Device-side Software Update Strategies for Automotive Grade Linux

Konsulko Group, sponsored by ATS Advanced Telematic Systems GmbH

Overview

This whitepaper explores the area of software update strategies for devices running Automotive Grade Linux. The starting point is understanding several key use cases for updating software in an AGL[1] system. Several open source device-side software update mechanisms are compared with a focus on their ability to meet the stated use cases. Finally, recommendations are made for an approach that can be implemented for inclusion in AGL.

Use Cases and Requirements

The topic of software update on any computing device is very broad and can only be examined properly by narrowing the scope of the system, operating conditions, and the policies established for executing a software update. The following sections describe the use cases considered when evaluating the various mechanisms that could be employed in an AGL software update strategy.

System Description

AGL systems include both IVI[2] and ADAS[3] devices that run Linux on at least one processor in the system. The complete IVI or ADAS system is not necessarily limited to a single processing node, but is normally part of a larger network that includes a number of ECU[4] nodes throughout the automobile. Due to this distributed architecture it is often necessary to update software on

[1] Automotive Grade Linux (https://www.automotivelinux.org/agl-specification)
[2] In-Vehicle Infotainment
[3] Advanced Driver Assistance System
[4] Electronic Control Unit

the ECUs as well as the node running AGL. The device running AGL is generally the node containing networking or other external communication capabilities to handle input of a software update image. As such, it will normally be the focal point for software update in the entire automobile providing update services for itself and multiple ECUs. For the AGL node, an update mechanism must support update of all software loadable bootloaders, the kernel and supporting configuration data, and all Linux filesystems as the system design may or may not require updating any and all of these software artifacts. Finally, AGL's build system is based on OpenEmbedded so any technology must be able to integrate into a well-behaved OpenEmbedded build environment.

Operating Conditions

An automobile product has several operating conditions that are both unique and also share similarities with other devices. First, and foremost, the automobile is a mobile device, which requires the ability to be updated at any location with minimal impact to the owner. The automobile industry has historically relied on costly on-site visits to service facilities to update software, however, with the increasing complexity and amount of software in an automobile this approach does not scale. For this reason, support of OTA[5] software updates via wireless networking has become a fundamental use case.

The automobile industry is subject to numerous safety regulations. As such, automobile software updates must also be conducted in a manner in which safety is not compromised. Failure is common and expected during software updates with causes that include image corruption during installation, corrupted image/filesystem on storage medium, failure to receive update image, and power failure during update. It is critical that a software update strategy support deployment of updates such that if failure occurs, a previous version of the software can be deployed such that the automobile is functional and safe to operate until a new update can be deployed. In order to minimize the possibility of failure, updates must be deployed in an atomic manner, guaranteeing integrity of the software update once it is deployed on the device. In addition, the cost nature of OTA delivery methods such as 4G networks requires QoS to be implemented to control the costs of deploying large software updates to a fleet of vehicle. A software update may need to be delivered at specific time of day and at a specific bandwidth rate corresponding to the region the automobile is sold or located within. Finally, the update mechanism must be able to verify that the software has not been tampered with before installation and must cooperate with a system-level chain of trust during the boot process that verifies images starting a power-on through to application lifecycle. This is commonly handled by executing in a trusted environment that leverages hardware features such as a Trusted Platform Module (TPM) and Trusted Execution Environment (TEE). Popular TEEs include ARM TrustZone and Intel TXT.

[5] Over The Air

Policies

A software update mechanism needs the ability to support a flexible set of OEM software update policies for the automobile. Each OEM has a different approach to the rate of updates, quantity, and size of updates. An OEM with a higher rate of updates will require an update mechanism that's optimized not only for the baseline requirements but also for speed. Some OEMs may provide value-add features that are unlocked by a recurring fee. These value-add features will require the ability to enable or disable a specific feature based on a fee. Regulations requiring a recall may require the OEM to rollback an update due to a recall notice. Finally, the policy of update deployment timing will be defined by the OEM to control operator experience (e.g. only at "key-off" and with operator acknowledgement) as well as meet any safety requirements.

Summary

The following requirements are derived from the use cases and are listed in priority order:

1. Atomic software release update
2. On failure, deploy previous working bootloader, kernel and configuration, and filesystems on AGL device
3. Update of bootloader, kernel and configuration data, and filesystems on AGL device
4. Support for OpenEmbedded-based builds
5. Support for updating both the AGL device and any ECU devices
6. Flexible delivery of software image(s) with QoS[6] controls and supporting arbitrary interfaces (WiFi, 4G, USB, etc.)
7. Support for signing of images and verification of images on installation
8. Support trusted boot and execution of software update in a trusted application environment leveraging the platform's hardware TPM and/or TEE features.
9. Enable/disable a specific feature and apply/rollback system updates incrementally rather than through a complete OS update that replaces the filesystem.

Open Source Software Update Tools

Historically, software update on embedded Linux systems has been implemented in a "one-off" manner by companies shipping products based on Linux. Typically, the details of how an update is performed has been very specific to the system and hardware design as well as the amount of storage space available to implement a software update mechanism. Over time, these ad hoc update methods have matured to where requirements have begun to show some commonality allowing the problem of software update to be solved in a general manner. The following projects provide implementations of common requirements for embedded Linux software update.

[6] Quality of Service

SWUpdate

SWUpdate[7] is an extensible software update framework that supports atomic update of arbitrary software images in a Linux system. SWUpdate defines a standard compound image format based on cpio which contains a header, software description in XML, and any number of software update sub-images. It allows for extensible image parsers to support new software image types as well as extensible handlers to support new protocols or installers for specific storage peripherals and layouts. The XML-based software description provides a manifest of all the software sub-images contained in the compound image. It defines attributes corresponding to the version of a software release and the name and type of each sub-image contained within that software release. For example, a software release for a typical Linux system may have sub-images for the zImage, dtb, root filesystem, and apps filesystem. The compound image and each sub-image can be hashed with SHA256 and signed with an AES256 key for use in verification before installation.

SWUpdate can operate in either a single copy or dual copy update scheme. Single copy updates require updates to be managed by the bootloader booting into a kernel and initrd that then runs the SWUpdate tool to install a new set of software images. This approach conserves space but does not allow for fallback to a known good software image on failure. Rather, it requires reinstall of a previously failed image or download of a new software after reboot from a failed install. This approach is improved upon by using a typical dual copy approach which involves keeping a known good recovery copy of the software in a second partition. SWUpdate supports a dual copy software update strategy by use of Software Collections. A Software Collection is simply two named alternate copies of software images with attributes defined to describe what the target installation location is for each copy. This allows SWUpdate to be called to deploy to either storage partition by specifying which Software Collection to deploy to the storage device.

There's no explicit support for integrating SWUpdate into a TEE but there's no reason it could not be ported to such an environment. SWUpdate provides a reference mongoose webserver for use as a target-based software update user interface. However, it also supports an API for arbitrary software to communicate with it in order to install images and receive status on installation status. It is possible to integrate the RVI SOTA client with SWUpdate using this API to perform the installation. For compound image generation, SWUpdate provides a meta-swupdate[8] layer which supports generation of signed images in OpenEmbedded. The project has a small community around it and is also included in the buildroot[9] project.

[7] https://sbabic.github.io/swupdate/swupdate.html
[8] https://github.com/sbabic/meta-swupdate/
[9] https://buildroot.org/

Mender

Mender[10] is a dual copy oriented framework that supports the end-to-end management of root filesystem updates and rollback. It supports a OTA delivery server and provides a target side software update tool to manage deployment of updates. Mender specifically supports only U-Boot as the bootloader and implements support for rollback to the second copy of an OS using U-Boot's *bootlimit* feature. It does not support update of the bootloader itself nor does it support incremental updates.

Mender is written in Go and provides a *meta-mender* layer for OpenEmbedded to support building embedded systems including Mender support. There's no explicit support for integrating Mender into a TEE but there's no reason it could not be ported to such an environment.

Mender does not appear to have any community adoption as evidenced by list posts and commits only originating from Mender Software itself.

Resin

Resin[11] is a container-based frame for delivering rolling updates to embedded Linux systems. It is a client/server system where the server has support for building packages and containers with the package content. In addition, the server has a docker registry for the containerized applications. On the target, a supervisor application runs in a container providing monitoring of the device as well as handling rollout of new docker containers with applications. This can all be managed from the cloud infrastructure maintained by resin.io.

Resin has no provision for managing updates to the core operating system binaries. It is completely focused on an immutable and non-updated base OS and fluidly updated applications within containers. These containers are stateless except that a */data* directory is provided that will persist per device between updates. Currently, support for binary deltas of Docker container updates is a feature that is in beta. Resin makes use of TLS for the client/server connection using RSA for key exchange and AES256 for data encryption. There's no explicit support for integrating Resin OS into a TEE but there's no reason it could not be ported to such an environment. A *meta-resin* layer for OpenEmbedded is provided to support building the base Resin OS for deployment on target devices.

Resin.io started as a commercial service only and then moved to open source their framework in late 2015. As Resin OS has support for basic I/O on several popular community boards (Raspberry Pi, Edison, BeagleBone) it seems to have gained some community support as evidenced by commits from those outside of Resin itself.

[10] https://mender.io/
[11] https://resin.io/

swupd

swupd[12] is a revisioned software update mechanism which was designed to meet the requirements of the ClearLinux[13] distribution project from Intel. It is intended for use in Linux systems that update software in small increments at a rapid pace. Swupd does not use packages as a unit of update like traditional Linux distributions. Instead, swupd defines bundles which are a composed of a set of specific package versions. A bundle is the smallest installable component in an swupd system. These bundles are combined into a OS release that carries a single version number. By default, swupd demands that the Linux distribution be designed to operated as a stateless Linux OS. That is, an unpopulated /etc/ directory results in a bootable system that is factory reset.

Swupd provides client and server side tools to manage updates. The swupd-client support adding and removing specific bundles or updating to a new OS release atomically. In addition, the client supports checking the update server for a new OS release. There's no explicit support for integrating swupd-client into a TEE but there's no reason it could not be ported to such an environment. The swupd-server provides tools to create bundles, create OS releases, and host them for consumption by the client. The meta-swupd[14] layer integrates the swupd-server tools for generation of bundles and OS releases for OpenEmbedded as well as the swupd-client tools to support software updates on the target filesystem.

An OS release is defined by a "Manifest of manifests" which lists each bundle that's part of a release. Each bundle has a manifest which lists each change to a file, directory, symlink, etc. Swupd makes use of bsdiff (binary diff utility) to minimize the size of an OS update. Each manifest and change is identified using HMAC-256 to support verification of the OS release content. When an update is requested, the client downloads all manifests, then uses those to determine the packs of updates to download, and then applies each change in sequence. Post update scripts can be triggered using the systemd update-triggers.target.

Swupd was created for the ClearLinux distribution and also adopted by the Ostro OS[15] distribution, both of which are Intel project. The project mailing lists are dominated by Intel employees and it appears there's no adoption of swupd outside of these Intel projects.

OSTree

OSTree is similar in scope to swupd, providing support for atomic upgrade of Linux filesystems. A server composes content to be used on a client system where OSTree manages deployment of the content. By default, it defines two persistent directories across updates, */etc* and */var*,

[12] https://github.com/clearlinux/swupd-client
[13] https://clearlinux.org/
[14] http://git.yoctoproject.org/cgit/cgit.cgi/meta-swupd/
[15] https://ostroproject.org/

rather than operating as a stateless system like many systems. OSTree is best described as a configuration management tool for filesystem binaries. It allows for multiple releases to be stored as deltas of files, generated using bsdiff on the server or build system. OSTree internally follows the design of git using tree objects to track the history of filesystems including both content and metadata. It uses SHA-256 to hash changes in the tracked filesystems and GPG[16] is used to sign and verify commits/releases.

An administrative tool is provided on top of this version control system to management deployment of filesystems. *ostree admin upgrade* will update a deployment tree to the latest release on the update server and prepare the system to boot that release on the next boot. Likewise, *ostree admin deploy* and *ostree admin undeploy* can deploy or rollback a specific commit or version for the next boot. OSTree can also manage bootloader configuration files that conform to the boot loader specification[17]. The Gnome Continuous project maintains a base Gnome OS that conforms to the distribution needs (/usr Merge[18]) and patches to support OSTree compliant builds for OpenEmbedded have been previously submitted[19].There's no explicit support for integrating the *ostree admin* tool into a TEE but there's no reason it could not be ported to such an environment.

OSTree has been adopted by Gnome Continuous[20], rpm-ostree/Project Atomic[21], and flatpak[22]. The ostree-list is not very high volume but shows a number of different email domains indicating use by a number of companies and individuals including those from the Gnome Project and Red Hat.

Other Technologies

There are a number of other technologies that are used in ad hoc software update mechanisms. Many make use of a dual copy scheme as described in the SWUpdate section. However, this is not always sufficient to manage incremental updates. This results in sometimes layering technologies such as overlayfs[23] on top of a dual copy scheme in order to support maintenance updates without requiring a complete filesystem update.

Recommendations

Before making the final recommendations, we compare and contrast the pros and cons of each of the update technologies.

---

[16] https://www.gnupg.org/
[17] https://www.freedesktop.org/wiki/Specifications/BootLoaderSpec/
[18] https://www.freedesktop.org/wiki/Software/systemd/TheCaseForTheUsrMerge/
[19] http://lists.openembedded.org/pipermail/openembedded-core/2013-August/083595.html
[20] https://wiki.gnome.org/Projects/GnomeContinuous
[21] http://www.projectatomic.io/
[22] https://github.com/flatpak/flatpak
[23] https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt

Both OSTree and swupd provide a robust way to manage Linux filesystem updates when the system is working properly. Both assume that the underlying filesystem is not corrupt and that there is no data loss. Because of this, neither satisfies all possible failure cases on its own. OSTree is more conservative that swupd in that it requires a reboot on any update when deploying its newly configured filesystem. Swupd does not seem to have wide community adoption (used only in two Intel projects) and so the recommendation is to focus on OSTree as a component of a software update solution for systems requiring fine-grained and frequent updates as well as rollback of updates. Both Mender and Resin OS seem to be primarily driven by their commercial efforts and not widely adopted as update mechanisms by other projects. Mender is focused on a dual-copy only strategy and Resin OS is fundamentally based on a base OS that requires containers for all applications.

As mentioned, OSTree is not sufficient to meet all failure scenarios on its own. In order to handle the case of a corrupted filesystem or updating boot loaders that do not reside on a Linux filesystem it's necessary to couple it with another system. Certainly if everything goes correctly, a pure OSTree system could update bootloaders on its own with appropriate tools in the filesystem. However, nothing OSTree offers handles the case of its own underlying filesystem become corrupt and the need to fall back to the last known working system. This is where a dual copy approach becomes very important in a reliable system. SWUpdate provides a reasonable implementation of a dual copy approach and has a number of people starting to make use of it as a simple software update framework. It needs to be extended to support additional software fetching APIs but the maintainer seems open to contributions.

It is our recommendation that the reference AGL software update strategy make use of SWUpdate in a dual copy configuration and integrate OSTree support. This allows recovery from a corrupt partition for the exception case, but also optimizes the common case where small, incremental updates can be quickly applied or rolled back as needed to me OEM policy. As a part of this effort, it will be necessary to also provide a reference port of the SWUpdate and OSTree administration tools to a TEE to demonstrate the ability to execute the update process in a trusted environment. With the update process executing in the TEE, the cryptographic verification performed by the SWUpdate and OSTree update mechanisms is now a trusted action. The software running in the TEE is only one component of the overall attack surface of the entire software update process. It is also critical that the client/server SOTA network protocol and key handling processes follow best security practices or else the actual content being delivered could be compromised. It is recommended to support OP-TEE using an ARM QEMU target for the proof-of-concept implementation such that anybody can download and test the solution running on QEMU. This comprehensive approach is designed to satisfy all the requirements set forth in this paper.