Inter-domain communication

Contents

This documents a suggested design for an inter-domain communication system, which exports services between different domains. Some domains can be trusted such as the automotive domain. Some domains are untrusted such as the consumer-electronics domain. Those domains can execute on a variety of possible configurations.

The major considerations with an inter-domain communication system are:

- Security. The purpose of having separate domains is for security, so that untrusted code (application bundles) can be run in one domain while minimizing the attack surface of the safety-critical systems which drive the car.
- Flexibility for different hardware configurations. The domains may be running in one of many configurations: virtualised under a hypervisor; on separate CPUs on the same board; on separate boards connected by a private in-vehicle network; as separate boards connected to a larger in-vehicle network with unrelated peers on it; in separate containers.
- Flexibility for services exposed. The services exposed by the automotive domain are dependent on the vendor which implemented the automotive domain. The consumer-electronics domain depends on third-parties. Their update and enhancement cycle and security rules may differ.
- Asynchronism and race conditions. This is a distributed system, and hence is subject to all of the challenges[1] typical of distributed systems.

Terminology and concepts

Automotive domain

The *automotive domain* (AD) is a security domain which runs automotive processes, with direct access to hardware such as audio output or the in-vehicle bus (for example, a CAN bus or similar).

In some literature this domain is known as the 'blue world'. This document will consistently use the term *automotive domain* or *AD*.

Consumer-electronics domain

The *consumer-electronics domain* (CE domain; CE) is a security domain which runs the user's infotainment processes, including downloaded applications and processing of untrusted content such as downloaded media. Apertis is one implementation of the CE domain.

In some literature this domain is known as the 'red world', 'infotainment domain' or 'IVI domain'. This document will consistently use the term *consumer-electronics domain* or *CE domain* or *CE*.

[1] https://www.cl.cam.ac.uk/teaching/1516/ConcDisSys/materials.html

Connectivity domain

In some setups the *AD* and *CE* are not directly exposed to external networks and hardware. In those cases a *connectivity domain* hosts agents which can directly access the Internet or plug-and-play hardware devices such as USB keys, SD cards or Bluetooth devices and provide their services to applications running in the more isolated domains. This domain can be referred to as *CD*.

Trusted path

A trusted path[2] is an end-to-end communications channel from the user to a specific software component, which the user can be confident has integrity, and is addressing the component they expect. This encompasses technical security measures, plus unforgeable UI indications of the trusted path.

An example of a trusted path is the old Windows login screen, which required the user to press Ctrl+Alt+Delete to open the login dialogue. If a malicious application was impersonating the login dialogue, pressing Ctrl+Alt+Delete would open the task manager instead of the login dialogue, exposing the subversion.

In the context of Apertis, an example situation calling for a trusted path is when the user needs to interact with a UI provided by the AD. They must be sure that this UI is not being forged by a malicious application running in the CE.

Control stream

A *control stream* is a network connection which transmits low bandwidth, latency insensitive messages which typically contain metadata about data being transferred in a data stream. In networking, it is sometimes known as the *control plane*.

A control stream for one protocol may be treated as a data stream if it is being carried by a higher layer (or wrapper) protocol, as the control data in the stream is meaningless to the higher layer protocol.

If a designer is concerned about whether a particular stream's performance requirements make it suitable for running as a control stream, it almost certainly is not a control stream, and should be treated as a data stream. A new control protocol should be built to carry more limited metadata about it.

A control stream can operate without a data stream (for example, if there is no performance-sensitive data to transmit).

Data stream

A *data stream* is a network connection which transmits the data referred to by a control stream. This data may be high bandwidth or latency sensitive, or it

[2] https://en.wikipedia.org/wiki/Trusted_path

may be neither. In networking, it is sometimes known as the *data plane*.

A data stream cannot operate without an associated control stream (which carries its metadata).

Traffic control

Traffic control (or bandwidth management[3]) is the term for a variety of techniques for measuring and controlling the connections on a network link, to try and meet the quality of service requirements for each connection, in terms of bandwidth and latency.

Use cases

A variety of use cases which must be satisfied by an inter-domain communication system are given below. Particularly important discussion points are highlighted at the bottom of each use case.

All of these use cases are relevant to an inter-domain communication system, but some of them (for example, Video or audio decoder bugs) may equally well be solved by other components in the system.

Standalone setup

An app-centric consumer electronics domain (CE) is running in a virtual machine on a developer's laptop, and they are using it to develop an application for Apertis. There is no automotive domain (AD) for this CE to run against, but it must provide all the same services via its SDK APIs as the CE running in a vehicle which has an Apertis device. The CE must run without an accompanying AD in this configuration.

Basic virtualised setup

An embedded automotive domain (AD) and an app-centric consumer electronics domain (CE) are running as separate virtualised operating systems under a hypervisor, in order to save costs on the bill of materials by only having one board and CPU. The AD has access to the underlying physical hardware; the CE does not. The two domains have a high bandwidth connection to each other (for example, Ethernet, USB, PCI Express or virtio). The two domains need to communicate so that the CE can access the hardware controlled by the AD.

Linux container setup

Containers are based on Linux kernel containment features, including, but not limited to, Linux kernel namespaces, control groups, chroots (pivot_root), capabilities.

[3] https://en.wikipedia.org/wiki/Bandwidth_management

Both AD and CE are dedicated Linux containers on a host directly running on the hardware or in a virtual machine. AD is allowed to access safety-sensitive devices. CE is not allowed any access to safety-sensitive devices but may be able to access external devices like smartphones over Bluetooth, USB mass storage or security keys.

Communication is based on the Unix Domain Sockets (UDS) mechanism provided by the Linux kernel.

This setup can be used both for production setups on hardware board and on a developer's system for Apertis application development. It can be possible to provide a fake AD container for emulation and testing purposes.

Isolation between containers is unavoidably limited when compared to the isolation between virtual machines, just like separate boards provide more isolation than VMs. This is due to the fact that a single kernel is shared by all containers. However in this document we assume processes are not able to escape from the isolated environment or get access to resources on the host system or other containers for which they haven't been explicitly granted access.

Multiple CE domains are allowed with the above setup. In this setup, a Connectivity Domain can also coexist with AD and CE. It is responsible for any interaction with external networks and provides isolation in the case a network stack is compromised when that stack is not implemented in the shared kernel.

Separate CPUs setup

The AD is running on one CPU, and the CE is running on another CPU on the same board. The two CPUs have separate memory hierarchies. They maybe using separate architectures or endianness. The AD has access to all of the underlying physical hardware; the CE only has access to a limited number of devices, such as its own memory and some kind of high bandwidth connection to the AD (for example, Ethernet, USB, or PCI Express). The two domains need to communicate so that the CE can access the hardware controlled by the AD.

Separate boards setup

The AD is running on one mainboard, and the CE is running on another mainboard, which is physically separate from the first. They may be using separate architectures or endianness. The two boards are connected by some kind of vehicle network (for example, Ethernet; but other technologies could be used). There are no other devices on this network. The vehicle owner (and any other attacker) might have physical access to this network. The AD has access to various devices which are connected to its board and not to the CE's board. The two domains need to communicate so that the CE can access the hardware controlled by the AD.

Separate boards setup with other devices

The AD is running on one mainboard, and the CE is running on another mainboard, which is physically separate from the first. They may be using separate architectures or endianness. The two boards are connected by some kind of vehicle network (for example, Ethernet; but other technologies could be used). There are many other devices on this network, which are addressable but whose traffic is irrelevant to the CE–AD connection (for example, a telematics modem, or a high-end amplifier). The vehicle owner (and any other attacker) might have physical access to this network. The AD has access to various devices which are connected to its board and not to the CE's board. The two domains need to communicate so that the CE can access the hardware controlled by the AD.

(Note: This is a much lower priority than other setups, but should still be considered as part of the overall design, even if the code for it will be implemented as a later phase.)

Multiple CE domains setup

The AD is running on one mainboard. Multiple CE domains are running, each on a separate mainboard, each physically separate from each other and from the AD. The boards are connected by some kind of vehicle network (for example, Ethernet; but other technologies could be used). There are many other devices on this network, which are addressable but whose traffic is irrelevant to the CE–AD connections (for example, a telematics modem, or a high-end amplifier). The vehicle owner (and any other attacker) might have physical access to this network. The AD has access to various devices which are connected to its board and not to the CEs' boards. Each CE domain needs to communicate with the AD so that it can access the hardware controlled by the AD.

(Note: This is a much lower priority than other setups, but should still be considered as part of the overall design, even if the code for it will be implemented as a later phase.)

Touchscreen events

The touchscreen hardware is controlled by the AD, but content from the CE is displayed on it. In order to interact with this, touch events which are relevant to content from the CE must be forwarded from the AD to the CE. Users expect a minimal latency for touch screen event handling. Touchscreen events must continue to be delivered reliably and on time even if there is a large amount of bandwidth being consumed by other inter-domain communications between AD and CE.

Wi-Fi access

The Wi-Fi hardware is controlled by the AD or CD. The CE needs to use it for internet access, including connecting to a network. The Wi-Fi device can return data at high bandwidth, but also has a separate control channel. The control channel always needs to be available, even if traffic is being dropped due to bandwidth limitations in the inter-domain communication channel.

As the Wi-Fi is used for general internet access, sensitive information might be transferred between domains (for example, authentication credentials for a website the user is logging in to). Attackers who are snooping the inter-domain connection must not be able to extract such sensitive data from the inter-domain communications link.

(*Note that they may still be able to extract sensitive data from insecure connections over the wireless connection itself, or elsewhere in transit outside the vehicle; so any solution here is the best mitigation we can manage for the problem of a website being insecure.*)

Bluetooth access

The Bluetooth hardware might be attached to the AD or CD. The CE needs to be able to send data bi-directionally to other Bluetooth devices and also needs to be able to control the Bluetooth device, controlling pairing and other functions of the Bluetooth hardware.

To support the A2DP and HSP/HFP audio profiles it may be desirable to keep the CE in charge of decoding and encoding the audio streams coming from and directed to the Bluetooth devices. The AD will be responsible for mixing the output streams directed to the car speakers and capturing input streams (possibly with noise cancellation) from the car microphones.

The following diagrams depict the data and control flow when the Bluetooth device is attached to the AD.

Sending audio stream from BT to AD

```
BT device                 AD                      CE
    |  ---   attach   ---> |                       |
    |  ---------      encoded audio      ---------> |
    |                      | <--- decoded audio --- |
              (mixing decoded audio in AD)
```

Sending audio stream from AD to BT

```
BT device                 AD                      CE
    |  ---   attach   ---> |                       |
    |                      | ---- LPCM audio ---->  |
    |  <--------      encoded audio      ---------  |
```

The following diagram depicts the data and control flow when the Bluetooth device is directly attached to the CE instead.

```
BT device                          CE                        AD
    | --------- attach -----------> |                         |
```

```
    | <-------- control ----------  |                         |
    |                               |                         |
    | --------- encoded audio ----> |                         |
    |                               | ------- LPCM audio ---> |
    |                               | <------ LPCM audio ---- |
    | <-------- encoded audio ----- |
```

The following diagram depicts the data and control flow when the Bluetooth device is directly attached to the CD.

```
BT device                      CD                CE          AD
    | ---- attach -----------> |                  |           |
    | <--- control ----------  |                  |           |
    |                          | <---- scan ----- |           |
    |                          | ---- result ---> |           |
    |                          | <---- play ----- |           |
    |                          |                              |
    | ---- encoded audio ----> |                              |
    |                          | --------- LPCM audio ------> |
    |                          | <-------- LPCM audio ------- |
    | <--- encoded audio ----- |
```

Multiple variations are possible on this model.

Audio transfer

The audio amplifier hardware might be attached to the AD hardware, or might be set up as a separate hardware amplifier attached to the in-vehicle network. The CE needs to be able to send multiple streams of decoded audio output to the AD, to be mixed with audio output from the AD according to some prioritisation logic.

The decoded audio streams should be in LPCM format, but other formats may be negotiated by the domains using application specific APIs.

Metadata can be sent alongside the audio, such as track names or timing information.

Audio output needs predictable latency output, and for video conferencing it needs low latency as well; conversely, some level of packet loss is acceptable for audio traffic. However, the latency should not exceed a certain amount of time in some specific cases:

- Voice recognition systems provided through phone integration require that the maximum latency of the audio buffer from the time it gets captured by the microphone controlled by the AD to the time it gets delivered to the phone attached to the CE domain must not exceed 35ms.
- Text-to-speech systems provided through phone integration require that the maximum latency of the audio buffer from the time it is received by

the CE domain from the attached phone to the time it gets played back on the speakers attached to the AD must not exceed 35ms.

- The total round-trip time must not exceed 275ms when the phone is attached to the CE domain through a wired transports (for instance, USB CDC-NCM as used by CarPlay or the Android Open Accessory Protocol) and 415ms on wireless transports (WiFi in particular, Bluetooth A2DP is not recommended in this case).
- Bluetooth SCO can be used when there is a latency constraint. It will be lower quality, but the transfer time over the air is guaranteed. The whole audio chain needs to satisfy the latency condition though. This is why in some setup, the Bluetooth audio is routed directly to the AD amplifier. When this is the case, an API to enable this link is provided by the domain that owns the Bluetooth hardware. It can be the AD, or the CD embedding a Bluetooth stack. The API calls would be issued by the CE domain.

Video decoding

There might be a specific hardware video decoder attached to the AD hardware, which the CE operating system wishes to use for offloading decoding of trusted or untrusted video content. This is high bandwidth, but means that the output from the video decoder could potentially be directed straight onto a surface on the screen.

(See the appendix on Audio and video decoding for a discussion of options for video and audio decoding.)

Video or audio decoder bugs

The CE has a software video or audio decoder for a particular video or audio codec, and a security critical bug is found in this decoder, which could allow malicious video or audio content to gain arbitrary code execution privileges when it's decoded. An update for the Apertis operating system is released which fixes this bug, and users need to apply it to their vehicles. To reduce the window of opportunity for exploitation, this update has to be applied by the vehicle owner, rather than taking the vehicle into a garage (which could take weeks).

For example, like the series of exploitable bugs which affected the 'secure' media decoding library on Android[4] in 2015.

This means we cannot securely support decoding untrusted video or audio content in the AD, due to its slow software update cycle, unless we use a *hardware* video decoder which is specifically designed to cope with malicious inputs.

[4] https://en.wikipedia.org/wiki/Stagefright_(bug)

Streaming media

The media player backend on the CE accesses local files or internet streams and sends the streams to the Media Player HMI running in the AD. The CE might be able to perform demuxing, decoding or at least partly verifying the streams.

The AD might accept fully decoded streams, but the media file or stream is usually encoded and multiplexed. In some cases, the multiplexed stream can have synchronization sensitive metadata like subtitles. Therefore, if demuxing and decoding are performed in different domains, the AD should support multiple channels and mix the streams with time synchronization information.

It is also possible that the AD sends the stream to the CE. For example, in the case of Internet phone applications, the CE provides the HMI and needs to be able to capture video and audio streams from the AD, before encoding and multiplexing them on the CE.

When handling data streams that don't need strict synchronization, the bulk data transfer mechanism is recommended. For example, sharing still pictures does not require real time processing so it is not suited for the streaming media mechanism.

Downloads of firmware updates

An OTA update agent in the Connectivity domain downloads or retrieves from an attached USB stick firmware images as large as 20GB each and needs to share them with the Automotive domain where the FOTA backend can flash the attached devices.

Since firmware are very large, storing them twice should be avoided as the available space may not be sufficient to do so.

Offline and online map data

An offline map agent in the Connectivity domain downloads map data for offline usage by the navigation system running in the Automotive domain.

Conversely, an online map agent in the Connectivity domain handles requests from the Automotive domain for map tiles to download.

Phonebook integration

A phonebook agent in the Connectivity domain retrieves approximately 500 256×256px profile pictures, validates and re-encodes them to PNG and makes them available to the Automotive domain, possibly using an uncompressed zip file instead of sharing 500 files.

Tinkering vehicle owner on the network

The owner of a vehicle containing an Apertis device likes to tinker with it, and is probing and injecting signals on the connection between the AD and CE, or even replacing the CE completely with a device under their control. They should not be able to make the automotive domain do anything outside its normal operating range; for example, uncontrolled acceleration, or causing services in the domain to crash or shut down.

The tampering must be detectable by the vendor when the vehicle is serviced or investigated after an accident.

Tinkering vehicle owner on the boards

The owner of a vehicle containing an Apertis device likes to tinker with it, and has gained access to the bootloaders and storage for both the AD and CE boards. They have managed to add some custom software to the CE image, which is now sending messages to the AD which it does not expect. Or vice-versa. The domain receiving the messages must not crash, must ignore invalid messages, and must not cause unsafe vehicle behaviour.

The tampering must be detectable by the vendor when the vehicle is serviced or investigated after an accident.

Secure bootloading[5] itself is a separate topic.

Support multiple AD operating systems

The OEM for a vehicle wants to choose the operating system used in the AD — for example, it might be GENIVI Linux, or QNX, or something else. There is limited opportunity to modify this operating system to implement Apertis-specific features. Whichever CE or CD system is installed needs to interface to it. Each AD operating system may expose its underlying hardware and services with a variety of different non-standardised APIs which use push- and pull-style APIs for transferring data. The OEM wishes to be provided with an inter-domain communication library to integrate into their choice of AD operating system, which will provide all the functionality necessary to communicate with Apertis as the CE or CD operating system.

Before-market upgrades

The OEM for a vehicle has chosen a specific version of an operating system for their AD, and has initially released their vehicle with Apertis 17.09 on another domain, such as CE and/or CD. For the latest incremental version of this vehicle, they want to upgrade the other domain to use Apertis 18.06. The OS in the AD cannot be changed, due to having stricter stability and testing requirements than the other domains.

[5] https://em.pages.apertis.org/apertis-website/architecture/secure-boot/

After-market upgrades

A user has bought a vehicle which runs Apertis 17.09 in its CE. Apertis 18.06 is released by their car vendor, and their garage offers it as an upgrade to the user as part of their next car service. The garage performs this software upgrade to the CE, without having to touch the AD. It verifies that the system is operational, and returns the car to the user, who now has access to all the new features in Apertis 18.06 which are supported by their vehicle's hardware.

Testability

When developing a new vehicle, an OEM wants to iterate quickly on changes to the CE, but also wants to test them thoroughly for compatibility against a specific AD version, to ensure that the two domains will work together. They want this testing to include a number of valid and invalid conversations between the CE and AD, to ensure that the two domains implement error handling (and hence a large part of their security) correctly.

Malicious CE

Somehow, a third party application installed onto the CE manages to compromise a system service and gain arbitrary code execution privileges in the CE. It uses these privileges to send malicious messages to the AD. From the user's point of view, this could result in a loss of IVI functionality, and unexpected behaviour from vehicle actuators, but must not result in loss of control of the vehicle.

Malicious CD

Recent protocol failures have been discovered that allowed an attacker to take control of a device remotely. To mitigate this, the network management stack has been moved to a Connectivity Domain. The impact of those attacks must be minimised. While the CD functionality can be degraded, it must not result in loss of control of the vehicle.

After-market upgrade of a domain

A user has bought a vehicle containing a low-end Apertis device. They wish to upgrade to a more fully-featured Apertis device, and this hardware upgrade is offered by their garage. The garage performs the upgrade, which replaces the existing CE hardware with a new separate CE board. If the existing hardware combined the AD and CE on a single board or virtualised processor, the entire board is replaced with two new, separate boards, one for each domain (though as this is a complex operation, some garages or vendors might not offer it). If the existing hardware already had separate boards for the two domains, only the CE board is upgraded — this may be a service offered by all garages.

Power cycle independence of domains (CE down)

Due to a bug, the CE crashes. The AD must not crash, and must continue to function safely. It may display an error message to the user, and the user may lose unsaved data. Once the CE restarts, the AD should reconnect to it and reestablish a normal user interface. The CE should reboot quickly and the cross-domain state be restored as much as reasonable once restarted.

Any partially-complete inter-domain communications must error out rather than remaining unanswered indefinitely.

The same situation applies if both domains are booting simultaneously, but the CE is slower to boot than the AD, for example — the AD will be up before the CE, and hence must deal with not being able to communicate with it. See also Plug-and-play CE device.

Power cycle independence of domains (AD down, single screen)

On a system where the AD and CE are sharing a single screen, if the AD crashes, the CE must not crash, and may gracefully shut down, and only restart once the AD has finished rebooting. The AD should reboot quickly and the cross-domain state be restored as much as reasonable once restarted

Any partially-complete inter-domain communications must error out rather than remaining unanswered indefinitely.

The same situation applies if both domains are booting simultaneously, but the AD is slower to boot than the CE, for example — the CE will be up before the AD, and hence must deal with not being able to communicate with it. See also Plug-and-play CE device.

Power cycle independence of domains (AD down, multiple screens)

On a system with multiple output screens, if the AD crashes, the CE must not crash, and should continue to run on all its screens, as another user may be using the CE (without requiring any functionality from the AD) on one of the screens. Once the AD restarts, the CE should reconnect to it and reestablish a normal user interface on all screens. The AD should reboot quickly and the cross-domain state be restored as much as reasonable once restarted.

Any partially-complete inter-domain communications must error out rather than remaining unanswered indefinitely.

The same situation applies if both domains are booting simultaneously, but the AD is slower to boot than the CE, for example — the CE will be up before the AD, and hence must deal with not being able to communicate with it. See also Plug-and-play CE device.

Temporary communications problem

There is a temporary communications problem between a service on the AD and its counterpart on the CE. Either:

- The service (on the AD or CE) has crashed.
- There is a problem with the physical connection between the domains, such as dropped packets due to congestion; but both domains are still running fine.
- The entire domain or its inter-domain communications service has crashed.

The different situations can be detected by the parts of the stack which are still working

If a service has crashed, the inter-domain communication service should return an appropriate error code to the other domain, which could propagate the error to a calling application, or wait for the other domain to restart that service and try again.

If there is packet loss, the reliability in the inter-domain communication protocol should cause the lost packets to be re-sent. Services should wait for that to happen. If the communications problem continues longer than a timeout, the domains must assume that each other have crashed and behave accordingly.

If a domain has crashed, the other domain must wait for it to be restarted via its watchdog, as in Power cycle independence of domains (CE down).

In all cases, the domain which is still running must not shut down or enter a 'paused' state, as that would allow denial of service attacks.

New version of AD software

An OEM has released a vehicle with version A of their AD operating system, and version 15.06 of Apertis running in the CE. For the next minor update to their vehicle, the OEM has made a number of changes to the underlying AD software, but not to its external interfaces. They wish to keep the same version of Apertis running in the CE and release the vehicle using this version B of their AD operating system, and version 15.06 of Apertis.

New version of AD interfaces

An OEM has released a vehicle with version A of their AD operating system, and version 15.06 of Apertis running in the CE. For the next minor update to their vehicle, the OEM has made a number of changes to the underlying AD software, and has changed a few of its external interfaces and exposed a few more vehicle-specific features in new interfaces. They want to make appropriate modifications to Apertis to align it with these changed interfaces, but do not wish to make major modifications to Apertis, and wish to (broadly) stick with version 15.06. They will release the vehicle using this version B of their AD operating system, and a tweaked version 15.06 of Apertis.

In other words, this scenario applies only when the OEM has updated the AD, and wants to make a corresponding update to the CE. For the reverse scenario where the CE has been upgraded, it is required that the AD does not need to be updated: see Plug-and-play CE device and After market CE upgrades.

Unsupported AD interfaces

An OEM uses an AD operating system which exposes a large number of interfaces to various esoteric automotive components. Only a few of these components are currently supported by Apertis version A, which they are running in their CE. Apertis version B supports some more of these components, and exposes them in its SDK APIs. The OEM wishes to release a new version of the same vehicle, keeping the same version of the AD operating system, but using version B of Apertis and exposing the now-supported components in the SDK APIs.

However, some of the other components which are exposed by the AD operating system in its inter-domain interface cannot be securely supported by Apertis (for example, they may allow unrestricted write access to the in-vehicle network). These should not be accessible by the SDK APIs at any time.

Contacts sharing

A vehicle maintains an address book in its AD operating system, which stores some of the user's contacts on a removable SD card. The user interface, run by the CE, needs to be able to display and modify these contacts in the Apertis address book application.

Protocol compatibility

An older vehicle, using an old version A of some AD operating system was using a corresponding version A of Apertis in its CE. The CE operating system is upgraded to a recent version of Apertis, version B, by the garage when the vehicle is taken in for a service. This version of Apertis uses a much more recent version of the underlying software for the inter-domain communication protocol. It needs to continue to work with the old version A of the AD operating system, which is running a much older version of the protocol software.

kdbus protocol compatibility

If, for example, the inter-domain communication protocol is implemented using dbus-daemon in version A of the AD operating system, and in the corresponding version A of Apertis; and version B of Apertis uses kdbus instead of dbus-daemon, the two OSs must still communicate successfully.

Navigation system

A proprietary navigation system is running on the AD, with full access to the vehicle's navigation hardware, including inertial sensors and a GPS receiver. A tour application on the CE wishes to use location-based services, reading the vehicle's location from the navigation system on the AD, then requesting to the navigation service to set its destination to a new location for the next place in the tour. It sends a stream of points of interest to the navigation system to display on the map while the driver is navigating. This stream is not high bandwidth; neither are the location updates from the GPS.

Marshalling resource usage

The 'proxy' software on either side of the inter-domain connection which handles the low-level communication link is the first software in a domain to handle malicious input. If malicious input is sent to a domain with the intent of causing a denial of service in that software, the rest of the software in the domain should be unaffected, and should treat the connection as timing out or compromised. The behaviour of the proxy software should be confined so that it cannot use excess resources in the domain and hence extend the denial of service attack to the whole domain.

Feedback for malicious applications

If an application uses SDK APIs incorrectly (for example, by providing parameters which are outside valid ranges), it may be reported to the Apertis store as a 'misbehaving application' and scheduled for further investigation and possible removal from the Apertis store. Similarly if the inter-domain communication APIs are used incorrectly (for example, if the AD returns an error stating that input validation checks have failed for an API call).

This could also result in an application being blacklisted by the CE's application manager, disallowing it from running in future until it is updated from the Apertis store.

Compromised CE with delayed fix

An attacker has somehow completely compromised the CE operating system, and has root access to it. It will take the OEM a few weeks to produce, test and distribute a fix for the exploit used by the attacker, but vehicle owners would like to continue to use their vehicles, with reduced functionality (no CE domain) in the meantime, because the attack has not compromised the AD. The OEM has provided them with an authenticated method of informing the AD to shut down the CE and keep it shut down until an authenticated update has been applied and has fixed the exploit and removed the attacker from the CE (probably by overwriting the entire OS with a fresh copy). This update can only be applied at a garage, but in order to allow speedy deployment, the user can switch the AD to this stand-alone mode themselves, using a trusted input path to the AD.

Denial of service through flooding

A speedometer application bundle constantly requests vehicle speed information from the AD. Hundreds of requests are made per second. The AD ensures this does not affect overall system performance, potentially at the cost of its responsiveness to the speedometer application's requests.

(Note: This assumes that the corresponding denial of service rate limiting which is implemented in the SDK API used by the speedometer application has somehow failed or been bypassed. In reality, all SDK APIs are also responsible for implementing their own rate limiting as a first level of protection against denial of service attacks.)

Malicious CE UI

An attacker has somehow completely compromised the CE operating system, and has root access to it. They can display whatever they like on the graphics output from the CE, which is shared with that from the AD on a single screen. The attacker tries to replicate the AD UI on the CE's output and trick the user into entering personal data or security credentials in this faked UI, believing it to be the actual AD UI. There should be a way for the user to determine whether they are inputting details via a trusted path to the AD.

Plug-and-play CE device

In a particular vehicle, the CE device can be unplugged from the dashboard by the user, and passed around the car so that, for example, a rear seat passenger could play a game. This disconnects it from the AD, but it should continue to function with some features (such as Wi-Fi or Bluetooth) disabled until it is reconnected. Once reconnected to the dashboard it should reestablish its connections. See also, Power cycle independence of domains (CE down), Power cycle independence of domains (AD down, single screen), Power cycle independence of domains (AD down, multiple screens)

(Note: This is a much lower priority than other setups, but should still be considered as part of the overall design, even if the code for it will be implemented as a later phase.)

Connecting an SDK to a development vehicle

A developer is running the SDK as a standalone CE system in a virtual environment on a laptop. They connect the laptop to the AD physically installed in a development car using an Ethernet cable, and expect to receive sensor data from the car, using the sensors and actuators SDK API, which was previously returning mock results from the standalone system.

Connecting an SDK to a production vehicle

The developer wonders what would happen if they tried connecting their SDK laptop to the AD in a production vehicle. They try this, and nothing happens — they cannot get sensor data out of the vehicle, nor use any of its other APIs.

Security model

See the Security concept design[6] for general terminology including the definitions used for *integrity*, *availability*, *confidentiality* and *trust*.

Attackers

Vehicle's owner

The vehicle's owner may be an attacker. They have physical access to the vehicle, including its in-vehicle network, the physical inter-domain communications link, and the board or boards which the automotive domain (AD) and consumer-electronics domain (CE) are on. We assume they do not have the capabilities to perform invasive attacks on silicon on the boards. Specifically, this means that in a virtualised setup where the AD and CE are run as separate virtual machines on the same CPU, we assume the attacker cannot read or modify the inter-domain communications link between them.

However, we do assume that they can perform semi-invasive or non-invasive attacks[7] on silicon on the boards. This means that they could (with difficulty) extract encryption keys from a secure key store on the board. A secure key store may be provided by the Secure Boot design, but may not be present due to hardware limitations — if so, the vehicle's owner will be able to extract encryption keys from the device more easily.

> As of February 2016, the Secure Boot design is still forthcoming

The vehicle's owner may wish to attack their vehicle in order to get access to licenced content which they would otherwise have to pay for.

> See the Conditional Access design[8]

We assume they do not want to take control of the vehicle, or to gain arbitrary code execution privileges — they can drive the vehicle normally, or develop and choose to install their own application bundle for this.

Passenger

The passenger is a special kind of third party attacker (Third parties), who additionally has access to the in-vehicle network. This may be possible if, for

[6] https://em.pages.apertis.org/apertis-website/concepts/security/
[7] http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.html
[8] https://em.pages.apertis.org/apertis-website/concepts/conditional_access/

example, the Apertis device in the vehicle is removable so it can be passed to a passenger, exposing a connector behind it.

The passenger may be trying to access confidential information belonging to the vehicle owner (if a multi-user system is in use).

Third parties

Any third party may be an attacker. We assume they have physical access to the exterior of the vehicle, but not to anything under the bonnet, including the in-vehicle network, the physical inter-domain communications link, and the board or boards which the domains are on. This means that all garage mechanics must be trusted. They do, however, have access to all communications into and out of the vehicle, including Bluetooth, 4G, GPS and Wi-Fi.

We assume any third party attacker can develop and deploy applications, and convince the owner of a vehicle to install them. These applications are subject to the normal sandboxing applied to any application installed on an Apertis system. These applications are also subject to the normal Apertis store validation procedures, but we assume that a certain proportion of malicious applications may get past these procedures temporarily, before being discovered and removed from the store.

We assume that a third party attacker does not have access to the Apertis store servers. This means that all staff who have access to them must be trusted.

A third party attacker may be trying to:

- Access confidential information belonging to the vehicle owner.
- Compromise the integrity of the vehicle's control system (the automotive domain). For example, to trigger unintended acceleration or to change the radio channel to spook the driver.
- Compromise the integrity of the CE domain to, for example, make it part of a botnet, or cause it to call premium rate numbers owned by the attacker to generate money.
- Compromise the availability of the vehicle's control system (the automotive domain) to bring the vehicle to a halt.
- Compromise the availability of the vehicle's infotainment system (the CE domain) to cause a nuisance to the driver or passengers.
- Compromise the confidentiality of the device key (see the Conditional Access design[9]) in order to extract licenced content (for example, music) from application bundles.

[9] https://em.pages.apertis.org/apertis-website/concepts/conditional_access/

Trusted dealer

As above, all authorized vehicle dealers, garages or other sale/repair locations have to be trusted, as they have more unsupervised access to the vehicle's hardware, and more capabilities, than the vehicle owner, passenger or a third party.

Security domains

- Automotive domain
  - There may be security sub-domains within the automotive domain, but for the purposes of this design it is treated as a black box
- Consumer-electronics domain:
  - Each application sandbox in the consumer-electronics domain
  - CE domain operating system (this includes all the daemons for the SDK APIs — these are technically separate security domains, but since they communicate only with sandboxes and the CE domain proxy, this makes the model more complex for no analytical advantage)
  - CE domain proxy for the inter-domain communication
- Connectivity domain:
  - Connectivity domain handles the communication between AD and the outer world.
  - Different protocol stacks.
  - CD domain proxy for communicating with AD
- Other devices on the in-vehicle network, and the outside world
- Hypervisor (if running as virtualised domains)

Security model

- Domains must assume that the inter-domain communication link has no confidentiality or integrity, and is controlled by an attacker (a man in the middle with the ability to modify traffic)
  - This means they must not trust any traffic from other devices on the network
- The AD, CD and CE operating systems must assume all input from external sources (Wi-Fi, Bluetooth, GPS, 4G, etc.) is malicious
- The CE operating system may assume all API calls from the AD (as proxied by the CE proxy) are *not* controlled by an attacker, assuming they have come over an authenticated channel which guarantees integrity

between the AD and CE proxy; in other words, the AD must not deny confidentiality or integrity to the CE

- The AD may deny availability to the CE operating system, by closing the inter-domain link in response to the user disabling the CE while waiting for a critical security update
- The AD must assume all API calls from the CE are malicious, in case the CE has been compromised
- The CE must assume that all input and output from third party applications in sandboxes is malicious, including all their API calls
- If a hypervisor is present:
  - The AD and CE operating systems may assume all control calls from the hypervisor are *not* controlled by an attacker
  - The hypervisor must assume all input from the CE is malicious
  - The hypervisor may assume that all input from the AD is *not* malicious
    - Note that, when combined with the fact that the AD cannot be updated easily, this makes security bugs in the AD extremely critical and extremely hard to fix
- Tampering with any domain software must be detectable even if it is not preventable (tamper evidence)
- If one vehicle is attacked and compromised, the same effort must be required to compromise other vehicles

Non-use-cases

Production CE domain used in multiple configurations

A production CE domain operating system cannot be used in multiple configurations, for example as both an operating system running on one CPU of a two-CPU board shared with the automotive domain OS; and then as an image running on a separate board connected to an in-vehicle network with other devices connected.

This requirement would mean that the inter-domain communications system would have to support runtime reconfiguration, which would be a vector for protocol-downgrade attacks while bringing no major benefits. An attacker could try to trick the CE domain into believing it was in (for example) a virtualised configuration when it wasn't, which could potentially disable its encryption, due to the assumption the domain could make about its inter-domain communications link having inbuilt confidentiality.

Requirements

Separated transport layer

The transport layer for transmitting inter-domain communications between the domains must be separated from the APIs being transported, in order to allow for different physical links between the domains, with different security properties.

Transport to SDK APIs

Support a configuration where the CE is running in a virtual machine with the Apertis SDK, so the peer (which would normally be the AD) is a mock AD daemon running against the SDK.

See Standalone setup.

Transport over virtio

Support a configuration where the CE and AD communicate over a virtio link between two virtual machines under a hypervisor.

See Basic virtualised setup.

Transport over a private Ethernet link

Support a configuration where the CE and AD are on separate CPUs and communicate over a point-to-point Ethernet link.

See Separate CPUs setup, Separate boards setup.

Transport over a private Ethernet link to a development vehicle

Support a configuration where the CE is running in an SDK on a laptop, and the AD is running in a developer-mode Apertis device in a vehicle, and the two communicate over a wider shared Ethernet.

See Connecting an SDK to a development vehicle.

Transport over a shared Ethernet link

Support a configuration where the CE and AD are on separate CPUs are are both connected to some wider shared Ethernet.

See Separate boards setup with other devices, Multiple CE domains setup.

Transport over Unix Domain Socket

Support a configuration where AD and CE are on the same host running as Linux containers and connected via UDS. The same transport can be used on OEM deployments and on SDK environments.

See Linux container setup, Multiple CE domains setup.

Message integrity and confidentiality in transport layer

Some of the possible physical links between domains do not guarantee integrity or confidentiality of messages, so these must be implemented in the software transport layer.

See Separate CPUs setup, Separate boards setup, Separate boards setup with other devices, Multiple CE domains setup, Wi-Fi access.

Reliability and error checking in transport layer

Some of the possible physical links between domains do not guarantee reliable or error-free transfer of messages, so these must be implemented in the software transport layer.

See Separate boards setup, Separate boards setup with other devices, Multiple CE domains setup.

Mutual authentication between domains

An attacker may interpose on the inter-domain communications link and attempt to impersonate the AD to the CE, or the CE to the AD. The domains must mutually authenticate before accepting any messages from each other.

See Tinkering vehicle owner on the network.

Separate authentication for developer and production mode devices

A CE running in an SDK must be able to connect to and authenticate with an AD running in a vehicle which is in a special 'developer mode'. If the same CE is connected to a production vehicle, it must not be able to connect and authenticate.

See Connecting an SDK to a development vehicle, Connecting an SDK to a production vehicle.

Individually addressed domains

In order to support multiple CE domains using the same automotive domain, each domain (consumer–electronics and automotive) must be individually addressable. The system must not assume that there are only two domains in the network.

See Multiple CE domains setup.

Traffic control for latency

In order to support delivery of touchscreen events with low latency (so that UI responsiveness is not perceptibly slow for the user), the system must guarantee a low latency for all communications, or provide a traffic control system to allow certain messages (for example, touchscreen messages) to have a guaranteed latency.

See Touchscreen events.

Traffic control for bandwidth

In order to prevent some kinds of high bandwidth message from using all the bandwidth provided by the physical link, the system must provide a traffic control system to ensure all types of message have fair access to bandwidth (where 'fairness' is measured according to some rigorous definition).

This may be implemented by separating 'control' and 'data' streams (see sections 2.4 and 2.5), or by applying traffic control algorithms.

See Wi-Fi access, Bluetooth access.

Traffic control for frequency

In order to prevent denial of service due to a service sending too many messages at once (so the communication overheads of those messages start to dominate bandwidth usage), the system must guarantee fair access to enqueue messages. This is subtly different from fair access to bandwidth: service A sending 100000 messages of 1KB per second and service B sending 1 message of 100000KB per second have the same bandwidth requirements; but if the inter-domain link saturates at 100000KB per second, some of the messages from service A must be delayed or dropped as the messaging overheads exceed the bandwidth limit.

See Denial of service through flooding.

Separation of control and data streams

Certain APIs will need to provide data and control streams separately, with different latency and bandwidth requirements for both. The system must support multiple streams; this may be via an explicit separation between 'control' and 'data' streams, or by applying traffic control algorithms.

See Wi-Fi access, Bluetooth access, Audio transfer, Video decoding.

No untrusted access to AD hardware

The entire point of an inter-domain communication system is to isolate the CE from direct access to sensitive hardware, such as vehicle actuators or hardware with direct memory access (DMA) rights to the AD CPU's memory. This must apply equally to decoder hardware — decoders or other hardware handling untrusted data from users must not be trusted by the AD if the CE can send untrusted user data to it, unless it is certified as a security boundary, able to handle malicious user input without being exploited.

Specifically, this means that hardware decoders must only access memory which is accessible by the AD CPU via an input/output memory management unit (IOMMU), which provides memory protection between the two, so that the hardware decoder cannot access arbitrary parts of memory and proxy that access to a malicious or compromised application in the CE.

Note that it is not possible to check audio or video content for 'badness' before sending it to a decoder, as that entails doing the full decoding process anyway.

See Audio transfer, Video decoding, Video or audio decoder bugs, Connecting an SDK to a production vehicle.

Trusted path for users to update the CE operating system

There must exist a trusted path from the user to the system updater in the CE, or to a component in the AD which will update the CE. The user must always have access to this update system (it must always be *available*).

This trusted path may also be used by garages to upgrade the CE when servicing a vehicle; or a different path may be used.

See Video or audio decoder bugs, After market CE upgrades, Malicious CE UI.

Safety limits on AD APIs

The automotive domain must apply suitable safety limits to all of its APIs, which are enforced within the AD, so that even if a properly authenticated and trusted CE makes an API call, it is ignored if the call would make the AD do something unsafe.

In this case, 'safety' is defined differently for each actuator or combination of actuator settings, and will vary between AD implementations. It might not be possible to detect all unsafe situations (in the sense of an unsafe situation which could lead to an accident).

See Tinkering vehicle owner on the boards, Malicious CE.

Rate limiting on control messages

The inter-domain service in the CE and AD should impose rate limiting on control messages coming from the CE, to avoid a compromised service in the CE from using a denial of service attack to prevent other messages being transmitted successfully.

This should be in addition to rate limiting implemented in the SDK APIs in the CE themselves, which are expected to be the first line of defence against denial of service attacks.

See Denial of service through flooding.

Ignore unrecognised messages

Both the CE and AD must ignore (and log warnings about) inter-domain communication messages which they do not recognise. If the message expects a reply, an error reply must be sent. The domains must not, for example, shut down or crash when receiving an unrecognised message, as that would lead to a denial of service vulnerability.

See Tinkering vehicle owner on the boards, Malicious CE.

Portable transport layer

The transport layer must be portable to a variety of operating systems and architectures, in order that it may be used on different AD operating systems. This means, for example, that it must not depend on features added to very recent versions of the Linux kernel, or must have fallback implementations for them.

See Support multiple AD operating systems.

Support push mode and pull mode communications

The CE must be able to use pull mode communications with the AD, where it makes a method call and receives a reply; and push mode communications, where the AD emits a signal for an event, and the CE receives this.

See Support multiple AD operating systems.

OEM AD integration API

In order to allow any OEM to connect their AD to the system, there must be a well defined API which they connect their OEM-specific APIs for vehicle functionality to, in order for that functionality to be exposed over the inter-domain communication link.

This API must support an implementation which uses the services in the Apertis SDK.

See Support multiple AD operating systems, Standalone setup.

Flexibility in OEM AD integration API

As the functionality exported by different ADs differs, the integration API for connecting it to the inter-domain communication system must be a general one — it must not require certain functionality or data types, and must support functionality which was not initially expected, or which is not currently supported by any CE. This functionality should be exposed on the inter-domain communications link, in case future versions of the CE can take advantage of it.

See Support multiple AD operating systems, Before market CE upgrades, After market CE upgrades, New version of AD software, New version of AD interfaces.

Inflexibility in OEM AD integration API

The OEM AD integration API must not allow access to arbitrary services or APIs on the AD. It must only allow access to the services and APIs explicitly exposed by the OEM in their use of the integration API.

See Unsupported AD interfaces.

Service discovery

Domains should be able to detect where specific services are hosted in case of multiple CE domains. If a service is moved from one CE domain to another CE domain, other domains should not require any reconfiguration. CE domains should not be able to spoof services that are meant to be provided by the AD.

Stability in inter-domain communications protocol

As the versions of the AD and CE change at different rates, the inter-domain communications protocol must be well defined and stable — it must not change incompatibly between one version of the CE and the next, for example.

If the protocol uses versioning to add new features, both domains must support protocol version negotiation to find a version which is supported if the latest one is not.

See Before market CE upgrades, After market CE upgrades, New version of AD software, Unsupported AD interfaces, Protocol compatibility.

Testability of protocols

All IPC links in the inter-domain communications system must be testable individually, without requiring the other parts of the system to be running. For example, the link between applications and SDK API services must be testable without running an automotive domain; the link between SDK API services and the inter-domain interface at the boundary of the CE domain must be testable without running an automotive domain; etc.

See Testability, New version of AD software, Unsupported AD interfaces.

Testability of protocol parsers and writers

All protocol parsers and writers in the inter-domain communications system must be testable individually, using unit tests and test vectors which cover all facets of the protocol. These tests must include negative tests — checks that invalid input is correctly rejected. For example, if a protocol requires a certificate to authenticate a peer, a test must be included which attempts a connection with different types of invalid certificate.

See Testability, New version of AD software, Unsupported AD interfaces.

Testability of processes

The code implementing all processes in the inter-domain communications system must be testable individually, without having to run each process as a subprocess in a test harness (because this makes testing slower and error prone). This means implementing each process as a library, with a well defined and documented API, and then using that library in a trivial wrapper program which hooks it up to input and output streams and accepts command line arguments.

See Testability, New version of AD software, Unsupported AD interfaces.

CE system services separated from transport layer

There must be a trust boundary between each service on the CE which has access to the inter-domain communication link, and the service which provides access to the inter-domain communications link itself. The inter-domain service should validate that messages from a service are related to that service (for example, by having a whitelist of types of message which each service can send).

This limits the potential for escalation if service A is exploited — then the attacker can only use the inter-domain service to impersonate A, rather than to impersonate all services in the CE. It also allows the resource usage of the inter-domain service to be limited, to limit the impact of a denial of service attack on it.

See Malicious CE, Marshalling resource usage.

No dependency on CE specific hardware

As the CE hardware may be upgraded by a garage at some point, the inter-domain communications should not depend on specific identifiers in this hardware, such as an embedded cryptographic key. Such keys may be used, but the AD should accept multiple keys (for example, all keys signed by some overall key provided by Apertis to all OEMs), rather than only accepting the specific key from the hardware it was originally run against.

This requirement may also be satisfied by including provisions for updating the copy of a key in the AD if such a dependency on a specific CE key is a sensible implementation choice.

See After market upgrade of a domain.

Immediate error response if service on peer is unavailable

If a service on the peer has crashed or is unresponsive, but the peer itself (including its inter-domain communications link) is still responsive, that peer should return an error to the other domain, which should propagate it to any caller of SDK APIs which use the failing service. An error response must be returned, otherwise the caller will time out.

See Power cycle independence of domains (CE down), Power cycle independence of domains (AD down, single screen), Power cycle independence of domains (AD down, multiple screens), Plug-and-play CE device

Immediate error response if peer is unavailable

If the peer has crashed, or is not currently connected to the physical inter-domain communications link (either because it has been unplugged or due to a fault), the other peer must generate a local error response in the inter-domain service and return that to any caller of SDK APIs which require inter-domain communications. An error response must be returned, otherwise the caller will time out.

See Power cycle independence of domains (CE down), Power cycle independence of domains (AD down, single screen), Power cycle independence of domains (AD down, multiple screens), Plug-and-play CE device

Timeout error response if peer does not respond

If the peer is unresponsive to a particular inter-domain message, the other peer must generate a local error response in the inter-domain service and return that to the caller of the SDK API which required inter-domain communications. An error response must be returned, otherwise the caller will wait for a response indefinitely (or have to implement its own timeout logic, which would be redundant).

See Power cycle independence of domains (CE down), Power cycle independence of domains (AD down, single screen), Power cycle independence of domains (AD down, multiple screens), Plug-and-play CE device

All inter-domain communications APIs are asynchronous

As inter-domain communications may have some latency, or may time out after a number of seconds, all SDK APIs which require inter-domain communications must be asynchronous, in the GLib sense[10]: the call must be started, a handler for its response added to the caller's main loop, and the caller must continue with other tasks until the response arrives from the other domain.

[10] https://developer.gnome.org/gio/stable/GAsyncResult.html

This encourages UIs to be written to not block on SDK API calls which might take multiple seconds to complete, as during that time, the UI would not be redrawn at all, and hence would appear to 'freeze'.

See Temporary communications problem.

Reconnect to peer as soon as it is available

If a domain has crashed and restarted, or was disconnected from the inter-domain communications link and then reconnected, the domain must reconnect to its peer as soon as the peer can be found on the network. If, for example, both domains had crashed, this may involve waiting for the peer to connect to the network itself.

See Plug-and-play CE device.

External domain watchdog

Both domains must be connected to an external watchdog device which will restart them if they crash and fail to restart themselves.

The watchdog must be external, rather than being the other domain, in case both domains crash at the same time.

See Power cycle independence of domains (CE down), Power cycle independence of domains (AD down, single screen), Power cycle independence of domains (AD down, multiple screens).

Reporting system for malicious applications

There should exist a trusted path from the application launcher in the CE to the Apertis store to allow the launcher to provide feedback about applications which are detected to have done 'malicious' things, such as called an SDK API with parameters which are obviously out of range.

If such a path exists, the inter-domain service in the CE must be able to detect error responses from the AD which indicate that malicious behaviour has been detected and rejected, and must be able to forward those notifications to the reporting system.

See Feedback for malicious applications.

Ability to disable the consumer–electronics domain

There must exist a trusted path to a setting in the AD to allow the vehicle owner to disable the CE because it has been compromised, pending taking the vehicle to a trusted dealer to install an update.

As well as preventing booting the CE, this must disable all inter-domain communications from within the inter-domain service in the AD.

See Compromised CE with delayed fix.

Tamper evidence

If the CE or AD, or communications between them are tampered with by an attacker, it must be possible for an investigator (who is trusted by and has access to tools provided by the OEM) to determine that the software or hardware was modified — although it might not be possible for them to determine *how* it was modified. This will allow for liability to be attributed in the event of an accident or warranty claim.

See Tinkering vehicle owner on the network, Tinkering vehicle owner on the boards.

No global keys in vehicles

The security which protects the inter-domain communication system (including any trusted boot security) must use unique keys for each vehicle, and must not have a global key (one which is the same in all vehicles) as a single point of failure.

This means that if an attacker manages to compromise one vehicle, they must not be able to learn anything (any keys) which would allow them to compromise another vehicle with less effort.

See Tinkering vehicle owner on the network, Tinkering vehicle owner on the boards.

Existing inter-domain communication systems

As this is quite a unique problem, we know of no directly comparable systems. More generally, this is an instance of a distributed system, and hence similar in some respects to a number of existing remote procedure call systems or distributed middleware systems.

If comparisons with specific systems would be beneficial, they can be included in a future revision of this document.

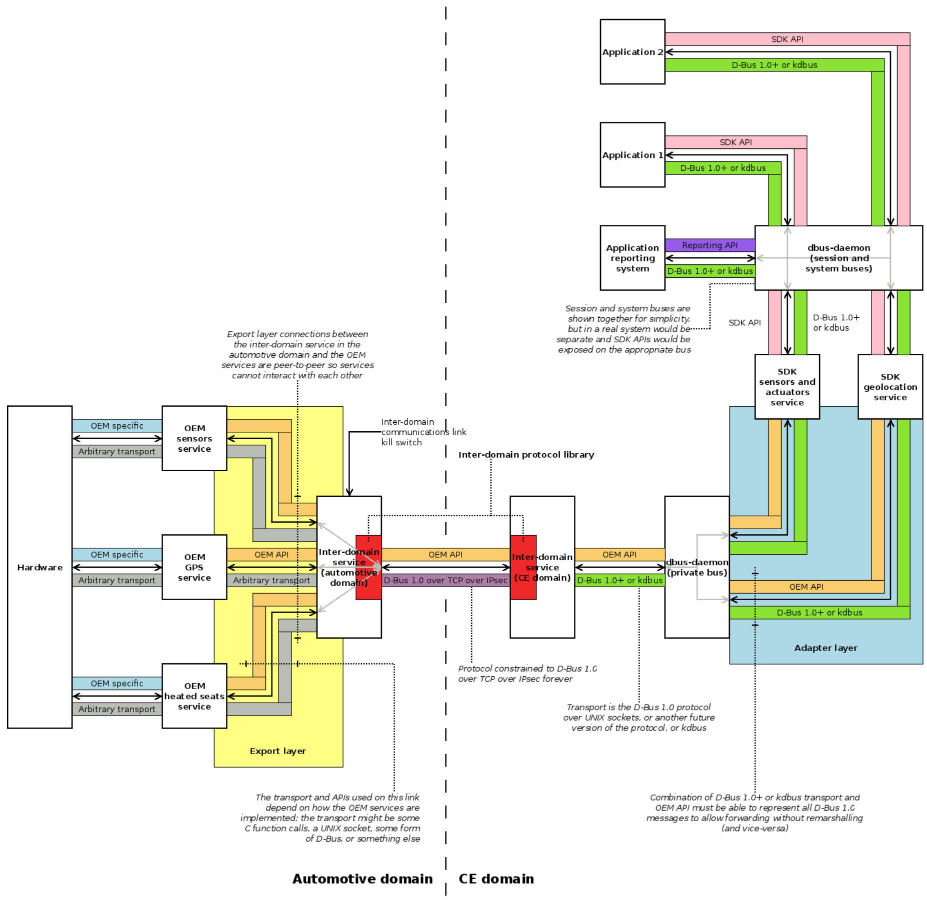
Open question: Are there any relevant existing systems to compare against?

Approach

Based on the [above research][Existing domain communications system] and Requirements, we recommend the following approach as an initial sketch of an inter-domain communication system.

Overall architecture

In the following figure, each box represents a process, and hence each connection between them is a trust boundary.



Apertis IDC architecture. The 'OEM specific' APIs are also known as 'native OEM APIs'; and the 'OEM API' is also known as the 'Apertis automotive API'. For more information on the export and adapter layer, see Automotive domain export layer and Consumer-electronics domain adapter layer.

APIs from the automotive domain are exported by an *export layer* (Automotive domain export layer) as D-Bus objects on the inter-domain communications link. This link runs a known version of the D-Bus protocol (and requires backwards compatibility indefinitely) between an *inter-domain service* process in each domain (Protocol library and inter-domain services). The inter-domain service in the CE domain sends and receives D-Bus messages for the objects exported by the automotive domain, and proxies them to a private bus in the

CE domain. SDK services in the CE domain connect to this bus, and an *adapter layer* Consumer-electronics domain adapter layer in each service converts the APIs from the automotive domain to the SDK APIs used in the version of Apertis in use in the CE domain. These SDK APIs are exported onto the normal D-Bus session bus, to be used by applications (Flow for a given SDK API call).

The export layer and adapter layer provide abstraction of the APIs from the automotive domain: the export layer converts them from C APIs, QNX message passing, or however they are implemented in the automotive OS, to a D-Bus API which is specific to that OEM, but which has stability guarantees through use of API versioning (Interaction of the export and adapter layers). The adapter layer converts from this D-Bus API to the current version of the Apertis SDK APIs. Both layers are OEM-specific.

The use of the D-Bus protocol throughout the system means that between the export layer and the adapter layer, message contents to not need to be remarshalled — messages only need their headers to be changed before they are forwarded. This should eliminate a common cause of poor performance (remarshalling).

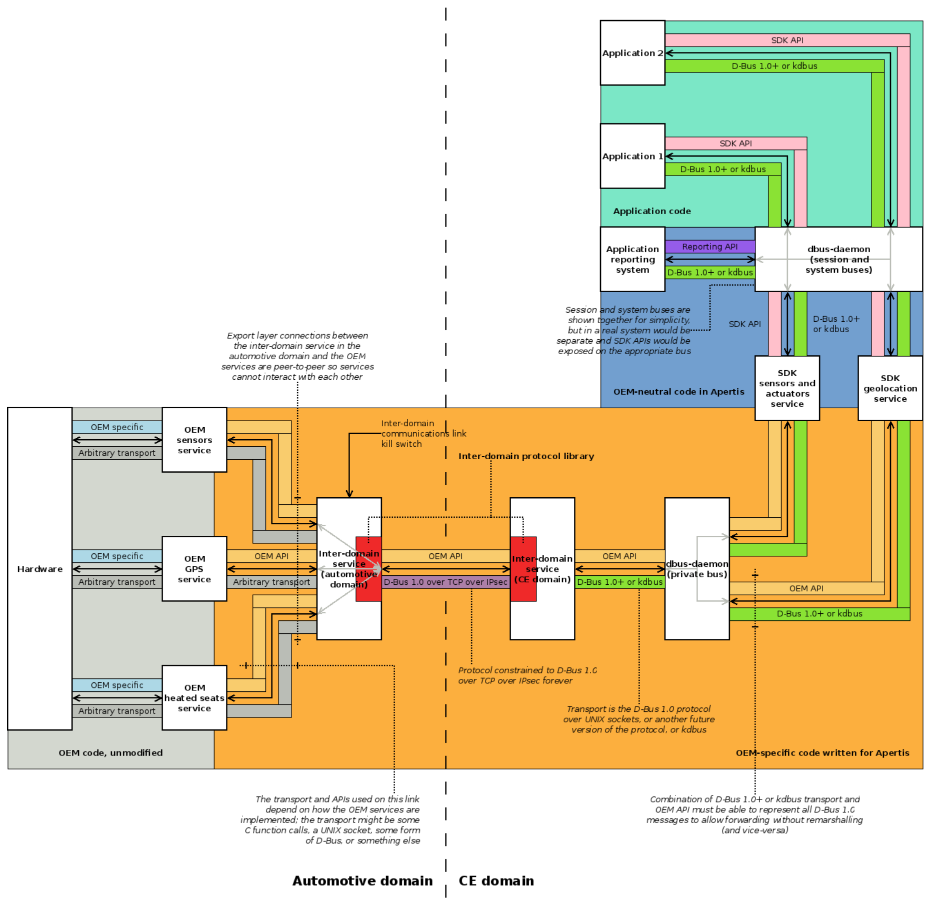
High-bandwidth Data connections are provided in parallel with the *control connection* which runs this D-Bus protocol (Control protocol). They use TCP, UDP or Unix sockets, and are opened between the two inter-domain services on request. Applications and services must define their own protocols for communicating over these links, which are appropriate to the data being transferred (for example, audio data or a Bluetooth file transfer).

Authentication, confidentiality and integrity of all inter-domain communications (the control connection and data connections) are provided by using IPsec as the bottom layer of the protocol stack (Encryption). The same protocol stack is used for all configurations of the two domains (from a standalone CE domain through to multiple CE domains on a shared network with an automotive domain), to ensure that the same code path is used for all configurations and hence is widely tested (Configuration designs).

Addressing and discovery of domains, before the initial connection between them, is provided by IPv6 neighbour discovery (Traffic control).

Traffic control is implemented in the CE domain using standard Linux kernel traffic control mechanisms, with the policy specified by the inter-domain service (section 8.4). It is applied for the control connection and for each data connection separately, as they are all separate TCP or UDP connections.

The only exception from the above is Linux container setup which uses Unix Domain Sockets as a trusted and reliable bottom transport layer instead of IPsec. In this case, there is no need for traffic control. Addressing and discovery of local domains in Linux container setup is based on common directories created and shared outside of the containers by the container manager.



Responsibilities for areas of code in the IDC architecture

Security domains

As process boundaries are the only way of enforcing trust boundaries, each of these security domains corresponds to at least one separate process in the system.

- Inter-domain service in the automotive domain. We recommend that this remains a separate security domain from the rest of the services and software running in the AD. This allows it to be isolated from other components to reduce the attack surface exposed by the AD.
- Rest of the automotive domain: as mentioned in Security domains, the automotive domain is essentially a black box.
- Each application sandbox in the consumer–electronics domain.
- Inter-domain service in the consumer–electronics domain.

- Each service for an SDK API in the consumer–electronics domain. The trust boundaries between them may not be enforced strongly (as all services in the consumer–electronics domain are considered as trusted parts of the operating system), but their trust boundaries with the inter-domain service should be enforced, and the inter-domain service should consider them as potentially compromised.
- Other devices on the in-vehicle network, and the outside world.
- Hypervisor (if running as virtualised domains).

Protocol design

The protocol for communicating data between the domains has two *planes*: the control plane, and the data plane. They have different requirements, but both require addressing, routing, mutual authentication of peers, confidentiality of data and integrity of data. In addition, the control plane must have bi-directional, in-order transmission, framing, reliability and error detection. Conversely, the data plane must have multiplexing, and the ability to apply traffic control to each of its connections (Traffic control).

The control plane is used for sending control data between the domains — these are the method calls which form the majority of inter-domain communications. They require low latency, and are low bandwidth. The [control protocol][Control protocol] itself provides push and pull method call semantics, and allows for new data connections (Data connections) to be opened. Only one control connection exists between a pair of domains, and it is always connected.

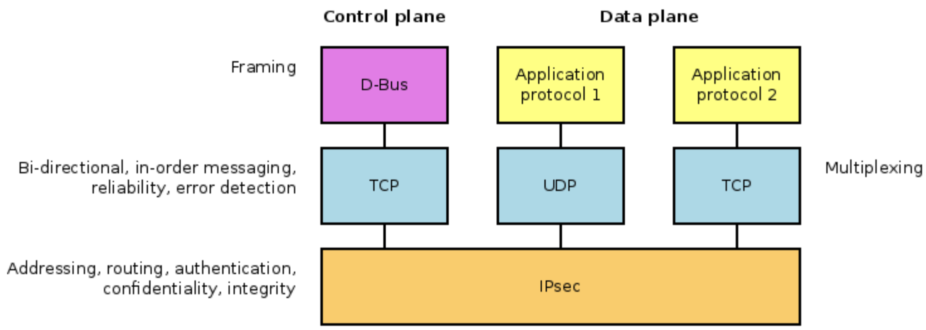
The data plane is used for high bandwidth data, such as video or audio streams, or Wi-Fi, 4G or Bluetooth downloads. The latency requirements are variable, but all connections are high bandwidth. The inter-domain communication system provides a plain stream for each data plane connection, and services must implement their own protocol on top which is appropriate for the specific type of data being transmitted (for example, audio or video streaming; or Wi-Fi downloads). Data connections are created between two domains on demand, and are closed after use.

IPsec versus TLS

An important design decision is whether to use IPsec[11] or TLS[12] (and DTLS) for providing the security properties of the inter-domain connection.

If IPsec is used (following figure), it forms the bottom layer of the protocol hierarchy, and implements addressing, routing, mutual authentication, confidentiality and integrity for *all* connections in the control and data planes.

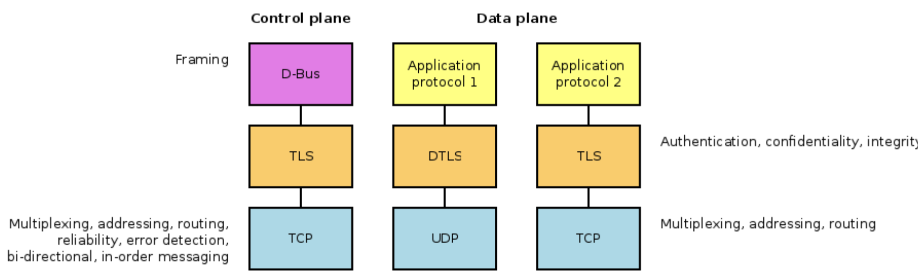
[11] https://en.wikipedia.org/wiki/IPsec
[12] https://en.wikipedia.org/wiki/Transport_Layer_Security



Protocol stacks for control and data planes if using IPsec.

If TLS is used (Following figure), it forms the layer just below the application protocols in the protocol hierarchy — the control plane would use a single TLS over TCP connection; and the data plane would use multiple TLS over TCP or DTLS over UDP connections. TLS (and hence DTLS — they have the same security properties) implements mutual authentication, confidentiality and integrity, but only for a single connection; each new connection needs a new TLS session.

The chief advantage of IPsec is its transparency: any protocol can be tunnelled using it, without needing to know about the security properties it has. However, to do this, IPsec needs to be supported by both the AD and CE kernels. Some automotive operating systems may not support IPsec (although, as a data point, QNX seems to).



Protocol stacks for control and data planes if using TLS.

A 2003 review of the IPsec protocol[13] identified a number of problems with it. However, since then, it has been updated by RFC 4301[14], RFC 6040[15] and RFC 7619[16]. These should be evaluated and the overall protocol security determined. In contrast, the security of TLS has been well studied, especially in recent years after the emergence of various vulnerabilities in it. TLS has the advantage that it is a smaller set of protocols than IPsec, and hence easier to study.

[13] https://www.schneier.com/cryptography/archives/2003/12/a_cryptographic_eval.html
[14] https://tools.ietf.org/html/rfc4301
[15] https://tools.ietf.org/html/rfc6040
[16] https://tools.ietf.org/html/rfc7619

Open question: What is the security of the IPsec protocol in its current (2015) state?

Performance-wise, TLS requires a handshake for each new connection, which imposes connection latency of at least one round trip (assuming use of TLS session resumption[17]) for each new connection (on top of other latency such as the TCP handshake). It is not possible to use a single TLS session and multiplex connections within it, as this puts the protocol reliability (TCP retransmission) below the multiplexing in the protocol stack, which makes the multiplexed connection prone to head of line blocking[18], which seriously impacts performance, and allows one connection to perform a denial of service attack on all others it is multiplexed with. IPsec has the advantage of not requiring this handshake for each connection, which significantly reduces the latency of creating new connections, but does not affect their overall bandwidth once they have reached a steady state.

Open question: What is the performance of TCP and UDP over IPsec, TLS over TCP and DTLS over UDP on the Apertis reference hardware?

Overall, we recommend using IPsec if it is expected to be supported by all automotive domain operating systems which will be used with Apertis systems. Otherwise, if an AD OS might not support IPsec, we recommend using TLS over TCP and DTLS over UDP for *all* configurations. We do *not* recommend providing a choice for OEMs between IPsec and TLS, as this doubles the possible configurations (and hence testing) of a part of the system which is both complex and security critical.

The remainder of this document assumes that IPsec is chosen. Throughout, please read 'IPsec' as meaning 'the IPsec protocol stack or the TLS protocol stack'.

Configuration designs

The physical links available between the domains differ between configurations of the domains, as do their properties. For some configurations (Standalone setup, Basic virtualised setup, Linux container setup) confidentiality and integrity of the inter-domain communications protocol are not strictly necessary, as the physical link itself cannot be observed by an attacker. However, for the other configurations, these two properties are important.

Since the first two configurations are the ones which are typically used for development, we suggest implementing confidentiality and integrity for them anyway, regardless of the fact it's not strictly necessary. This avoids the situation where the code running on production configurations is vastly different from that running on development configurations. Such a situation often leads to inadequate testing of the production code.

[17] https://tools.ietf.org/html/rfc5077
[18] https://en.wikipedia.org/wiki/Head-of-line_blocking

This should be weighed against the potential performance gains from eliminating encryption from those connections, and the potential gains in debuggability (for the Standalone setup and Linux container setup) by being able to inspect network traffic without needing to extract the encryption key.

Open question: What trade-off do we want between performance and testability for the different transport layer configurations?

Standalone setup

IPsec running on a loopback interface[19] to a service running in the SDK which mocks up the inter-domain service running in the AD. The security properties it provides are technically not needed, as the standalone setup is for development and is ignored by the security model.

Even though there are only two peers communicating, they will both have and use a full addressing scheme (Addressing and peer discovery).

Basic virtualised setup

A virtio-net connection must be set up in the CE and AD virtual guests, using a private network containing those two peers. If the AD cannot be modified to enable a virtio-net connection, a normal virtualised Ethernet connection must be used.

> Virtio-net is the name of the KVM paravirtualised network driver (*http://www.linux-kvm.org/page/Virtio*). Similar paravirtualised drivers exist for most hypervisors; so an appropriate one for the hypervisor should be used. For simplicity, this document will use 'virtio-net' to refer to them all.

In either case, the transport layer will use IPsec between the two. The security properties it provides are technically not needed for a virtualised configuration, as the security model guarantees that the hypervisor maintains confidentiality and integrity of the connection.

Even though there are only two peers on the network, they will both have and use a full addressing scheme (Addressing and peer discovery).

Separate CPUs setup

A normal Ethernet connection must be used to connect the AD and CE on a private network. IPsec will be used over this Ethernet link, providing all the necessary transport layer properties.

Even though there are only two peers on the network, they will both have and use a full addressing scheme, described below.

Separate boards setup

Same as for the separate CPUs setup.

[19] https://en.wikipedia.org/wiki/Loopback#Virtual_loopback_interface

Separate boards setup with other devices

Same as for the separate CPUs setup.

Multiple CE domains setup

Same as for the separate CPUs setup. Each domain's address must be unique, and the use of addressing in this configuration becomes important.

Linux container setup

The communication is based on Unix Domain Sockets (UDS) shared between the counterpart domains; this means that a common directory must be shared for each pair of communicating domains. This directory must be writable by at least one container, such that its gateway layer or adapter layer can create the named unix domain socket file and listen on it, and must be readable on the other container, which will connect to the shared named unix domain socket file. The dedicated shared directory for communication may support space limits for writing and inodes creation, for example: dedicated `tmpfs` mount or `btrfs` subvolume quota, to prevent denials of service due to filesystem space exhaustion.

The container manager is responsible for the actions below when each container is started or stopped:

- a shared storage space (a size-constrained `tmpfs` mount or `btrfs` subvolume) must be defined for each pair of containers on the host system, for instance `${IDC_HOST_DIR}/automotive-connectivity` for the link connecting the `automotive` and `connectivity` domains
- the shared storage must be mounted by the container manager with read/write permissions on the first domain of the pair, for instance as `${IDC_DIR}/connectivity` in the `automotive` domain
- the same shared storage must be mounted by the container manager with read permissions on the second domain of the pair, for instance as `${IDC_DIR}/automotive` in the `connectivity` domain
- when the container is stopped, the shared storage and mounts associated with the container must be unmounted

The variables `${IDC_HOST_DIR}` and `${IDC_DIR}` mentioned above represent the paths where the shared spaces are mapped on the host and containers filesystems respectively. By default, both variables `${IDC_HOST_DIR}` and `${IDC_DIR}` are defined in a common manner as `/var/lib/idc/`. OEM or developer's setup may require to redefine these paths for the customised environment.

Addressing and peer discovery

Network addressing and peer discovery

Each domain will be identified by its IPv6 address, and domains will be discovered using the IPv6 protocol's secure neighbour discovery[20] protocol. As domains do not need to be human-addressable (indeed, the users of the vehicle need never know that it has multiple domains running in it), there is no need to use DNS or mDNS for addressing.

The neighbour discovery protocol includes a feature called neighbour unreachability detection, which should be used as one method of determining that one of the domains has crashed. When a domain crashes, the other domain should poll for its existence on the network at a constant frequency (for example, at 2Hz) until it reappears at the same address as before. This frequency of polling is a trade-off between not flooding the network with connectivity checks, but also detecting reappearance of the domain rapidly.

When reconnecting to a restarted domain, the normal authentication process should be followed, as if both domains were starting up normally. There is no state to restore for the inter-domain link itself but, for example, SDK services may wish to re-query the automotive domain for the current vehicle state after reconnecting. They should do this after receiving an error response from the AD for an inter-domain communication which indicated that the other domain had crashed. Such behaviour is up to the implementers of each SDK service, and is not specified in this design.

Container-based addressing and peer discovery

Each container must be assigned an unique name on the filesystem to be used as domain identifier for addressing and peer discovery purposes.

The `${IDC_DIR}` directory in the container contains a directory entry for each associated domain to be connected through the inter-domain communication mechanism. As described in Linux container setup, the container manager is responsible for mounting a dedicated shared space to host the socket for the container pairs.

The name of mount point for the shared directory in the container should be the same as the name of counterpart peer. For example, to connect an `automotive` and a `connectivity` domain, the shared space must be mounted in the `automotive` container on the `${IDC_DIR}/connectivity/` path and must be mounted in the `connectivity` container on the `${IDC_DIR}/automotive/` path.

On startup, each container in the pair must try to `unlink()` any stale file in the shared spaces and then create a Unix Domain Socket named `socket` there. Since the shared directory is mounted with write permissions only on a single domain, the `unlink()` and `bind()` calls on the unix socket file will fail on the other domain, which only has read permissions.

Once it has removed any stale file and successfully created the socket, the first container in the pair must then `listen()` on it: for instance the `automotive`

[20] https://en.wikipedia.org/wiki/Secure_Neighbor_Discovery

domain must listen on the `${IDC_DIR}/connectivity/socket` unix socket. The second container in the pair must instead wait for the `socket` file to be available and must connect to it as soon it is created: for instance the `connectivity` must wait for the `${IDC_DIR}/automotive/socket` file to appear and connect to it.

Encryption

The confidentiality, integrity and authentication of the inter-domain communications link is provided by IPsec in transport mode for networked setups, and by kernel-provided Unix Domain Sockets on [container-based setups][Linux container setup].

Open question: What more detailed configuration options can we specify for setting up IPsec? For example, disabling various optional features which are not needed, to reduce the attack surface. What IKE service should be used?

The system should use an IPsec security policy which drops traffic between the CE and AD unless IPsec is in use. The security policy should not specify behaviour for communications with other peers.

Each domain must have an X.509 certificate (essentially, a public and private key pair), which are used for automatic keying for the IPsec connections. The certificates installed in the automotive domain must be signed by a certificate authority (CA) specific to the automotive domain and possibly the OEM. The certificates installed in the CE domain must be signed by a CA specific to the CE domain and possibly the OEM.

A domain (automotive or CE) which is in developer mode must use a certificate which is signed by a developer mode CA, not the production mode CA. This allows a production mode domain to prevent connections from a developer mode domain.

See Appendix: Software versus hardware encryption for a comparison of software and hardware encryption.

In order to maintain confidentiality of the connection, the keys for the IPsec connection must be kept confidential, which means they must be stored in memory which is not accessible to an attacker who has physical access to the system (see Tamper evidence and hardware encryption); or they must be encrypted under a key which is stored confidentially (a key-encrypting key, KEK). Such a confidential key store should be provided by the Secure Boot design — if available, confidentiality of the inter-domain communications can be guaranteed. If not available, inter-domain communications will not be confidential if an attacker can extract the boot keys for the system and use them to extract the inter-domain communications keys.

As of February 2016, the Secure Boot design is still forthcoming

See section 8.15 for further discussion of the hardware base for confidentiality and integrity of the system.

Open question: A lot of business logic for control over OEM licencing can be implemented by the choice of the CA hierarchy used by the inter-domain communication system. What business logic should be possible to implement?

Open question: Consider key control, revocation, protocol obsolescence, and various extensions for pinning keys and protocols.

Open question: What can be done in the automotive domain to reduce the possibility of exploits like Heartbleed[21] affecting the inter-domain communications link? This is a trade-off between the stability of AD updates (high; rarely released) and the pace of IPsec and TLS security research and updates and the need for crypto-agility (fast). Heartbleed was a bug in a bad implementation of an optional and not-very-useful TLS extension.

Control protocol

The control protocol provides push and pull method call semantics and a type system for marshalling method call parameters and return values — but it does not prescribe a specific set of APIs which it will transport. It must be flexible in the set of APIs which it transports.

We suggest using D-Bus over TCP as the control protocol, using a private bus between the automotive domain and the consumer–electronics domain. For multiple CE domain configurations, each automotive—consumer–electronics domain pair would have its own private bus.

The transport should be implemented using D-Bus' TCP socket transport[22] mechanism. Authentication, confidentiality and integrity are provided by the underlying IPsec connection. D-Bus implements its own datagram framing on top of the TCP stream.

On this bus, APIs from the automotive domain would be exposed as services; the CE domain can then call methods on those services, or receive signals from them.

D-Bus was chosen as it implements the necessary functionality, reuses a lot of the technologies already in use in Apertis, is stable, and is familiar to Apertis developers. Note that we suggest D-Bus the *protocol*, not necessarily dbus-daemon the *message bus daemon* or libdbus the reference *protocol library*. D-Bus the protocol provides:

- Method calls (pull semantics) with exactly one reply, supporting timeouts
- Error responses
- Signals (push semantics)
- Properties

[21] https://en.wikipedia.org/wiki/Heartbleed
[22] http://dbus.freedesktop.org/doc/dbus-specification.html#transports-tcp-sockets

- Strong type system
- Introspection

There are several important points here: introspection means that the D-Bus services on the AD can send their API definitions to the CE at runtime if needed, so that the CE does not have to have access to header files (or similar) from the AD. It also means the API definition can change without needing to recompile things — for example, an update to the AD could expose new APIs to the CE without needing to update header files on the CE. Finally, method calls support 'in' and 'out' parameters (multiple return values) which allows for bi-directional communication in the control protocol.

Open question: How should the multiple CE configuration (Configuration designs interact with D-Bus signals? Can the adapter layer perform the broadcast to all subscribers?

The D-Bus protocol is stable, and has maintained backwards compatibility with all previous versions since 2006[23]. If changes to the D-Bus protocol are introduced in future, they will be introduced as extensions which are used optionally, if supported by both peers on the bus. Hence backwards compatibility is maintained.

Data connections

If a service wishes to send high-bandwidth data between the domains, it must open a new data connection. Data connections are created on demand, and are subject to traffic control, so the AD may, for example, reject a connection request or throttle its bandwidth in order to maintain quality of service for existing connections.

The inter-domain communication protocol provides two types of data connection: TCP-like and UDP-like. These are implemented as TCP or UDP connections between the two domains, running over IPsec. IPsec provides the necessary authentication, confidentiality and integrity of the data; TCP or UDP provide the multiplexing between connections (see the IPsec protocol stacks figure in IPSec versus TLS). For Linux container setup a Unix domain socket is used as the IDC link; the local kernel provides the needed authentication, confidentiality and integrity of the data. Services must implement their own application-specific protocols on top of the TCP or UDP connection they are provided. For example, a video service may use a lossy synchronised audio/video protocol over UDP for sending video data together with synchronised audio; while a download service may use HTTP over TCP for sending downloads between domains. (See [here][Appendix: Audio and video decoding] for a discussion of options for implementing video and audio decoding.) Such protocols are not defined as part of this design — they are the responsibility of the services themselves to design and implement.

[23] http://dbus.freedesktop.org/doc/dbus-specification.html#stability

Data connections are opened by sending a request to one of the inter-domain services (Protocol library and inter-domain services), specifying desired characteristics for the connection, such as whether it should be TCP-like or UDP-like, its bandwidth and latency requirements, etc. The connection will be opened and a unique identifier and file descriptor for it returned to the requesting service. This service must then send the identifier over the control connection so that the corresponding service in the other domain can request a file descriptor for the other end of the connection from its inter-domain service.

Open question: Could this be simplified by using D-Bus' support for file descriptor passing? D-Bus' TCP transport currently explicitly does not support file descriptor passing, so implementing it that way without introducing incompatibilities requires planning.

It is tempting to extend D-Bus' support for file descriptor (FD) passing so that it operates over TCP to provide these data connections. However, that would effectively be a fork of the D-Bus protocol, which we do not want to maintain as part of this system. Secondly, due to the way FD passing works, with the peer passing an FD to the dbus-daemon and asking for it to be forwarded — this would mean that the peer (i.e. an SDK or OEM service) has the responsibility for opening the data connection within the IPsec tunnel, which would be very complex.

Instead, we recommend a custom API provided by the inter-domain service which an SDK or OEM service can call to open a new data connection, passing in the parameters for the connection (such as TCP/UDP, quality of service requirements, etc.). The inter-domain service would communicate over a private control API with the other inter-domain service to open and authenticate the connection at both ends, and return a file descriptor and cryptographic nonce (securely random value at least 256 bits long) to the original SDK or OEM service. This service can use that file descriptor as the data connection, and should pass the nonce over its own control protocol to the corresponding OEM or SDK service. This service should then pass the nonce to its inter-domain service and will receive the file descriptor for the other end of the data connection in reply.

Both inter-domain services should retain their file descriptors (which they have shared with the OEM and SDK services) for the data connection, so that if the kill switch (Disabling the CE domain) is enabled, they can call shutdown() on the data connection to forcibly close it.

The inter-domain services must reserve all well-known names starting with `org.apertis.InterDomain` (for example, `org.apertis.InterDomain1` or `org.apertis.InterDomain1.DataConnections`), and similarly all D-Bus interface names. This means they must not allow these names to be used as part of the OEM API shared between the export and adapter layers (Interaction of the export and adapter layers).

A data connection cannot exist without an associated control connection (though one control connection may be associated with many data connections). As data connections are opened and controlled through APIs defined on the inter-domain services, there is no need for standard network-style service discovery using protocols like DNS-SD[24] or SSDP[25].

Time synchronization

As a distributed system, the inter-domain services may require a shared clock across the domains. Time synchronization is critical to correlate events and this is specially important when playing audio and video streams, for example. If those streams are decoded on the CE and needs to played by the AD, the AD and the CE should agree on the meaning of the timestamps embedded in the streams.

For the synchronization, there are two suitable protocols:

- NTP[26] is a well-known protocol to synchronise time among remote systems. It provides millisecond or sub-millisecond accuracy over the Internet or local area networks respectively;
- PTP[27] provides microsecond or sub-microsecond accuracy and is designed for local area networks.

In terms of latency calculation, both protocols satisfy the requirements, but we recommends PTP for the following reasons:

- NTP uses hierarchical time sources, whereas PTP has a simpler master/slave model. That means any system that is even untrusted domain in a network is able to be taken by the other CE domain as a NTP source;
- PTP supports hardware assisted timestamps to improve accuracy. Under Linux, the PTP hardware clock (PHC) subsystem is used to produce timestamps on supported network devices.

Audio streams

To share audio streams RTP[28] and its companion protocol RTCP[29] are recommended both on networked and container-based setups, for encoded and decoded streams.

They provide jitter compensation, out-of-sequence handling and synchronization across multiple different streams.

[24] https://en.wikipedia.org/wiki/Zero-configuration_networking#DNS-SD
[25] https://en.wikipedia.org/wiki/Simple_Service_Discovery_Protocol
[26] https://en.wikipedia.org/wiki/Network_Time_Protocol
[27] https://en.wikipedia.org/wiki/Precision_Time_Protocol
[28] https://en.wikipedia.org/wiki/Real-time_Transport_Protocol
[29] https://en.wikipedia.org/wiki/RTP_Control_Protocol

In particular [multiplexed RTP/RCTP][Appendix: Multiplexing RTP and RTCP] can be used to multiplex both protocols over the kind of data connections described above.

Decoded video streams

A fully decoded video stream consumes large quantities of bandwidth and sharing it between domains using the same approach used by audio (RTP) can only work for very small resolutions (see Memory bandwith usage on the i.MX6 Sabrelite for the bandwidth limitations on one of the platforms targeted by Apertis).

If a domain sends uncompressed 1080p video stream at 25fps in YUV422 format to another domain it requires just a bit more than 100MB/s for just the stream transfer. This already makes it prohibitive on Gigabit Ethernet systems, which have a theoretical maximum bandwith of 125MB/s, without including any framing overhead. Even for local transfers this is a significant portion of the total memory bandwidth, even more so if taking in account other activities including the actual decoding and playback, plus the need for the same memory bandwidth toward the GPU where the decoded frames need to be composed.

To be able to handle 1080p video streams it is very important that zero-copy mechanisms are used for the transfer of frames, see Appendix: Audio and video decoding for further considerations about how a protocol can be defined to match such expectations.

Bulk data transfers

Data connections are suitable for transfers that involve large amounts of static contents such as firmware images.

To avoid storing multiple copies of the same data on the limited local storage, for instance in cases where the contents are downloaded from the Internet from a lower-privilege domain before being handed over to a more isolated higher-privilege domain, validation of the data such as checksum verification should be done on the fly by the originator, and only the recipient should store the data on its local storage.

Raw direct TCP connections over IPSec or raw UDP sockets can be suitable for the inter-domain data transfer, as they both provide reliability, integrity and confidentiality. The downside of this approach is that each application would need to handle data validation and resumable transfers on its own: for this reason it is preferable to handle basic data validation in the inter-domain communication layers and provide the data to the receiver only once it is complete and matches the specified cryptographic hashes.

The basic API thus is aimed at senders downloading large contents from the Internet and directly streaming across the domains without storing them locally, doing on-the-fly cryptographic validation of the streamed data. The contents are received and re-validated on the destination domain, where they are stored in a file which is passed to the destination service once the transfer is complete and valid.

When the destination service has received the file handle it must perform any additional verification of the contents. It can also link the anonymous file descriptor to a locally-accessible file path using the `linkat()`[30] syscall with the `AT_EMPTY_PATH` flag or use the `copy_file_range()`[31] syscall to get a copy of the contents in the most efficient way that the kernel can provide.

A different mechanism can be defined where the sender stores the contents in a private file and passes a file descriptor pointing to it to the inter-domain communication subsystem. The receiving side then uses the `copy_file_range()` syscall to get a copy of the data that cannot be altered by the sender and then validates the data. On filesystems that supports reflinks, `copy_file_range()` will automatically use them to provide fast copy-on-write clones of the original file: this would make the operation nearly-instantaneous regardless of the amount of data, and would avoid doubling the storage requirements. When reflinks cannot be used, `copy_file_range()` will do an in-kernel copy, avoiding unnecessary context-switches over normal user-space copy operations. Such approach can be used on container-based setups or when a cluster file system is shared across networked domains. Not many filesystems can handle reflinks, but Btrfs and the OCFS2 cluster filesystem support them.

On systems set up such that reflinks can be used, this solution is much more efficient than the alternatives, but imposes constraints on the whole system that may not be acceptable, such as requiring filesystems that support reflinks (such as Btrfs or OCFS2) on all the domains and ensuring that the appropriate shared filesystem mounts are available to SDK services. For this reason, the socket-based approach is recommended in the general case.

Data connections API

This section defines the draft for a proposed D-Bus API that SDK services could use to request the creation of data channels separated from the control plane connection.

The gateway and adapter layers are responsible for the creation and initialization of those channels, while other services and applications must not be able to directly create them.

The gateway and adapter layers use instead file descriptors passing to share the channel endpoints with the requesting services and applications.

The API drafted here is meant to only provide a very rough guideline for those implementing any real data channel API and it's not meant to be normative: real implementations can diverge from the interfaces described here and the

[30] https://manpages.debian.org/stretch/manpages-dev/link.2.en.html
[31] https://manpages.debian.org/stretch/manpages-dev/copy_file_range.2.en.html

actual API to be used by SDK services must be documented in a separate specification.

```
/* The interface exported by the adapter/gateway to SDK services to initiate channel creation. */
interface org.apertis.InterDomain.DataConnection1 {
  /* @id: the app-specific unique token used to to identify and authorize the channel
   * @destination: the bus name of the service which should be at the other end of the channel
   * @type: the kind of data and the protocol to be used for the data exchange.
   *        Use 'audio-rtp' for multiplexed RTP/RFC5761.
   * @metadata_in: a dictionary of extra information that can be used to authorize/validate the transfer
   * @metadata_out: the @metadata_in dictionary with additional information
   * @fd: the file descriptor for the actual data exchange using the protocol specified by @type */
  method CreateChannel (in s id,
                        in s destination,
                        in s type,
                        in a{sv} metadata_in,
                        out a{sv} metadata_out,
                        out h fd)

  /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
   *
   * If the receiver was not able to validate the channel, the `org.apertis.InterDomain.ChannelError`
   * error is raised.  */
  method CommitChannel(in s id)

  /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel() */
  method AbortChannel(in s id)

  /* @refclk: the reference to the IDC shared clock, in the format of defined
   * by the `clksrc` production of RFC7273 for the `ts-refclk:` parameter */
  method GetClockReference(out s refclk)
}

/* The interface to be exported by services that can handle incoming channels.
 * Domains that do not use a local dbus-daemon can implement a similar mechanism
 * with the native IPC system. */
interface org.apertis.InterDomain.DataConnectionClient1 {
  /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
   * @sender: the bus name of the service which initiated the channel creation
   * @type, @metadata_in, @metadata_out: see org.apertis.InterDomain.DataConnection1.CreateChannel()
   * @proceed: true if the channel should be set up, false if it should be refused */
  method ChannelRequested(in s id,
                          in s sender,
                          in s type,
                          in a{sv} metadata_in,
                          out a{sv} metadata_out,
```

```
                            out b proceed)

  /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
   * @success: whether the connection has been successfully set up and @fd is usable
   * @fd: the file descriptor from which to read the incoming data with the
          previously agreed protocol
  method ChannelCreated(in s id,
                        in b success,
                        in h fd)
}

/* The interface private to gateway/adapter services to cross the domain boundary. */
interface org.apertis.InterDomain.DataConnectionInternal1 {
  /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
   * @sender: see org.apertis.InterDomain.DataConnectionClient1.ChannelRequested()
   * @destination, @type, @metadata_in, @metadata_out: see org.apertis.InterDomain.DataConnection1.CreateChann
   * @proceed: see org.apertis.InterDomain.DataConnectionClient1.ChannelRequested()
   * @nonce: a one-time value used to authenticate the socket
   * @socket_addr: the proto:addr:port string to be used to connect to the remote service
  method RequestChannel(in s id,
                        in s sender,
                        in s destination,
                        in s type,
                        in a{sv} metadata_in,
                        out a{sv} metadata_out,
                        out b proceed,
                        out s nonce,
                        out s socket_addr)

  /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
   * @sender: see org.apertis.InterDomain.DataConnectionClient1.ChannelRequested()
   * @destination: see org.apertis.InterDomain.DataConnection1.CreateChannel()
   *
   * If the receiver was not able to validate the channel, the `org.apertis.InterDomain.ChannelError`
   * error is raised.  */
   */
  method CommitChannel(in s id,
                       in s sender,
                       in s destination)

  /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
   * @sender: see org.apertis.InterDomain.DataConnectionClient1.ChannelRequested()
   * @destination: see org.apertis.InterDomain.DataConnection1.CreateChannel()
   */
  method AbortChannel(in s id,
                      in s sender,
```

```
                      in s destination)
}
```

Data channel API flow example for a media player streaming audio

A possible use-case of the API is a Media Player frontend hosted on the AD with the backend on the CE. The frontend requests the backend to decode a specific stream using an application specific API and passing a token with the request.

```
                 AD                            |                       CE
media player           gateway          |       adapter            media player
 frontend                                |                              backend
    o ------ Play() ------------o------------|------------o------------------
----> o
                                         |          o <-- CreateChannel() -- o
                                o <-- RequestChannel() -- o
    o <-- ChannelRequested() -- o            |
    o -- ChannelRequested() --> o            |
               reply                         |
                                o -- RequestChannel() --> o
                                          reply
                                o <- connect and nonce -- o
   o <-- ChannelCreated() ---- o         |          o -- CreateChannel() --> o
                                             |                     reply
    o <------------------------------ data channel -------------------------
----- o
```

The Media Player frontend initially calls the application-specific `Play()` method on its backend, with the IDC system transparently proxying the request across domains. This call must also carry an application-specific token that will be used to identify the request during the channel creation procedure.

Once the Media Player backend has gathered some metadata about the stream to be played, it requests the creation of an `audio-rtp` channel directed to the Media Player frontend by calling the `org.apertis.InterDomain.DataConnection1.CreateChannel()` on the local adapter service.

The adapter service will then access the inter-domain link by calling the `org.apertis.InterDomain.DataConnectionInternal1.RequestChannel()` method of the remote gateway peer.

The gateway service on the AD notifies the Media Player frontend that a channel has been requested, passing the request token and other application-specific metadata. If the token matches and the metadata is acceptable, the Media Player frontend replies to the gateway service telling it to proceed.

Once the request has been accepted by the destination, the gateway service creates a listening socket for the requested channel type and returns the information needed to connect to it to the remote adapter peer, including a nonce to authenticate the connection.

As soon as the adapter gets the socket information it connects to it and sends the nonce over it. On the other side the gateway will read the nonce and if does not matches it immediately closes the connection.

Once the connection has been set up and the nonce has been successfully shared, the adapter and gateway services will hand over the file descriptors of the sockets that have been set up.

Data channel API flow example for an update manager sharing firmware images

The bulk data transfer API is meant to be useful for update managers where an agent in the Connectivity Domain fetches firmware images from the Internet and shares them with the update manager in the AD which has access to the devices to be updated.

```
                 AD                             |                      CD
update manager            gateway        |       adapter             OTA agent
    o ----> GetUpdate()  -------o------------|------------o------------------
----> o
                                       |           o <-- CreateChannel() -- o
                               o <-- RequestChannel() -- o
    o <-- ChannelRequested() -- o            |
    o -- ChannelRequested() --> o            |
               reply                         |
                               o -- RequestChannel() --> o
                                         reply
                               o <- connect and nonce -- o
                                       |           o -- CreateChannel() --> o
                                             |                     reply
                        o <-----data channel ---------------------------- o
                                       |           o <- CommitChannel() --- o
                               o <-- CommitChannel() --- o
   o <-- ChannelCreated() ---- o         |           o -- CommitChannel() --> o
                                             |                     reply
```

The update manager calls the `GetUpdate()` method of the agent, with a token identifying the request. The OTA agent retrieves the metadata of the file to be shared, in particular the size and a set of cryptographic hashes. With that information, it requests the creation of a `bulk-data` channel with the `org.apertis.InterDomain.DataConnection1.CreateChannel()` method of the local adapter service. The OTA agent must specify the `size` parameter and a known cryptographic hash such as `sha512` in the `metadata_in` parameter. It must then check in the `metadata_out` for the `offset` parameter to figure out if it must resume an interrupted download.

The adapter service accesses the inter-domain link by calling the `org.apertis.InterDomain.DataConnectionInternal` method of the remote gateway peer.

The flow is analogous to the one in the [streaming media player case][Data channel API flow example for a media player streaming audio] until the point where the inter-domain socket is created: while the receiving end of the socket in the streaming case is meant to be used by the receiving service, in the bulk data case it is used directly by the gateway, which stores the received data in a local file.

While it sends data through the socket, the OTA agent is expected to perform on-the-fly data validation by computing cryptographic hashes on the streamed contents: once it has sent all the data the agent can close the socket and call `org.apertis.InterDomain.DataConnectionInternal1.CommitChannel()` to signal that all the data has been shared successfully and that the computed hashes match, or `AbortChannel()` otherwise.

Upon receiving the `CommitChannel()` message, the gateway checks that the file size and cryptographic hashes match the expected values and raises the `ChannelError` error otherwise. If and only if the data is valid it instead shares the file descriptor pointing to the file to the OTA updater with a `ChannelCreated()` call.

Traffic control

Traffic control[32] should be set by the inter-domain service (Protocol library and inter-domain services) in the CE domain, using the standard Linux traffic control functionality in the kernel[33]. As the control connection and each data connection are separate TCP or UDP connections, they can have traffic controls applied to them individually, which allows different quality of service settings for individual data connections; and allows the control connection to have a higher quality of service than all data connections, to help ensure it has guaranteed low latency.

Applying traffic control in the CE domain has the advantage of knowing what kernel functionality is available — if it were applied in the automotive domain, its functionality would be limited by whatever is provided by the automotive OS (for example, QNX). It has the disadvantage, however, of being vulnerable to the CE domain being compromised: if an attacker gains control of the inter-domain service in the CE domain, they can disable traffic control. However, if they have gained control of that service, the only remaining mitigation is for the automotive domain to shut down the CE domain, so having control over traffic policy has little effect.

[32] https://en.wikipedia.org/wiki/Network_traffic_control
[33] http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html

The specific traffic control policies used by the inter-domain service can be determined later, based on the relative priorities an OEM assigns to different types of traffic.

Protocol library and inter-domain services

The inter-domain communications protocol should be implemented as a library, containing all layers of the protocol. The particular domain configuration which the library targets should be a configure-time option, though the library must support enabling the Standalone setup transport in conjunction with another transport, when in developer mode (see Mock SDK implementation).

By implementing the protocol as a library, it can be tested easily by being linked into unit tests — rather than trying to wrap the entire inter-domain service daemon in a test harness. Internally, the library should implement all protocol layers separately and expose them to the unit tests so that they can be tested individually.

Furthermore, this allows the protocol code to be reused between the inter-domain service in the automotive domain, and the inter-domain service in the CE domain.

The main advantage of implementing the protocol as a library is the flexibility this provides for integrating it into different automotive domain implementations — it can be integrated into an existing system service (bearing in mind the suggestion to keep it in a separate trust domain, Security domains), or could be used as a stand-alone service daemon.

A reference implementation of such a stand-alone inter-domain service program should be provided with the protocol library. This should provide the necessary systemd service file and AppArmor profile to allow itself to be strictly confined if the automotive domain OS supports this.

As the inter-domain communications protocol uses D-Bus, the protocol library must contain an implementation of the D-Bus protocol. Note that this is *not* a D-Bus daemon; it is a D-Bus library, like libdbus or GDBus. See Appendix: D-Bus components and licensing for details about the different components in D-Bus and their licensing.

Apart from its D-Bus library dependency, the protocol library should be designed with minimal dependencies in order to be easily integratable into a variety of automotive domain operating systems (from Linux through to other Unixes, QNX or Autosar). If the chosen D-Bus library is available as part of the automotive OS (which is more likely for libdbus than for other D-Bus libraries), it could be linked against; otherwise, it could be statically linked into the protocol library.

libdbus itself is already quite portable, having been known to work on Linux, Windows, OS X, NetBSD and QNX. It should not be difficult to port to other

POSIX-compliant operating systems.

Rate limiting on control messages should be implemented in the protocol library, so that the same functionality is present in both the automotive and CE domains.

The protocol library should expose the encryption keys for the IPsec connection used in the inter-domain communications link, including signals for when those keys change (due to cookie renegotiation on the link). The keys must only be exposed in development builds of the protocol library. See Debuggability for more details.

Non Linux-based domains

The suggested implementation uses D-Bus the *protocol*, not necessarily dbus-daemon the *message bus daemon* or libdbus the *protocol library*.

This means that for inter-domain communications purposes, only the serialization format of D-Bus is used as a well defined RPC protocol. There's no requirement that domains run `dbus-daemon` or that they use a specific D-Bus implementation to talk to other domains.

Several implementations of the D-Bus serialization format exists and their use is strongly encouraged rather than reimplementing the protocol from scratch:

- GDBus[34] is a GTK+/GNOME oriented implementation of the D-Bus protocol in GLib
- QtDBus[35] is Qt module that implements the D-Bus protocol
- node-dbus[36] is a D-Bus protocol implementation for NodeJS written in pure JavaScript
- libdbus[37] is the reference implementation of the D-Bus protocol
- dbus-sharp[38] is a C#/.net/Mono implementation of the D-Bus protocol
- pydbus[39] is a python implementation of the D-Bus protocol

On networked setups the D-Bus-based protocol is transported over TCP, relying on IPSec for authentication, confidentiality and reliability.

If IPSec nor TLS are available, those properties cannot be guaranteed, and thus such setup is strongly discouraged. In that case every input should be treated as potentially malicious: the trusted domains must export only a very reduced

[34] https://developer.gnome.org/gio/stable/gdbus.html
[35] http://doc.qt.io/qt-5/qtdbus-index.html
[36] https://github.com/sidorares/node-dbus
[37] https://dbus.freedesktop.org/doc/api/html/
[38] https://github.com/mono/dbus-sharp
[39] https://github.com/LEW21/pydbus

set of interfaces, which must be conceived in a way that any kind of misuse does not lead to harm.

Service discovery

Accordingly to the use of the D-Bus serialization protocol, each service exported over the inter-domain communication channels is identified by a well-known name subject specific constraints[40], starting with the reversed DNS domain name of the author of the service (for instance, `com.collabora.CarOS.ClimateControl1` for a potential service written by Collabora[41].

Only one service at a time can own such names on each domain, but the ownership is not tracked across domains and collision may happen due to a transitional state during an upgrade or other causes: each setup is thus responsible to define a deterministic collision resolution procedure should two domains export the same service name.

The adapter layer is responsible to inspect on which channel each service is available. The `NameOwnerChanged signal`[42] must be used by the adapter layer to track the availability of services on each connection and to detect when a service is no longer available or changed ownership (for example because it has been restarted). The `org.freedesktop.DBus.ListActivatableNames()`[43] message can be used to gather the initial list of available services.

After an upgrade a domain may stop providing a specific service and another domain may start providing it instead: both the old and new domains must trigger the `NameOwnerChanged signal`[44] in response to the `org.freedesktop.DBus.ReleaseName()`[45] and `org.freedesktop.DBus.RequestName()`[46] calls. No specific ordering is required and thus the service may be temporarily unavailable or the two domains may export the same service name at the same time: the collision resolution procedure must choose the one on the connection with the highest priority.

In the simplest case, each domain must be given an unique priority with the AD having the highest priority. The relative priority between the CE domains is used to provide deterministic service access when a service name exists on multiple connections. As a result, the priority list must be static and the priority of CE domains can be assigned arbitrarily for each specific setup.

[40] https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-bus
[41] https://collabora.com
[42] https://dbus.freedesktop.org/doc/dbus-specification.html#bus-messages-name-owner-changed
[43] https://dbus.freedesktop.org/doc/dbus-specification.html#bus-messages-list-activatable-names
[44] https://dbus.freedesktop.org/doc/dbus-specification.html#bus-messages-name-owner-changed
[45] https://dbus.freedesktop.org/doc/dbus-specification.html#bus-messages-release-name
[46] https://dbus.freedesktop.org/doc/dbus-specification.html#bus-messages-request-name

When accessing a service name that exists on more than one connection, the service that exists on the connection with the highest priority must be given precedence by the adapter layer.

CE domains should not be able to spoof trusted services exported by the AD: for this reason a static list of services meant to be exported only by the AD must be defined and the adapter layer must ignore matching services exported by other connections, even if the service is not currently available on the AD connection itself.

Particular care must be taken to ensure each domain can be fully booted without blocking on services hosted on other domains, to avoid untracked circular dependencies.

SDK services must access the above service names through the private bus instance exported by the adapter layer, which proxies them from all the inter-domain channels, abstracting the complexities of inter-domain communications. SDK services are not aware of the fact that the services are hosted on different domains.

Automotive domain export layer

To integrate the inter-domain communications system into an automotive domain operating system, the APIs to be shared must be exported as objects on the D-Bus connection provided by the inter-domain service. This is done as an *export layer* in the inter-domain service in the automotive domain, customised for the OEM and their specific APIs. The export layer could be implemented as pure C calls from within the same process (no protocol at all), or D-Bus, or kdbus, or QNX message passing, or something else entirely. If D-Bus bus is used, a D-Bus daemon would need to be running on the automotive domain; otherwise, no D-Bus daemon would be needed.

For example, if the automotive domain provides the APIs which are to be exposed over the inter-domain connection as:

- C APIs in headers — the inter-domain service would call those APIs directly, and the export layer would essentially be those C calls;
- daemons with UNIX socket connections — the inter-domain service would connect to those sockets and run whatever protocol is specified by the daemons, and the export layer would essentially be the socket connections and protocol implementations;
- D-Bus services — the inter-domain service would connect to a D-Bus daemon on the automotive domain and translate the services' D-Bus APIs into an API to expose on the inter-domain communications link (see below), and the export layer would be the D-Bus daemon, D-Bus library in the inter-domain service, and the code to translate between the two D-Bus APIs.

The APIs must be exported under well-known names[47] formatted as reverse-DNS names owned by the OEM. For example, if the AD operating system was written by Collabora, APIs would be exported using well-known names starting with com.collabora, such as com.collabora.CarOS.EngineManagement1 or com.collabora.CarOS.ClimateControl1.

The API formed by these exported D-Bus objects is vendor-specific, but should maintain its own stability guarantees — for every backwards-incompatible change to this API, there must be a corresponding update to the CE domain to handle it. Consequently, we recommend versioning the exported D-Bus APIs[48].

APIs which the OEM does not want to make available on the inter-domain communications link (for example, because they are not able to handle untrusted data, or are too powerful to expose) must not be exported onto the D-Bus connection. This effectively forms a whitelist of exposed services.

For each piece of functionality exposed by the AD, suitable safety limits must be applied (Safety limits on AD APIs). If the implementation of that functionality already applies the safety limits, nothing more needs to be done. Otherwise, the safety limits must be enforced in the interface code which exports that functionality onto the inter-domain D-Bus connection.

Similarly, for each piece of functionality exposed by the AD, if it fails to respond to a call by the inter-domain service, the service must return an error to the CE over the inter-domain D-Bus connection, rather than timing out. This is especially important in systems where the export layer is a set of C calls — the implementation must take care to ensure those calls cannot block the inter-domain service.

If the vendor wants to implement per-API kill switches for services exported by the automotive domain, these must be implemented in the export layer (see Disabling the CE domain).

Consumer-electronics domain adapter layer

Paired with the OEM-specific API export code in the automotive domain is an *adapter layer* in the CE domain. This adapts the API exported by the services on the automotive domain to the stable SDK APIs used by applications in the CE domain. The layer has an implementation in each of the SDK services in the CE domain.

This adapter layer does not have a trust boundary — each part of it lies within the trust domain of the relevant SDK service.

These adapters connect to a private D-Bus bus, which the inter-domain service in the CE domain is also connected to. The inter-domain service exports the

[47] http://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-bus
[48] http://dbus.freedesktop.org/doc/dbus-api-design.html#api-versioning

OEM APIs from the automotive domain on this bus, and the adapters consume them.

The private bus could be implemented either by running dbus-daemon with a custom bus configuration, or by implementing it directly in the inter-domain service, and having all adapters connect directly to the service. In both cases, the trust boundary between the adapters (within the trust domains of the SDK services) and the inter-domain service are enforced.

Interaction of the export and adapter layers

The interaction between the export and adapter layers is important in maintaining compatibility between different versions of the AD and CE as they are upgraded separately. The CE is typically upgraded much more frequently than the AD. Both are customised to the OEM.

Initial deployment

The OEM develops both layers, and stabilises an initial version of their inter-domain API, using a version number (for example, 1). The export layer exports objects from the automotive domain, and the adapter layer imports those same objects. There may be functionality exposed on the objects which the SDK APIs currently do not support — in which case, the adapter layer ignores that functionality.

CE is upgraded, AD remains unchanged

A new release of Apertis is made, which expands the SDK APIs to support more functionality. The OEM integrates this release of Apertis and updates their adapter layer to tie the new SDK APIs to previously-unused objects from the inter-domain link.

The version number of the inter-domain API remains at 1.

AD is upgraded, CE remains unchanged

The automotive domain OS is upgraded, and more vehicle functionality becomes available to expose on the inter-domain connection. The OEM chooses to expose most of this functionality using the inter-domain service. For some objects, this results in no API changes. For other objects, it results in new methods being added, but no old ones are changed. For some objects, it results in some old methods being removed or their semantics changed. For these objects, the OEM now exports *two* interfaces on the inter-domain service: one at version 1, exporting the old API; and one at version 2, exporting the new API. The version number of other inter-domain APIs remains at 1.

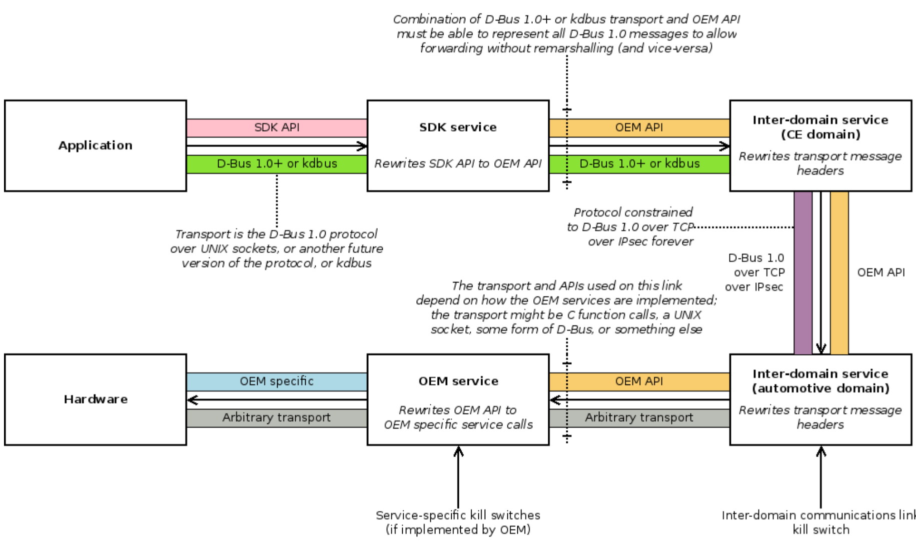
The CE domain software remains unchanged, which means it continues to use the version 1 APIs. This continues to work because all objects on the inter-domain API continue to export version 1 APIs (in addition to some version 2 APIs).

CE is upgraded again

The next time the CE domain is upgraded, its adapter layer can be modified by the OEM to use the new version 2 APIs for some of the services. If this updated version of the CE domain is guaranteed to only be used with new versions of the AD, the adapter layer can drop support for version 1 APIs. If the updated CE domain may be used with old versions of the AD, it must support version 1 and version 2 (or just version 1) APIs, and use whichever it prefers.

Flow for a given SDK API call

In the following figure, particular attention should be paid to the restrictions on the protocols in use for each link. For the links between the application and the inter-domain service in the CE domain, any version of the D-Bus protocol can be used, including kdbus or another future version. This depends only on the dbus-daemon and D-Bus libraries available in the CE domain. For the link between the two inter-domain services, the protocol must always be at least D-Bus 1.0 over TCP over IPsec. If both peers support a later version of the protocol, they may use it — but both must always support D-Bus 1.0 over TCP over IPsec. For the link between the inter-domain service in the automotive domain and the OEM service, whatever protocol the OEM finds most appropriate for implementing their export layer should be used. This could be pure C calls from within the same process (no protocol at all), or D-Bus, or kdbus, or QNX message passing, or something else entirely.



Apertis IDC message flow, following a message being sent from application to hardware; the message flow is the same in reverse for message replies from the hardware

Trusted path to the AD

Providing a trusted input and output path between the user and the automotive domain is out of scope for this design — it is a problem to be solved by the graphics sharing and input handling designs. However, it is worth noting that the solution must not involve communication (unauthenticated, or authenticated via the CE domain) over the inter-domain link. If it did, a compromised CE domain could be used to forge this communication and gain control of the trusted path to the AD — which likely results in a large privilege escalation.

A trusted path should be implemented by direct communication between the input and output devices and the automotive domain, or mediating such communication through the hypervisor, which is trusted.

Developer mode

In order to support connecting the CE domain from an SDK on a developer's laptop to the automotive domain in a development vehicle, the 'separate boards setup with other devices' configuration must be used, with the CE domain and the automotive domain connected to the developer's network (which might have other devices on it).

In order to allow the SDK to connect, the vehicle must be in a 'developer mode'. This is because the CE domain is entirely untrusted when it is provided by the SDK, because the developer may choose to disable security features in it (indeed, they may be working on those security features).

Open question: What cryptography should be used to implement this authentication, and the division of trust between development and production devices? A likely solution is to only have the AD accept the CE connection if it connects with a 'production' key signed by the vehicle OEM.

Mock SDK implementation

In order to allow applications to be developed against the Apertis SDK, implementations of all the SDK APIs need to be provided as part of the official SDK virtual machine distribution. These implementations need to be fully featured, otherwise application developers cannot develop against the unimplemented features.

There are two implementation options:

1. Have an Apertis SDK adapter layer which provides the mock implementations, and which does not use an inter-domain service or mock up any of the automotive domain.
2. Write the mock implementations as stand-alone services which are logically part of the automotive domain (even though there is no domain

separation in the SDK). Expose these services on the inter-domain link using an Apertis SDK export layer; and adapt the services to the actual SDK APIs using an Apertis SDK adapter layer.
The inter-domain services would be running in the same domain (the SDK) and would communicate over a loopback TCP socket (see Standalone setup).

Option #1 has a much simpler implementation, but option #2 means that the inter-domain communications code paths are tested by all application developers. Similarly, option #1 introduces the possibility for behavioural differences between the mock adapter layer and the production inter-domain communication system, which could affect how application developers write their applications; option #2 reduces the potential for that considerably.

As option #2 uses the inter-domain service in the CE domain, it also allows for the possibility of connecting the CE domain to a different automotive domain — rather than the mock one provided by the SDK, a developer could connect to the automotive domain in a development vehicle (Developer mode).

Hence, our recommendation is for option #2.

Debuggability

The debuggability of the inter-domain communications link is important for many reasons, from integrating two domains to bringing up a new automotive domain (with its export and adapter layers) to developing a new SDK API.

Referring to the figure in Overall architecture, debugging of:

- *applications and the SDK services* happens using normal tools and methods described in the Debug and Logging design[49];
- *communications between the dbus-daemon (private bus) and inter-domain service (CE domain)* happens using normal D-Bus monitoring tools (such as Bustle[50] or dbus-monitor[51]), though this requires the developer to gain access to the private bus' socket;
- *communications between the inter-domain services* happens using a special debug option in the services (see below);
- *the export layer and OEM services* happens using tools and methods specific to how the OEM has implemented the export layer.

If possible, all debugging should happen on the SDK side, in the adapter layer or above, as this allows the greatest flexibility in debugging techniques — none of the communications at that level are encrypted, so are accessible to a developer user with the appropriate elevated permissions.

[49] https://em.pages.apertis.org/apertis-website/concepts/debug-and-logging/
[50] http://willthompson.co.uk/bustle/
[51] http://dbus.freedesktop.org/doc/dbus-monitor.1.html

If the connection between the inter-domain services (the TCP/IPsec link between domains) needs to be debugged, it can be complex, as any debugging tool needs to be able to decrypt the IPsec encryption. Wireshark is able to do this[52], if given the encryption key in use by the IPsec connection. This key may change over the lifetime of a connection (as the connection cookie is refreshed), and hence needs to be exported dynamically by the inter-domain service. In order to allow debugging both ends of the connection, it should be implemented in the protocol library (Protocol library and inter-domain services). In the CE domain, it should be exposed as a D-Bus interface on the private bus which is part of the adapter layer. This limits its access to developers who have access to that bus.

```
Interface org.apertis.InterDomainConnection.Debug1 {
  /* Mapping from IKEv1 initiator cookie to encryption key. */
  readonly property a{ss} Ike1Keys;
  /* Mapping from IKEv2 tuple of (initiator SPI, responder SPI) to tuple
   * of (SK_ei, SK_er, encryption algorithm, SK_ai, SK_ar, integrity
   * algorithm). Algorithms are enumerated types, with values to be
   * documented by the implementation. Other parameters are provided as
   * hexadecimal strings to allow for varying key lengths. */
  readonly property a((ss)(ssssussu)) Ike2Keys;
}
```

A new Lua plugin[53] in Wireshark could connect to this interface and listen for signals of updates to the connection's keys, and use those to update Wireshark's IKE decryption table. Wireshark is the suggested debugging tool to use, as it is a mature network analysis tool which is well suited to analysing the protocols being sent over the inter-domain connection.

In the automotive domain, the key information provided by the protocol library should be exposed in a manner which best fits the debugging infrastructure and tools available for the automotive operating system.

In both domains, this interface must only be exposed in developer builds of the inter-domain services. It must not be available in production, even to a user with elevated privileges. To expose it would allow all inter-domain communications to be decrypted.

External watchdog

There must be an external watchdog system which watches both the automotive and consumer–electronics domains, and which restarts either of them if they crash and fail to restart themselves.

In order to prevent one compromised domain from preventing a restart of the other domain (a denial of service attack), each domain must only be able to send

[52] https://ask.wireshark.org/questions/12019/how-can-i-decrypt-ikev1-andor-esp-packets
[53] https://ask.wireshark.org/questions/44562/update-decryption-table-from-lua

heartbeats to its own watchdog, and not the watchdog of the other domain.

The implementation of the watchdog depends on the configuration:

- Standalone setup: No watchdog is necessary, as the configuration is not safety critical.
- Basic virtualised setup: The watchdog should be a software component in the hypervisor, exposed as virtualised watchdog hardware in the guests.
- Separate CPUs setup: A hardware watchdog on the board should be used, connected to both domains. As an exception to the general principle that the CE domain should not be allowed to access hardware, it must be able to access its own watchdog (and must not be able to access the automotive domain's watchdog).
- Separate boards setup: A hardware watchdog on each board should be used, connected to the domain on that board.
- Separate boards setup with other devices: Same as the separate boards setup.
- Multiple CE domains setup: Same as the separate boards setup.

Tamper evidence and hardware encryption

The basic design for providing a root of confidentiality and integrity for the system in hardware should be provided by the Secure Boot design — this design can only assume that some confidential encryption key is provided which is used to decrypt parts of the system on boot which should remain confidential.

As of February 2016 the Secure Boot design is still forthcoming

One possibility for implementing this is for a confidential key store to be provided by the automotive domain, storing keys which encrypt the bootloader and root key store for the CE. When booting the CE, the AD would decrypt its bootloader and hence its root key store, making the keys necessary for inter-domain communications (amongst others) available in the CE's memory. Note that this suggestion should be ignored if it conflicts with recommendations in the Secure Boot design, once that's published.

A critical requirement of the system is that none of the keys for encrypting inter-domain communications (or for protecting those keys) can be shared between vehicles — they must be unique per vehicle (No global keys in vehicles). This implies that keys must be generated and embedded into each vehicle as a stage in the imaging process for the domains.

A corollary to this is that none of those keys can be stored by the vendor, trusted dealer or other global organisations associated with the vehicles; as to do so would provide a single point of failure which, if compromised by an attacker, could reveal the keys for all vehicles and hence potentially allow them all to be compromised easily.

Tamper evidence is an important requirement for the system (Tamper evidence), providing the ability to determine if a vehicle has been tampered with in case of an accident or liability claim.

The most appropriate way to provide tamper evidence for the hardware depends on the hardware and how it is packaged in the vehicle. Typical approaches to tamper evidence involve sealing the domain's circuitry, including all access and I/O ports, in a casing which is sealed with tamper evident seals[54]. If a garage or trusted vehicle dealer needs to access the domain for maintenance or updates, they must break the seals, enter this in the vehicle's maintenance log, and replace the seals with new ones once maintenance is complete.

Tamper evidence for software should be provided through the integrity properties of the Secure Boot design, as in any trusted platform module[55] system.

Disabling the CE domain

The automotive domain must be able to disable the power supply to the CE domain (or otherwise prevent it from booting), and must be able to prevent inter-domain communications at the same time.

Preventing inter-domain communications should be implemented by having the automotive domain inter-domain service read a 'kill switch' setting. If this is set, it should close any open inter-domain communication links, and refuse to accept new ones while the setting is still set.

Preventing the CE domain from booting can be done in a variety of ways, depending on the hardware functionality available. For example, it could be done by controlling a solid-state relay on the CE domain's power supply. Or, if the CE domain implements secure boot, the boot process could require the automotive domain to decrypt part of the CE domain bootloader using a key known only to the automotive domain — if the kill switch is set, this key would be unavailable.

Open question: What hardware provisions are available for controlling the power supply or boot process of the CE domain? How should this integrate with the secure boot design?

The kill switch is intentionally kept simple, controlling whether *all* inter-domain communications are enabled or disabled, and providing no finer granularity. This is intended to make it completely robust — if support were added for selectively killing some of the control APIs or data connections on the inter-domain communications link, but not others, there would be much greater scope for bugs in the kill switch which could be exploited to circumvent it.

[54] https://en.wikipedia.org/wiki/Security_seal
[55] https://en.wikipedia.org/wiki/Trusted_Platform_Module

If the OEM wants to provide finer grained kill switches for different APIs in the automotive domain, they must implement them as part of those services, or as part of the export layer which connects those services to the inter-domain service.

Reporting malicious applications

There are three options for reporting malicious behaviour of applications to the Apertis store:

1. Report from the inter-domain service in the automotive domain, based on error responses from the OEM APIs.
2. Report from the inter-domain service in the CE domain, based on error responses from the automotive domain.
3. Report from the SDK API adapter layers, based on error responses from the automotive domain.

They are presented in decreasing order of reliability, and increasing order of helpfulness.

Option #1 is reliable (an attacker can only prevent a detected malicious action from being reported by compromising the automotive domain), but not helpful (the automotive domain does not have contextual information about the access, such as the application bundle which originally made the request — bundle identifiers cannot be sent across the inter-domain link as that would mean partially defining the OEM APIs). This option has the additional disadvantage that it requires the AD to communicate directly with the Apertis store without going via the CE, which likely means the AD is on the Internet and could potentially be compromised by a Heartbleed-style vulnerability in a communication path that was intended to be secure. Options #2 and #3 do not have this disadvantage, because in those options it is the CE that needs to communicate on the Internet.

Option #3 is unreliable (an attacker can prevent a detected malicious action from being reported by compromising that SDK service in the CE domain), but most helpful (the CE domain knows all contextual information about the access, including the application bundle identifier, parameters sent to the SDK API by the application, and the output of the adapter layer which was sent to the inter-domain link).

We recommend option #3 as it is the most helpful, and we believe that the additional contextual information it provides outweighs the potential loss of reports from most severely compromised vehicles. This is one part of many which contribute to the security of the system.

An alternative would be to implement two or all of the options, and leave it up to the Apertis store software to combine or deduplicate the reports.

Suggested roadmap

One the design has been reviewed, it can be compared to the existing state of the inter-domain communication system, and a roadmap produced for how to reconcile the differences (if there are any).

Open question: How does this design compare to the existing state of the inter-domain communication system?

Requirements

Open question: Once the design is finalised a little more, it can be related back to the requirements to ensure they are all satisfied.

Open questions

- Existing inter-domain communication systems: Are there any relevant existing systems to compare against?
- IPSec versus TLS: What is the security of the IPsec protocol in its current (2015) state?
- IPSec versus TLS: What is the performance of TCP and UDP over IPsec, TLS over TCP and DTLS over UDP on the Apertis reference hardware?
- Configuration designs: What trade-off do we want between performance and testability for the different transport layer configurations?
- Configuration designs: What more detailed configuration options can we specify for setting up IPsec? For example, disabling various optional features which are not needed, to reduce the attack surface. What IKE service should be used?
- Configuration designs: A lot of business logic for control over OEM licencing can be implemented by the choice of the CA hierarchy used by the inter-domain communication system. What business logic should be possible to implement?
- Configuration designs: Consider key control, revocation, protocol obsolescence, and various extensions for pinning keys and protocols.
- Configuration designs: What can be done in the automotive domain to reduce the possibility of exploits like Heartbleed affecting the inter-domain communications link? This is a trade-off between the stability of AD updates (high; rarely released) and the pace of IPsec and TLS security research and updates and the need for crypto-agility (fast). Heartbleed was a bug in a bad implementation of an optional and not-very-useful TLS extension.
- Control protocol: How should the multiple CE configuration (section 8.3.2) interact with D-Bus signals? Can the adapter layer perform the

broadcast to all subscribers?

- Developer mode: What cryptography should be used to implement this authentication, and the division of trust between development and production devices? A likely solution is to only have the AD accept the CE connection if it connects with a 'production' key signed by the vehicle OEM.
- Disabling the CE domain: What hardware provisions are available for controlling the power supply or boot process of the CE domain? How should this integrate with the secure boot design?
- Suggested roadmap: How does this design compare to the existing state of the inter-domain communication system?
- Requirements: Once the design is finalised a little more, it can be related back to the requirements to ensure they are all satisfied.

Summary of recommendations

Open question: Once the design is finalised a little more, and a suggested roadmap has been produced (Suggested roadmap), it can be summarised here.

Appendix: D-Bus components and licensing

The terminology around D-Bus can sometimes be confusing; here are some details of its components and their licensing.

- *D-Bus* is a protocol[56] which defines an on-the-wire format for marshalling and passing messages between peers, a type system for structuring those messages, an authentication protocol for connecting peers, a set of transports for sending messages over different underlying connection media, and a series of high-level APIs for implementing common API design patterns such as properties and object enumeration. It has a reference implementation (libdbus and dbus-daemon), but these are by no means the only implementations. The protocol has had full backwards compatibility since 2006[57].
- A *D-Bus daemon* (for example: dbus-daemon, kdbus) is a process which arbitrates communication between D-Bus peers, implementing multicast communications (such as signals) without requiring all peers to connect to each other. Different D-Bus daemons have different performance characteristics and licensing. For example, kdbus runs in the kernel to improve performance by reducing context switching overhead, at the cost of some features; dbus-daemon runs in user space with more overhead, but is still quite performant.

[56] http://dbus.freedesktop.org/doc/dbus-specification.html
[57] http://dbus.freedesktop.org/doc/dbus-specification.html#stability

- A *D-Bus library* (for example: libdbus, GDBus) is a set of code which implements the D-Bus protocol for one peer, converting high-level D-Bus API calls into on-the-wire messages to send to another peer or a D-Bus daemon to send to other peers. Different D-Bus libraries have different performance characteristics and licensing.

Licensing

- The D-Bus Specification is freely licensed and has no restrictions on who may implement it or how those implementations are licensed.
- libdbus and dbus-daemon are both licensed under your choice of the AFLv2.1[58], or the GPLv2[59] (or later versions).
  - Hence, if the AFL license is chosen, libdbus and dbus-daemon may be used in non-open-source products.
- GDBus is part of GLib, and hence is licensed under the LGPLv2.0[60] (or later versions).

Appendix: D-Bus performance

libdbus and dbus-daemon are reasonably performant, having been used in various low-resource products (such as mobile phones) over the years. There have not been any quantitative evaluations of their performance in terms of latency or memory usage recently, but some have been done in[61] the[62] past[63].

As indicative numbers *only*, D-Bus (using dbus-python[64] and dbus-daemon, not kdbus) gives performance of roughly:

- 20,000 messages per second throughput
- 130MB per second bandwidth
- 0.1s end-to-end latency between peers for a given message
  - This is likely an overestimate, as ping-pong tests written in C have given latency of 200µs
- 2.5MB memory footprint (RSS) for dbus-daemon in a desktop configuration

[58] https://spdx.org/licenses/AFL-2.1.html
[59] http://spdx.org/licenses/GPL-2.0+
[60] http://spdx.org/licenses/LGPL-2.0+
[61] https://desktopsummit.org/sites/www.desktopsummit.org/files/will-thompson-dbus-performance.pdf
[62] http://blog.asleson.org/index.php/2015/09/01/d-bus-signaling-performance/
[63] https://blogs.gnome.org/abustany/2010/05/20/ipc-performance-the-return-of-the-report/
[64] http://www.freedesktop.org/wiki/Software/DBusBindings/

    - So this could likely be reduced if needed — the amount of message buffering dbus-daemon provides is configurable

Note that these numbers are from performance evaluations on various versions of dbus-daemon, so should be considered indicative of an order of magnitude only. As with all performance measurements, accurate values can only be measured on the target system in the target configuration.

The most commonly accepted disadvantage of using D-Bus with dbus-daemon is the end-to-end latency needed to send a message from one peer, through the kernel, to the dbus-daemon, then through the kernel again, to the receiving peer. This can be reduced by using kdbus, which halves the number of context switches needed by implementing the D-Bus daemon in kernel space[65]. However, kdbus has not yet been accepted into the upstream kernel, and (as of February 2016) there is some concern that this might not happen due to kernel politics. It can be integrated into distributions as a kernel module, although it relies on a few features only available in kernel version 4.0 or newer. This means it should be straightforward to integrate in the CE, but potentially not in the AD (and certainly not if the AD doesn't run Linux — in such cases, dbus-daemon can be used).

Overall, the performance of a D-Bus API depends strongly on the API design. Good [D-Bus API design] eliminates redundant round trips (which have a high latency cost), and offloads high-bandwidth or latency sensitive data transfer into side channels such as UNIX pipes, whose identifiers are sent in the D-Bus API calls as FD handles[66].

Appendix: Software versus hardware encryption

The choice about whether to use software or hardware encryption is a tradeoff between the advantages and disadvantages of the options. There are actually several ways of providing 'hardware encryption', which should be considered separately. In order from simplest to most complex:

- **Encryption acceleration instructions** in the processor, such as the AES instruction set[67], CLMUL[68] or the ARM cryptography extensions[69]. These are available in most processors now, and provide assembly instructions for performing expensive operations specific to certain encryption standards, typically AES, SHA and Galois/Counter Mode (GCM) for block ciphers. Intel architectures have the most extensions, but ARM architectures also have some.
- **Secure cryptoprocessor**[70]. These are separate, hardened hardware de-

[65] http://www.freedesktop.org/wiki/Software/systemd/kdbus/
[66] http://dbus.freedesktop.org/doc/dbus-specification.html#idp9446907251
[67] https://en.wikipedia.org/wiki/AES_instruction_set
[68] https://en.wikipedia.org/wiki/CLMUL_instruction_set
[69] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0514g/index.html
[70] https://en.wikipedia.org/wiki/Secure_cryptoprocessor

vices which implement all encryption operations and some key storage and handling within a tamper-proof chip. They are conceptually similar to hardware video decoders — the CPU hands off encryption operations to the coprocessor to happen in the background. They typically do not have their own memory.

- **Hardware security module**[71] (HSM). These are even more hardened secure cryptoprocessors, which typically come with their own tamper-proof memory and supporting circuitry, including tamper-proof power supply. They handle all aspects of encryption, including all key storage and management (such that keys never leave the HSM).

Software encryption (without encryption acceleration instructions)

- Lowest encryption bandwidth.
- Highest attack surface area, as keys and in-progress encryption values have to be stored in system memory, which can be read by an attacker with physical access to the hardware.
- Certain versions of some cryptographic libraries are FIPS[72]-certified, but not all. GnuTLS has been FIPS certified in various devices, but is not routinely certified[73]. OpenSSL is not routinely certified, but provides a OpenSSL FIPS Object Module which *is* certified[74] as a drop-in replacement for OpenSSL, provided that it's used unmodified. The Linux kernel's IPsec support has been certified in Red Hat Enterprise Linux 6, but is not routinely certified[75].
- Cheaper than hardware.
- Provides the possibility of upgrading to use different encryption algorithms in future.
- Possible to check the software implementation for backdoors, although it's a lot of work. Some of this work is being done by other users of open source encryption software[76].

Software encryption (with encryption acceleration instructions)

- Same advantages and disadvantages as software encryption without encryption acceleration instructions, except that the use of acceleration gives

[71] https://en.wikipedia.org/wiki/Hardware_security_module
[72] https://en.wikipedia.org/wiki/FIPS_140-2
[73] http://www.gnutls.org/manual/html_node/Certification.html
[74] https://www.openssl.org/docs/fips.html
[75] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/sect-Security_Guide-Federal_Standards_And_Regulations-Federal_Information_Processing_Standard.html
[76] http://www.zdnet.com/article/ncc-group-to-audit-openssl-for-security-holes/

a higher encryption bandwidth (on the order of a factor of 10 improvement).

- Same software interface as without acceleration.
- Both TLS and IPsec provide various cipher suite options, at least some of which would benefit from hardware acceleration — both use AES-GCM[77] for data encryption, which benefits from AES instructions.

Secure cryptoprocessor

- Higher encryption bandwidth.
- Reduced attack surface area, as keys and in-progress encryption values are handled within the encryption hardware, rather than in general memory, and hence cannot be accessed by an attacker with physical access. Keys may still leave the cryptoprocessor, which gives some attack surface.
- Typical secure cryptoprocessors have tamper evidence features in the hardware.
- Typically hardware is FIPS-certified.
- More expensive than software.
- Provides a limited set of encryption algorithms, with no option to upgrade them once it's fixed in silicon.
- No possibility to audit the hardware implementation to check for backdoors, so you have to trust that the hardware vendor has not been secretly required to provide a backdoor by some government.
- Typical cryptoprocessors originate from mobile or embedded networking hardware, both of which need to support TLS, and hence cryptoprocessors typically have support for AES, DES, 3DES and SHA. This is sufficient for accelerating the common cipher suites in TLS and IPsec.
- Have to be supported by the Linux kernel crypto API (`/dev/crypto`) in order to be usable from software.

Hardware security module

- Highest encryption bandwidth.
- Minimal attack surface area, with keys never leaving the HSM.
- All hardware is tamper-proof and tamper-evident, and typically can destroy stored keys automatically if tampering is detected.
- Hardware is almost universally FIPS-certified.
- Most expensive.

[77] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

- Provides a range of encryption algorithms, but with no option to upgrade them.
- No possibility to audit the hardware implementation to check for backdoors, so you have to trust that the hardware vendor has not been secretly required to provide a backdoor by some government.
- Some modules can handle encryption of network streams transparently, taking a plaintext network stream as input and handling all TLS or IPsec operations for it with peers.

Conclusion

According to one evaluation[78], using encryption acceleration instructions should reduce the number of cycles per byte for AES encryption from 28 to 3.5. Assuming the inter-domain connection is being used to transmit a HD video at 250kB·s ^{1}, that means encryption requires 7MHz of CPU compute without acceleration, and 875kHz with it. Performing symmetric encryption on a data stream doesn't significantly increase the required memory bandwidth compared to copying the stream around without encryption.

Hence, overall, if we assume a peak bandwidth requirement on the inter-domain communications link on the order of 250kB·s ^{1} then using software encryption with acceleration instructions should give sufficient performance.

The hardware security (tamper-proofing) provided by a HSM is overkill for an in-vehicle system, and is better suited to data centres or military equipment. We recommend either using software encryption with acceleration, or a secure cryptoprocessor, depending on the balance of the advantages and disadvantages of the two for the particular OEM and vehicle. For the purposes of this design, both options provide all features necessary for inter-domain communications.

Appendix: Audio and video streaming standards

There are several standards to enable reliable audio and video streaming between various systems. These standards aim to address the synchronization problem with different approaches.

- AES67[79]: The AES67 standard combines PTP and RTP using PTP clock source signalling (RFC7273[80]) to synchronize multiple streams with an external clock, focusing on high-performance audio based on RTP/UDP.
- VSF TR-03: This is a technical recommendation from the Video Service Forum[81] (VFS). The TR-03 standard is similar to AES67 in terms of using

[78] https://en.wikipedia.org/wiki/AES_instruction_set#Performance
[79] https://en.wikipedia.org/wiki/AES67
[80] https://tools.ietf.org/html/rfc7273
[81] http://www.videoservicesforum.org/

PTP for clock synchronization, but it extends AES67 to cover other kinds of uncompressed streams, including video and metadata.

- AVB[82]: The Audio Video Bridging (AVB) is a small extensions to standard layer-2 MACs and bridges. An advantage of AVB is that the time synchronization information is periodically exchanged through the network so it provides great synchronization precision. However, it requires to implement AVB for all of devices in the network because the device should allocate a fraction of network bandwith for AVB traffic.

The following comparison table depicts the characteristics of the standards.

	AES67	VSF TR-03	AVB
Time synchronization	external (PTP)	external (PTP)	supported by the network
Kernel support	not required	not required	required
Transport protocol	RTP	RTP	RTP, HTTP(s), IEEE 1722
Related open source project	GStreamer	N/A	OpenAvnu

Note that VFS TR-03 has no explicit open source implementation, but as it combines RTP for transport and PTP for clock synchronization, it is generally supported by GStreamer.

Appendix: Multiplexing RTP and RTCP

RTP requires the RTP Control Protocol (RTCP) to exchange control packets and timing information such as latency and QoS. Usually RTP and RTCP use two different channels on different network ports, but it is also possible to use a single port for both protocols using the RFC 5761[83] standard, supported by the GStreamer `funnel` element.

The following diagram shows how a RFC 5761 pipeline can be set up in GStreamer:

```
/--------\     /---------\     /--------\         /---------------\          /-
---------\
| audio  | === | audio   | === | rtpbin | = rtp = | rtp payloader |  = rtp = |        |    /-
---------\
|  source  |        |  convert  |        |            |             \---------------
/           |   funnel   | === | udp sink |
\--------/    \---------/    |       | ========================== rtcp = |        |    \-
---------/
                                \--------/                              \----------/
```

[82] https://en.wikipedia.org/wiki/Audio_Video_Bridging
[83] https://tools.ietf.org/html/rfc5761

Appendix: Audio and video decoding

As a system which handles a lot of multimedia, deciding where to perform audio and video decoding is important. There are two major considerations:

- minimising the amount of raw communications bandwidth which is needed to transmit audio or video data between the domains; and
- ensuring that an exploit does not give access to arbitrary memory from either domain (especially not the automotive domain).

The discussion below refers to video encoding and decoding, but the same considerations apply equally well to audio.

Software encoding is a large CPU burden, and introduces quality loss into videos — so decoding and re-encoding videos in one domain to check their well-formedness is not a viable option. If decoding is being performed, the decoded output might as well be used in that form, rather than being re-encoded to be sent to the other domain.

In order to avoid spending a lot of CPU time and CPU–memory bandwidth on video decoding, it should be performed by hardware. However, this hardware does not necessarily have to be in the domain where the encoded video originates. For example, it is entirely possible for videos to be sent from the CE to be decoded in the AD.

The original designs which were discussed in combination with the GPU video sharing design planned to create a GStreamer plugin in the CE which treats the AD as a hardware video decoder which accepts encoded video, decodes it, and returns a handle which can be passed to the GL scene being output by the CE, via a GL extension (similar to EXT_image_dma_buf_import[84]). This is the same model as used for 'normal' hardware decoders, and ensures that decoded video data remains within the AD, rather than being sent back over the inter-domain communications link (which would incur a very high bandwidth cost, which for uncompressed 1080p video in YUV 422 format at 60fps amounts to $16\ \text{bits/pixel} \times (1920 \times 1080)\ \text{pixels/frame} \times 60\ \text{frames/s} = 1898\ \text{Mbit/s} = 237\ \text{MB/s}$).

Regarding security, a hardware decoder is typically a DMA[85]-capable peripheral which means that, unless constrained by an IOMMU[86], it can access all areas of physical memory. The threat here is that a malicious or corrupt video could trigger the decoder into reading or writing to areas of memory which it shouldn't, which could allow it to overwrite parts of the (hypervisor) operating system or running applications. This concern exists regardless of which domain is driving the decoder. We highly recommend that hardware is chosen which uses an IOMMU to restrict the access a video decoder has to physical memory.

[84] https://www.khronos.org/registry/egl/extensions/EXT/EGL_EXT_image_dma_buf_import.txt
[85] https://en.wikipedia.org/wiki/Direct_memory_access
[86] https://en.wikipedia.org/wiki/Input-output_memory_management_unit

Note that the same security threat applies to the GPU, which has direct access to physical memory (if shared with the CPU — some systems use dedicated memory for the GPU, in which case the issue isn't present). GPUs have a much larger attack surface, as they have to handle complex GL commands which are provided from untrusted sources, such as WebGL.

We recommend investigating the hardening and security applied to video decoders on the particular hardware platforms in use, but there is not much which can be done by software to improve their security if it is lacking — the performance cost is too high.

Memory bandwith usage on the i.MX6 Sabrelite

This section refers to some benchmarks evaluating the available memory bandwidth on the i.MX6 Sabrelite platform used in the reference hardware for Apertis. This data is very system dependent, but the order of magnitude should provide a general guide for evaluating approaches.

The iMX6 memory bandwith usage benchmark[87] describes some tools that can be used to measure how memory is used, and reports that a 1080p @ 60fps loopback pipline[88] using GStreamer requires up to 1744.46 MB/s of memory bandwidth.

Another useful benchmark is the one evaluating the cost of memory copies[89] done with the `memcpy()` function. The effective usable memory bandwidth measured with this test amounts to roughly 800 MB/s.

Security Vulnerabilities in GStreamer

To list vulnerabilities by type we can refer to the statistics available from the CVE[90] data source.

According to the CVE Details[91] website, a third party that provides summaries of CVE vulnerabilities, GStreamer had total 17 vulnerabilities[92] since 2009.

Examining the DoS and Code Execution vulnerability types, the statistics showed different characteristics for demuxers and decoders. There have been 12 DoS vulnerabilities affecting demuxers, but only one issue could lead to Code Execution. For decoders, all the the 5 DoS issues which were found can be escalated to Code Execution as well.

[87] https://developer.ridgerun.com/wiki/index.php?title=IMX6_Memory_Bandwidth_usage

[88] https://developer.ridgerun.com/wiki/index.php?title=IMX6_Memory_Bandwidth_usage#1080p60_loopback

[89] https://community.nxp.com/thread/309197

[90] http://cve.mitre.org/

[91] https://www.cvedetails.com

[92] https://www.cvedetails.com/vendor/9481/Gstreamer.html

This report indicates that demuxers might have a smaller attack surface than decoders from the arbitrary code execution viewpoint. However, it is also possible to have a security hole similar to Video or audio decoder bugs.

Both demuxing and possibly even decoding in the CE can help to mitigate the described attacks. If the CE is responsible of demuxing the AD does not need to deal with content detection and container formats, and the CE provides a kind of partial verification of the stream even without decoding it.

Decoding in the CE poses some challenges in terms of bandwidth, as the amount of data generated by fully decoded video streams is very high. It's not going to be a viable solution on ethernet-based setups, and advanced zero-copy mechanisms to transfer frames are recommended on single board setups (virtualised or container-based).