0.4 **obsolete - use 1.0.4**	17.1 **(final)**	N (7.0|7.1)	v1.28 BFW: 4.2.3 [1]	1.2.4 [2]	avko cypress
1.0.5	17.1 **(final)**	N (7.0|7.1)	v1.28 BFW: 4.2.3 [1]	1.2.4 [2]	avko cypress

[1]. the 7439 (avko) BFW version is not the default one for this bolt revision.
[2]. an EK3 'random' 16-bytes blob is required for signing, content is irrelevant, it is simply to pass the signing tool checks.

avb-2.x.y: android O

note: avb-2.x.y is the package version provided by Broadcom; it has no relation to Google's Android Verified Boot version 2, which is currently not integrated with any reference platforms.

AVB	URSR	AOSP	BOLT	IMGTOOL	REFERENCE
2.0.0pre **obsolete - use 2.0.1**	17.3	O (8.x) **not compatible with prior AOSP (N and below)**	v1.32	1.2.6	avko cypress
2.0.1 **obsolete - use 2.0.2**	17.3	O (8.x)	v1.32	1.2.6	avko cypress
2.0.2 **obsolete - use 2.0.2.1**	17.3	O, O-MR1 (8.0, 8.1)	v1.32	1.2.6	avko cypress
2.0.2.1 **obsolete - use 2.0.2.2**	17.3	O (8.0)	v1.32	1.2.6	avko cypress
2.0.2.2	17.3	O (8.0)	v1.32	1.2.6	avko cypress
2.0.3 **obsolete - use 2.0.4**	17.4	O, O-MR1 (8.0, 8.1)	v1.34	1.2.6	avko cypress
2.0.4 **obsolete -**	17.4	O (8.0)	v1.34	1.2.6	avko cypress

use 2.0.5					
2.0.5	17.4	O (8.0)	v1.34	1.2.6	avko cypress

Example - generating a bootloader.img

For the sake of example, we assume that the build has produced a secure boot compliant bolt image and android bsu image; now we want to sign them and package them into a single bootloader.img using the standard image tool signing procedure.

This example illustrates the minimum configuration required.

Step 1 - sign bolt image

```
imagetool -L bolt -O bolt.cfg -P Generic
```

Step 2 - sign android bsu image

```
imagetool -L kernel -O bsu.cfg -P Generic
```

In this step, not the use the "kernel" signing profile; which tells imagetool to threat the bsu as a "blob".

The bsu.cfg content is as follows (note: this is an sample configuration, you may need to adjust some settings for you own configuration keys):

```
[global]

filetype=kernelformat2 # needed for imagetool 1.2.4 onward.
binaryfile=bsu.elf
signing=true
encrypting=false
keyfile=<keyfile.txt>
keysigfile=<keysig.bin>
epoch=0xff
epochmask=0xff
keyrights=0x09
marketid=0
marketidmask=0xffffffff

[params]

EK3file=<ramdom-16-bytes.bin> # needed for imagetool 1.2.4, remove if using imagetool >=
1.2.6.

[output]

outfile=bsu.signed.elf
```

Step 3 - package signed images

```
bootloaderimg.py bolt.signed.bin bsu.signed.elf
bootloader.signed.img
```

This final step is simply using the bootloaderimg packaging script delivered with the android bsu to put together the two signed images into a single container. There is no signing applied there as mentioned before.

Signing the boot.img

The boot.img is considered a "blob" from the point of view of the secure boot and it is to be signed with the same secure key as the bootloaders. For those reasons, the boot.img is expected to be signed in a similar manner as the bsu image signing, which is documented in the above section example.

Deltas from avb Version to the Next

This section provides information regarding the changes from avb version to the next, The main purpose is not to list all changes, rather to describe what has changed that one needs to be aware of, it will also list the main bugs that have been addressed when such are relevant.

avb for Android N

avb-1.0.2 to avb-1.0.3 [delivered]

- package sync'ed up to the latest bolt-v.1.28+ from android n-mr1-tv-dev.
- addressed improper calculation of lpddr4 memory configuration when running secure bootloader.

avb-1.0.3 to avb-1.0.4 [delivered]

- package sync'ed up to the latest bolt-v.1.28+ from android n-mr1-tv-dev.
- increased staging buffer for image validation from 24MB to 32MB.
- do not reboot system on key|image validation failure
    - only applies to android-bsu and boot.img signing.
- for a|b systems: add recovery attempt (retries and automatic slot switch) on various boot.img failure:
    - failure to validate signature, or failure to authenticate the image.
    - once the boot.img is found proper and loaded, the standard mechanism applies for retry and recovery.

- for a|b systems: add dm-verity failure feedback capture and proper settings.
    - this requires a kernel update and a matching boot_control hal update.

avb-1.0.4 to avb-1.0.5 [delivered]

- added ability to force a recovery boot using the [-recovery] command switch for boot|vboot commands.
- clean out region validation failure to allow re-use of the region for next attempt within the same boot scope.
- add bsp settling delay after key loading to fix lockup when building ssbl without console logging support.
- do not fail rpmb bound read|write if the key was never programmed, instead assume device is equivalent to an 'avb unlocked' setup.

avb for Android O

avb-2.0.0pre [delivered]

- first avb port to O bootloader.
- all problems identified and solved in avb-1.0.x are also addressed here.
- package sync'ed up to the latest bolt-v.1.32+ from android o-tv-dev.
- integrated automatic signing and generation of the bootloader.img (signed) when signing tool and signing package are available to the environment.
- **pre-release, do not use for O production.**

avb-2.0.0 to avb-2.0.1 [delivered]

- moved to 'release' state.
- package sync'ed up to the latest bolt-v.1.32+ from android o-tv-dev.

avb-2.0.1 to avb-2.0.2 [delivered]

- package sync'ed up to the latest bolt-v.1.32+ from android o-tv-dev.
- added explicit wait following region validation termination to prevent deadlock of secure processor.

avb-2.0.2 to avb-2.0.2.1 [delivered]

- package sync'ed up to the latest bolt-v.1.32+ from android o-tv-dev.
- added option to pass in encrypted images for bsu and boot.img to the vboot|verifyboot commands, allowing to include encryption as part of the secure boot flow for android boot.

avb-2.0.2.1 to avb-2.0.2.2 [delivered]

- fix proc-ins kernel decryption byte swapping error (required for decryption of bsu|boot.img withouth needing to swap bytes during offline signing process).

avb-2.0.2 to avb-2.0.3 [delivered]

- package sync'ed up to the latest bolt-v.1.34+ from android o-tv-dev.
- added saving android os and patch level version to rpmb for security processor use in secure services (keymaster, gatekeeper, etc...)

avb-2.0.3 to avb-2.0.4 [delivered]

- package sync'ed up to the latest bolt-v.1.34+ from android o-tv-dev.
- added option to pass in encrypted images for bsu and boot.img to the vboot|verifyboot commands, allowing to include encryption as part of the secure boot flow for android boot.

avb-2.0.4 to avb-2.0.5 [delivered]

- fix proc-ins kernel decryption byte swapping error (required for decryption of bsu|boot.img withouth needing to swap bytes during offline signing process).

Integrating avb Code Package

This section describes how to integrate the avb aware code into a standard android delivery. This assumes:

- an android release is available and contains the bootloader E1 (non secure) codebase.
- a corresponding bootloader E2 (secure) package has been obtained.
    - for a given android release, there is a specific E2 package that can be requested and obtained.
    - the android E2 package is a set of git patches to be applied on top of the E1 release.

In order to simplify the E1 to E2 integration, the android E2 package contains everything required in order to support the avb integration. The only parts not covered by it is the signing of the final images. Information on signing is found in the above section of this document.

Step-by-step Integration

The following is a step-by-step integration methodology which can be followed but is not necessarily mandated. However whatever you do, it is a good idea to refer to it.

- create a "**bolt-vb**" repository which is the exact clone of the "**bolt**" one from the android release and which is located at the same root.

```
$ ls -l vendor/broadcom

drwxr-xr-x 11 pierre clearusers 4096 Feb 22 10:44 bcm_platform
drwxr-xr-x 20 pierre clearusers 4096 Feb 22 10:44 bolt
```

```
drwxr-xr-x 20 pierre clearusers 4096 Feb 23 07:03 bolt-vb
drwxr-xr-x 10 pierre clearusers 4096 Nov  9 20:30 drivers
drwxr-xr-x  9 pierre clearusers 4096 Nov  9 20:31 refsw
```

    - we recommend that the bolt-vb clone contains the same .git root at the bolt, This allows simple patching of the E2 package as a git release, it also allows to easily keep track of changes.
- **Important**: the build steps for the avb aware bootloaders is integrated already in the device build.mk, the build assumes that the E2 package is going to be found under the "bolt-vb" root. If you choose to change that root, you need to modify the build script as well to instead point to your local tree.
- apply the content of the android E2 package on top of the "**bolt-vb**" tree.
    - the android E2 package comes as a set of git patches; you can either apply them as so (git am) or as a standard patch (patch -p1).
    - **either ways, it is recommended to keep the same number of patches as delivered, the patches content are already logically separated, each bringing a particular functionality required, or fixing a problem.**
- **create the target for your specific bootloader configuration**.
    - the rule of thumb is that you want to trim *all* targets from the configuration family you intend to build, except the one you care about.
    - the android E2 package shows a couple examples of this:
        - in the "**config/family-7439b0.cfg**" setup where all targets are removed except for the "**BCM97252SSFFG_NOAVS**" (which happens to be the target for the "**avko**" reference design).
        - in the "**config/family-7271b0.cfg**" setup where all targets are removed except for the "**BCM97271T**" (which happens to be the target for the "**cypress**" reference design).
    - once you have removed all but one target, the only other configuration change to make is to add a "**-fixed -**" identifier as last element of the "**ddr**" configuration rule for the device. see example:

```
# 2GB from two 4Gx16 per MEMC
ddr -n 0 -size_mb 1024 -base_mb    0 -clk 1067MHz -size_bits 4G -width 16 -phy 32
-fixed -
ddr -n 1 -size_mb 1024 -base_mb 2048 -clk 1067MHz -size_bits 4G -width 16 -phy 32
-fixed -
```

- now you can build your avb aware bootloaders.
    - the build target are "build_bolt_vb" and "**build_android_bsu_vb**"; the latter will also build the former.
        - starting in avb 2.x, we also add the "**build_bootloaderimg_vb**" rule which will not only build all the packages, but also sign them assuming you provided the signing information.
    - the result of the build will be found in the out directory as for a E1 package build.
    - note that the output image name is appended with a "**-vb**" to differentiate it from the standard (E1 package) images.

```
ls -l out/target/product/avko

-rwxr-xr-x 1 pierre clearusers  191746 Feb 28 07:48 android_bsu.elf           #
E1 android BSU
```

```
-rwxr-xr-x 1 pierre clearusers  200441 Feb 23 07:09 android_bsu-vb.elf         #
E2 android BSU
-rw-r--r-- 1 pierre clearusers 1032704 Feb 28 07:48 bolt-bb.bin                #
E1 android BOLT
-rw-r--r-- 1 pierre clearusers 1032704 Feb 23 07:09 bolt-bb-vb.bin             #
E2 android BOLT
```

- the last step will be to sign and package the bootloader images into a single container, those actions are described in the section prior.

Help! Recovery

Broadband Studio Download

In the case where the device isn't booting bolt for any reason, the only recovery path available is to use broadband studio to flash the bolt on the emmc device.

Note: broadband studio would only flash image to the "boot1" partition of the emmc flash; therefore you can 'safely' download an image to it that will replace the BOLT binary existing by flashing the downloaded image to offset 0.

The "flash explorer" functionality of the broadband studio is used for downloading the new image. the below picture illustrates the flashing step.

Special Recovery

In the case where you have a bad bsu (as example) yet are otp'ed secure boot in avb locked state; due to the command set restrictions enforced, you will have no simple way to flash a good bsu to get out of that problem when using the standard android bolt.

In such case, you need to do the following:

- get a non android signed bolt image (i.e. a reference bolt signed for the platform) and flash it using broadband studio (as mentioned above).
- once booted, from bolt command line, flash a proper bsu (signed) and proper android bolt (signed) individually (i.e. you cannot use the fastboot flash bootloader command in this case, so go back to old'skool flashing of individual partitions using the bolt native flash command).
- reboot.
- now you should be back online with a properly signed bootloader.img equivalent allowing you to proceed to next steps (avb locking or unlocking, etc...).

One Time Bolt Image

Allowing to run a one time signed only for the device android bolt image with special credentials to allow unlocking a avb locked state for a bricked device is under consideration as a recovery mechanism.