



Inter-domain communication

Contents

2	Terminology and concepts	5
3	Automotive domain	5
4	Consumer-electronics domain	5
5	Connectivity domain	6
6	Trusted path	6
7	Control stream	6
8	Data stream	6
9	Traffic control	7
10	Use cases	7
11	Standalone setup	7
12	Basic virtualised setup	7
13	Linux container setup	7
14	Separate CPUs setup	8
15	Separate boards setup	8
16	Separate boards setup with other devices	9
17	Multiple CE domains setup	9
18	Touchscreen events	9
19	Wi-Fi access	9
20	Bluetooth access	10
21	Audio transfer	11
22	Video decoding	12
23	Streaming media	13
24	Downloads of firmware updates	13
25	Offline and online map data	13
26	Phonebook integration	13
27	Tinkering vehicle owner on the network	14
28	Tinkering vehicle owner on the boards	14
29	Support multiple AD operating systems	14
30	Before-market upgrades	14
31	After-market upgrades	15
32	Testability	15
33	Malicious CE	15
34	Malicious CD	15
35	After-market upgrade of a domain	15
36	Power cycle independence of domains (CE down)	16
37	Power cycle independence of domains (AD down, single screen)	16
38	Power cycle independence of domains (AD down, multiple screens)	16
39	Temporary communications problem	17
40	New version of AD software	17
41	New version of AD interfaces	17
42	Unsupported AD interfaces	18
43	Contacts sharing	18
44	Protocol compatibility	18
45	Navigation system	19

46	Marshalling resource usage	19
47	Feedback for malicious applications	19
48	Compromised CE with delayed fix	19
49	Denial of service through flooding	20
50	Malicious CE UI	20
51	Plug-and-play CE device	20
52	Connecting an SDK to a development vehicle	20
53	Security model	21
54	Attackers	21
55	Security domains	23
56	Security model	23
57	Non-use-cases	24
58	Production CE domain used in multiple configurations	24
59	Requirements	24
60	Separated transport layer	24
61	Message integrity and confidentiality in transport layer	25
62	Reliability and error checking in transport layer	26
63	Mutual authentication between domains	26
64	Separate authentication for developer and production mode devices	26
65	Individually addressed domains	26
66	Traffic control for latency	26
67	Traffic control for bandwidth	26
68	Traffic control for frequency	27
69	Separation of control and data streams	27
70	No untrusted access to AD hardware	27
71	Trusted path for users to update the CE operating system	28
72	Safety limits on AD APIs	28
73	Rate limiting on control messages	28
74	Ignore unrecognised messages	28
75	Portable transport layer	29
76	Support push mode and pull mode communications	29
77	OEM AD integration API	29
78	Flexibility in OEM AD integration API	29
79	Inflexibility in OEM AD integration API	29
80	Service discovery	30
81	Stability in inter-domain communications protocol	30
82	Testability of protocols	30
83	Testability of protocol parsers and writers	30
84	Testability of processes	30
85	CE system services separated from transport layer	31
86	No dependency on CE specific hardware	31
87	Immediate error response if service on peer is unavailable	31
88	Immediate error response if peer is unavailable	32
89	Timeout error response if peer does not respond	32
90	All inter-domain communications APIs are asynchronous	32
91	Reconnect to peer as soon as it is available	32

92	External domain watchdog	33
93	Reporting system for malicious applications	33
94	Ability to disable the consumer-electronics domain	33
95	Tamper evidence	33
96	No global keys in vehicles	34
97	Existing inter-domain communication systems	34
98	Approach	34
99	Overall architecture	34
100	Security domains	37
101	Protocol design	38
102	Traffic control	55
103	Protocol library and inter-domain services	55
104	Non Linux-based domains	56
105	Service discovery	57
106	Automotive domain export layer	59
107	Consumer-electronics domain adapter layer	60
108	Interaction of the export and adapter layers	60
109	Flow for a given SDK API call	61
110	Trusted path to the AD	62
111	Developer mode	62
112	Mock SDK implementation	63
113	Debuggability	64
114	External watchdog	65
115	Tamper evidence and hardware encryption	66
116	Disabling the CE domain	67
117	Reporting malicious applications	67
118	Suggested roadmap	68
119	Requirements	68
120	Open questions	68
121	Summary of recommendations	69
122	Appendix: D-Bus components and licensing	70
123	Licensing	70
124	Appendix: D-Bus performance	71
125	Appendix: Software versus hardware encryption	72
126	Software encryption (without encryption acceleration instructions)	72
127	Software encryption (with encryption acceleration instructions)	73
128	Secure cryptoprocessor	73
129	Hardware security module	74
130	Conclusion	74
131	Appendix: Audio and video streaming standards	75
132	Appendix: Multiplexing RTP and RTCP	76
133	Appendix: Audio and video decoding	76
134	Memory bandwidth usage on the i.MX6 Sabrelite	77
135	Security Vulnerabilities in GStreamer	78
136	This documents a suggested design for an inter-domain communication sys-	

tem, 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>

172 **Connectivity domain**

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

178 **Trusted path**

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

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

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

191 **Control stream**

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

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

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

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

205 **Data stream**

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

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

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

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

211 **Traffic control**

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

216 **Use cases**

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

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

223 **Standalone setup**

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

230 **Basic virtualised setup**

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

238 **Linux container setup**

239 Containers are based on Linux kernel containment features, including, but not
240 limited to, Linux kernel namespaces, control groups, chroots (pivot_root), ca-
241 pabilities.

³https://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 Aptaris 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.

281 **Separate boards setup with other devices**

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

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

295 **Multiple CE domains setup**

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

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

309 **Touchscreen events**

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

317 **Wi-Fi access**

318 The Wi-Fi hardware is controlled by the AD or CD. The CE needs to use it
319 for internet access, including connecting to a network. The Wi-Fi device can

320 return data at high bandwidth, but also has a separate control channel. The
 321 control channel always needs to be available, even if traffic is being dropped due
 322 to bandwidth limitations in the inter-domain communication channel.

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

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

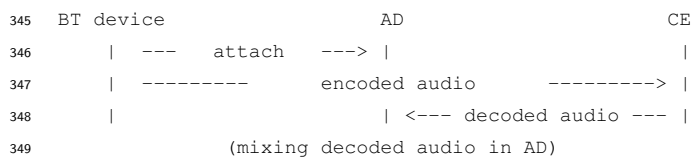
332 Bluetooth access

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

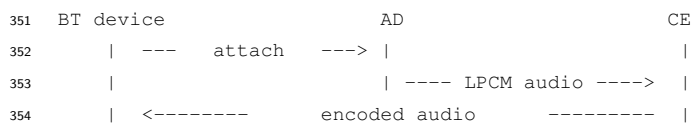
337 To support the A2DP and HSP/HFP audio profiles it may be desirable to keep
 338 the CE in charge of decoding and encoding the audio streams coming from
 339 and directed to the Bluetooth devices. The AD will be responsible for mixing
 340 the output streams directed to the car speakers and capturing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

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

344 Sending audio stream from BT to AD



350 Sending audio stream from AD to BT



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



```

359 | <----- control ----- |
360 | | |
361 | ----- encoded audio ----> |
362 | | ----- LPCM audio ----> |
363 | | <----- LPCM audio ---- |
364 | <----- encoded audio ----- |

```

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

```

367 BT device          CD          CE          AD
368 | ---- attach -----> | | |
369 | <--- control ----- | | |
370 | | <---- scan ----- | | |
371 | | ---- result ----> | | |
372 | | <---- play ----- | | |
373 | | | |
374 | ---- encoded audio ----> | | |
375 | | ----- LPCM audio -----> | | |
376 | | <----- LPCM audio ----- | | |
377 | <--- encoded audio ----- | | |

```

378 Multiple variations are possible on this model.

379 Audio transfer

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

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

387 Metadata can be sent alongside the audio, such as track names or timing infor-
388 mation.

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

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

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

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

414 Video decoding

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

420 (See the appendix on [Audio and video decoding](#) for a discussion of options for
421 video and audio decoding.)

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

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

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

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

435 **Streaming media**

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

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

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

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

452 **Downloads of firmware updates**

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

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

459 **Offline and online map data**

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

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

464 **Phonebook integration**

465 A phonebook agent in the Connectivity domain retrieves approximately 500
466 256×256px profile pictures, validates and re-encodes them to PNG and makes
467 them available to the Automotive domain, possibly using an uncompressed zip
468 file instead of sharing 500 files.

469 **Tinkering vehicle owner on the network**

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

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

478 **Tinkering vehicle owner on the boards**

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

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

487 [Secure bootloading](#)⁵ itself is a separate topic.

488 **Support multiple AD operating systems**

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

499 **Before-market upgrades**

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

⁵<https://apertis-website-0b3586.pages.apertis.org/architecture/platform/secure-boot/>

506 **After-market upgrades**

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

513 **Testability**

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

520 **Malicious CE**

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

527 **Malicious CD**

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

533 **After-market upgrade of a domain**

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

543 **Power cycle independence of domains (CE down)**

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

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

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

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

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

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

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

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

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

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

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

579 **Temporary communications problem**

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

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

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

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

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

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

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

601 **New version of AD software**

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

608 **New version of AD interfaces**

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

616 version 15.06. They will release the vehicle using this version B of their AD
617 operating system, and a tweaked version 15.06 of Apertis.

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

622 **Unsupported AD interfaces**

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

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

635 **Contacts sharing**

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

640 **Protocol compatibility**

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

648 **kdbus protocol compatibility** If, for example, the inter-domain commu-
649 nication protocol is implemented using dbus-daemon in version A of the AD
650 operating system, and in the corresponding version A of Apertis; and version B
651 of Apertis uses kdbus instead of dbus-daemon, the two OSs must still commu-
652 nicate successfully.

653 **Navigation system**

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

662 **Marshalling resource usage**

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

671 **Feedback for malicious applications**

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

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

681 **Compromised CE with delayed fix**

682 An attacker has somehow completely compromised the CE operating system,
683 and has root access to it. It will take the OEM a few weeks to produce, test
684 and distribute a fix for the exploit used by the attacker, but vehicle owners
685 would like to continue to use their vehicles, with reduced functionality (no CE
686 domain) in the meantime, because the attack has not compromised the AD. The
687 OEM has provided them with an authenticated method of informing the AD
688 to shut down the CE and keep it shut down until an authenticated update has
689 been applied and has fixed the exploit and removed the attacker from the CE
690 (probably by overwriting the entire OS with a fresh copy). This update can only
691 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.

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

734 Security model

735 See the [Security concept design](#)⁶ for general terminology including the defini-
736 tions used for *integrity*, *availability*, *confidentiality* and *trust*.

737 Attackers

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

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

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

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

755 See the [Conditional Access design](#)⁸

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

759 **Passenger** The passenger is a special kind of third party attacker (**Third**
760 **parties**), who additionally has access to the in-vehicle network. This may be
761 possible if, for example, the Apertis device in the vehicle is removable so it can
762 be passed to a passenger, exposing a connector behind it.

⁶https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_security/security/

⁷<http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.html>

⁸https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_security/conditional_access/

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

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

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

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

781 A third party attacker may be trying to:

- 782 • Access confidential information belonging to the vehicle owner.
- 783 • Compromise the integrity of the vehicle's control system (the automotive
784 domain). For example, to trigger unintended acceleration or to change
785 the radio channel to spook the driver.
- 786 • Compromise the integrity of the CE domain to, for example, make it part
787 of a botnet, or cause it to call premium rate numbers owned by the attacker
788 to generate money.
- 789 • Compromise the availability of the vehicle's control system (the automo-
790 tive domain) to bring the vehicle to a halt.
- 791 • Compromise the availability of the vehicle's infotainment system (the CE
792 domain) to cause a nuisance to the driver or passengers.
- 793 • Compromise the confidentiality of the device key (see the [Conditional
794 Access design](https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_security/conditional_access/)⁹) in order to extract licenced content (for example, music)
795 from application bundles.

796 **Trusted dealer** As above, all authorized vehicle dealers, garages or other
797 sale/repair locations have to be trusted, as they have more unsupervised ac-
798 cess to the vehicle's hardware, and more capabilities, than the vehicle owner,
799 passenger or a third party.

⁹https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application_security/conditional_access/

800 Security domains

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

818 Security model

- 819 • Domains must assume that the inter-domain communication link has no
 - 820 confidentiality or integrity, and is controlled by an attacker (a man in the
 - 821 middle with the ability to modify traffic)
 - 822 – This means they must not trust any traffic from other devices on the
 - 823 network
- 824 • The AD, CD and CE operating systems must assume all input from ex-
 - 825 ternal sources (Wi-Fi, Bluetooth, GPS, 4G, etc.) is malicious
- 826 • The CE operating system may assume all API calls from the AD (as
 - 827 proxied by the CE proxy) are *not* controlled by an attacker, assuming
 - 828 they have come over an authenticated channel which guarantees integrity
 - 829 between the AD and CE proxy; in other words, the AD must not deny
 - 830 confidentiality or integrity to the CE
- 831 • The AD may deny availability to the CE operating system, by closing the
 - 832 inter-domain link in response to the user disabling the CE while waiting
 - 833 for a critical security update

- 834 • The AD must assume all API calls from the CE are malicious, in case the
835 CE has been compromised
- 836 • The CE must assume that all input and output from third party applica-
837 tions in sandboxes is malicious, including all their API calls
- 838 • If a hypervisor is present:
 - 839 – The AD and CE operating systems may assume all control calls from
840 the hypervisor are *not* controlled by an attacker
 - 841 – The hypervisor must assume all input from the CE is malicious
 - 842 – The hypervisor may assume that all input from the AD is *not* mali-
843 cious
 - 844 * Note that, when combined with the fact that the AD cannot be
845 updated easily, this makes security bugs in the AD extremely
846 critical and extremely hard to fix
- 847 • Tampering with any domain software must be detectable even if it is not
848 preventable (tamper evidence)
- 849 • If one vehicle is attacked and compromised, the same effort must be re-
850 quired to compromise other vehicles

851 **Non-use-cases**

852 **Production CE domain used in multiple configurations**

853 A production CE domain operating system cannot be used in multiple config-
854 urations, for example as both an operating system running on one CPU of a
855 two-CPU board shared with the automotive domain OS; and then as an im-
856 age running on a separate board connected to an in-vehicle network with other
857 devices connected.

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

865 **Requirements**

866 **Separated transport layer**

867 The transport layer for transmitting inter-domain communications between the
868 domains must be separated from the APIs being transported, in order to allow

869 for different physical links between the domains, with different security proper-
870 ties.

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

874 See [Standalone setup](#).

875 **Transport over virtio** Support a configuration where the CE and AD com-
876 municate over a virtio link between two virtual machines under a hypervisor.

877 See [Basic virtualised setup](#).

878 **Transport over a private Ethernet link** Support a configuration where
879 the CE and AD are on separate CPUs and communicate over a point-to-point
880 Ethernet link.

881 See [Separate CPUs setup](#), [Separate boards setup](#).

882 **Transport over a private Ethernet link to a development vehicle** Sup-
883 port a configuration where the CE is running in an SDK on a laptop, and the
884 AD is running in a developer-mode Apertis device in a vehicle, and the two
885 communicate over a wider shared Ethernet.

886 See [Connecting an SDK to a development vehicle](#).

887 **Transport over a shared Ethernet link** Support a configuration where
888 the CE and AD are on separate CPUs are are both connected to some wider
889 shared Ethernet.

890 See [Separate boards setup with other devices](#), [Multiple CE domains setup](#).

891 **Transport over Unix Domain Socket** Support a configuration where AD
892 and CE are on the same host running as Linux containers and connected via
893 UDS. The same transport can be used on OEM deployments and on SDK envi-
894 ronments.

895 See [Linux container setup](#), [Multiple CE domains setup](#).

896 **Message integrity and confidentiality in transport layer**

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

900 See [Separate CPUs setup](#), [Separate boards setup](#), [Separate boards setup with](#)
901 [other devices](#), [Multiple CE domains setup](#), [Wi-Fi access](#).

902 **Reliability and error checking in transport layer**

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

906 See [Separate boards setup](#), [Separate boards setup with other devices](#), [Multiple](#)
907 [CE domains setup](#).

908 **Mutual authentication between domains**

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

912 See [Tinkering vehicle owner on the network](#).

913 **Separate authentication for developer and production mode devices**

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

918 See [Connecting an SDK to a development vehicle](#), [Connecting an SDK to a](#)
919 [production vehicle](#).

920 **Individually addressed domains**

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

925 See [Multiple CE domains setup](#).

926 **Traffic control for latency**

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

932 See [Touchscreen events](#).

933 **Traffic control for bandwidth**

934 In order to prevent some kinds of high bandwidth message from using all the
935 bandwidth provided by the physical link, the system must provide a traffic

936 control system to ensure all types of message have fair access to bandwidth
937 (where ‘fairness’ is measured according to some rigorous definition).

938 This may be implemented by separating ‘control’ and ‘data’ streams (see sections
939 2.4 and 2.5), or by applying traffic control algorithms.

940 See [Wi-Fi access](#), [Bluetooth access](#).

941 **Traffic control for frequency**

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

950 See [Denial of service through flooding](#).

951 **Separation of control and data streams**

952 Certain APIs will need to provide data and control streams separately, with dif-
953 ferent latency and bandwidth requirements for both. The system must support
954 multiple streams; this may be via an explicit separation between ‘control’ and
955 ‘data’ streams, or by applying traffic control algorithms.

956 See [Wi-Fi access](#), [Bluetooth access](#), [Audio transfer](#), [Video decoding](#).

957 **No untrusted access to AD hardware**

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

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

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

972 See [Audio transfer](#), [Video decoding](#), [Video or audio decoder bugs](#), [Connecting](#)
973 [an SDK to a production vehicle](#).

974 **Trusted path for users to update the CE operating system**

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

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

980 See [Video or audio decoder bugs](#), [After market CE upgrades](#), [Malicious CE UI](#).

981 **Safety limits on AD APIs**

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

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

990 See [Tinkering vehicle owner on the boards](#), [Malicious CE](#).

991 **Rate limiting on control messages**

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

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

999 See [Denial of service through flooding](#).

1000 **Ignore unrecognised messages**

1001 Both the CE and AD must ignore (and log warnings about) inter-domain com-
1002 munication messages which they do not recognise. If the message expects a
1003 reply, an error reply must be sent. The domains must not, for example, shut
1004 down or crash when receiving an unrecognised message, as that would lead to
1005 a denial of service vulnerability.

1006 See [Tinkering vehicle owner on the boards](#), [Malicious CE](#).

1007 **Portable transport layer**

1008 The transport layer must be portable to a variety of operating systems and
1009 architectures, in order that it may be used on different AD operating systems.
1010 This means, for example, that it must not depend on features added to very
1011 recent versions of the Linux kernel, or must have fallback implementations for
1012 them.

1013 See [Support multiple AD operating systems](#).

1014 **Support push mode and pull mode communications**

1015 The CE must be able to use pull mode communications with the AD, where
1016 it makes a method call and receives a reply; and push mode communications,
1017 where the AD emits a signal for an event, and the CE receives this.

1018 See [Support multiple AD operating systems](#).

1019 **OEM AD integration API**

1020 In order to allow any OEM to connect their AD to the system, there must
1021 be a well defined API which they connect their OEM-specific APIs for vehicle
1022 functionality to, in order for that functionality to be exposed over the inter-
1023 domain communication link.

1024 This API must support an implementation which uses the services in the Apertis
1025 SDK.

1026 See [Support multiple AD operating systems](#), [Standalone setup](#).

1027 **Flexibility in OEM AD integration API**

1028 As the functionality exported by different ADs differs, the integration API for
1029 connecting it to the inter-domain communication system must be a general one
1030 —it must not require certain functionality or data types, and must support func-
1031 tionality which was not initially expected, or which is not currently supported
1032 by any CE. This functionality should be exposed on the inter-domain commu-
1033 nications link, in case future versions of the CE can take advantage of it.

1034 See [Support multiple AD operating systems](#), [Before market CE upgrades](#), [After](#)
1035 [market CE upgrades](#), [New version of AD software](#), [New version of AD interfaces](#).

1036 **Inflexibility in OEM AD integration API**

1037 The OEM AD integration API must not allow access to arbitrary services or
1038 APIs on the AD. It must only allow access to the services and APIs explicitly
1039 exposed by the OEM in their use of the integration API.

1040 See [Unsupported AD interfaces](#).

1041 **Service discovery**

1042 Domains should be able to detect where specific services are hosted in case of
1043 multiple CE domains. If a service is moved from one CE domain to another
1044 CE domain, other domains should not require any reconfiguration. CE domains
1045 should not be able to spoof services that are meant to be provided by the AD.

1046 **Stability in inter-domain communications protocol**

1047 As the versions of the AD and CE change at different rates, the inter-domain
1048 communications protocol must be well defined and stable—it must not change
1049 incompatibly between one version of the CE and the next, for example.

1050 If the protocol uses versioning to add new features, both domains must support
1051 protocol version negotiation to find a version which is supported if the latest
1052 one is not.

1053 See [Before market CE upgrades](#), [After market CE upgrades](#), [New version of AD](#)
1054 [software](#), [Unsupported AD interfaces](#), [Protocol compatibility](#).

1055 **Testability of protocols**

1056 All IPC links in the inter-domain communications system must be testable in-
1057 dividually, without requiring the other parts of the system to be running. For
1058 example, the link between applications and SDK API services must be testable
1059 without running an automotive domain; the link between SDK API services and
1060 the inter-domain interface at the boundary of the CE domain must be testable
1061 without running an automotive domain; etc.

1062 See [Testability](#), [New version of AD software](#), [Unsupported AD interfaces](#).

1063 **Testability of protocol parsers and writers**

1064 All protocol parsers and writers in the inter-domain communications system
1065 must be testable individually, using unit tests and test vectors which cover
1066 all facets of the protocol. These tests must include negative tests—checks that
1067 invalid input is correctly rejected. For example, if a protocol requires a certificate
1068 to authenticate a peer, a test must be included which attempts a connection
1069 with different types of invalid certificate.

1070 See [Testability](#), [New version of AD software](#), [Unsupported AD interfaces](#).

1071 **Testability of processes**

1072 The code implementing all processes in the inter-domain communications system
1073 must be testable individually, without having to run each process as a subprocess
1074 in a test harness (because this makes testing slower and error prone). This means
1075 implementing each process as a library, with a well defined and documented API,

1076 and then using that library in a trivial wrapper program which hooks it up to
1077 input and output streams and accepts command line arguments.

1078 See [Testability](#), [New version of AD software](#), [Unsupported AD interfaces](#).

1079 **CE system services separated from transport layer**

1080 There must be a trust boundary between each service on the CE which has access
1081 to the inter-domain communication link, and the service which provides access
1082 to the inter-domain communications link itself. The inter-domain service should
1083 validate that messages from a service are related to that service (for example,
1084 by having a whitelist of types of message which each service can send).

1085 This limits the potential for escalation if service A is exploited —then the at-
1086 tacker can only use the inter-domain service to impersonate A, rather than to
1087 impersonate all services in the CE. It also allows the resource usage of the inter-
1088 domain service to be limited, to limit the impact of a denial of service attack
1089 on it.

1090 See [Malicious CE](#), [Marshalling resource usage](#).

1091 **No dependency on CE specific hardware**

1092 As the CE hardware may be upgraded by a garage at some point, the inter-
1093 domain communications should not depend on specific identifiers in this hard-
1094 ware, such as an embedded cryptographic key. Such keys may be used, but the
1095 AD should accept multiple keys (for example, all keys signed by some overall
1096 key provided by Apertis to all OEMs), rather than only accepting the specific
1097 key from the hardware it was originally run against.

1098 This requirement may also be satisfied by including provisions for updating the
1099 copy of a key in the AD if such a dependency on a specific CE key is a sensible
1100 implementation choice.

1101 See [After market upgrade of a domain](#).

1102 **Immediate error response if service on peer is unavailable**

1103 If a service on the peer has crashed or is unresponsive, but the peer itself (includ-
1104 ing its inter-domain communications link) is still responsive, that peer should
1105 return an error to the other domain, which should propagate it to any caller of
1106 SDK APIs which use the failing service. An error response must be returned,
1107 otherwise the caller will time out.

1108 See [Power cycle independence of domains \(CE down\)](#), [Power cycle independence](#)
1109 [of domains \(AD down, single screen\)](#), [Power cycle independence of domains \(AD](#)
1110 [down, multiple screens\)](#), [Plug-and-play CE device](#)

1111 **Immediate error response if peer is unavailable**

1112 If the peer has crashed, or is not currently connected to the physical inter-
1113 domain communications link (either because it has been unplugged or due to a
1114 fault), the other peer must generate a local error response in the inter-domain
1115 service and return that to any caller of SDK APIs which require inter-domain
1116 communications. An error response must be returned, otherwise the caller will
1117 time out.

1118 See [Power cycle independence of domains \(CE down\)](#), [Power cycle independence](#)
1119 [of domains \(AD down, single screen\)](#), [Power cycle independence of domains \(AD](#)
1120 [down, multiple screens\)](#), [Plug-and-play CE device](#)

1121 **Timeout error response if peer does not respond**

1122 If the peer is unresponsive to a particular inter-domain message, the other peer
1123 must generate a local error response in the inter-domain service and return that
1124 to the caller of the SDK API which required inter-domain communications. An
1125 error response must be returned, otherwise the caller will wait for a response
1126 indefinitely (or have to implement its own timeout logic, which would be redun-
1127 dant).

1128 See [Power cycle independence of domains \(CE down\)](#), [Power cycle independence](#)
1129 [of domains \(AD down, single screen\)](#), [Power cycle independence of domains \(AD](#)
1130 [down, multiple screens\)](#), [Plug-and-play CE device](#)

1131 **All inter-domain communications APIs are asynchronous**

1132 As inter-domain communications may have some latency, or may time out after
1133 a number of seconds, all SDK APIs which require inter-domain communications
1134 must be asynchronous, in the [GLib sense](#)¹⁰: the call must be started, a handler
1135 for its response added to the caller's main loop, and the caller must continue
1136 with other tasks until the response arrives from the other domain.

1137 This encourages UIs to be written to not block on SDK API calls which might
1138 take multiple seconds to complete, as during that time, the UI would not be
1139 redrawn at all, and hence would appear to 'freeze'.

1140 See [Temporary communications problem](#).

1141 **Reconnect to peer as soon as it is available**

1142 If a domain has crashed and restarted, or was disconnected from the inter-
1143 domain communications link and then reconnected, the domain must reconnect
1144 to its peer as soon as the peer can be found on the network. If, for example,
1145 both domains had crashed, this may involve waiting for the peer to connect to
1146 the network itself.

¹⁰<https://developer.gnome.org/gio/stable/GAsyncResult.html>

1147 See [Plug-and-play CE device](#).

1148 **External domain watchdog**

1149 Both domains must be connected to an external watchdog device which will
1150 restart them if they crash and fail to restart themselves.

1151 The watchdog must be external, rather than being the other domain, in case
1152 both domains crash at the same time.

1153 See [Power cycle independence of domains \(CE down\)](#), [Power cycle independence](#)
1154 [of domains \(AD down, single screen\)](#), [Power cycle independence of domains \(AD](#)
1155 [down, multiple screens\)](#).

1156 **Reporting system for malicious applications**

1157 There should exist a trusted path from the application launcher in the CE to
1158 the Apertis store to allow the launcher to provide feedback about applications
1159 which are detected to have done ‘malicious’ things, such as called an SDK API
1160 with parameters which are obviously out of range.

1161 If such a path exists, the inter-domain service in the CE must be able to detect
1162 error responses from the AD which indicate that malicious behaviour has been
1163 detected and rejected, and must be able to forward those notifications to the
1164 reporting system.

1165 See [Feedback for malicious applications](#).

1166 **Ability to disable the consumer-electronics domain**

1167 There must exist a trusted path to a setting in the AD to allow the vehicle
1168 owner to disable the CE because it has been compromised, pending taking the
1169 vehicle to a trusted dealer to install an update.

1170 As well as preventing booting the CE, this must disable all inter-domain com-
1171 munications from within the inter-domain service in the AD.

1172 See [Compromised CE with delayed fix](#).

1173 **Tamper evidence**

1174 If the CE or AD, or communications between them are tampered with by an
1175 attacker, it must be possible for an investigator (who is trusted by and has access
1176 to tools provided by the OEM) to determine that the software or hardware was
1177 modified —although it might not be possible for them to determine *how* it was
1178 modified. This will allow for liability to be attributed in the event of an accident
1179 or warranty claim.

1180 See [Tinkering vehicle owner on the network](#), [Tinkering vehicle owner on the](#)
1181 [boards](#).

1182 **No global keys in vehicles**

1183 The security which protects the inter-domain communication system (including
1184 any trusted boot security) must use unique keys for each vehicle, and must not
1185 have a global key (one which is the same in all vehicles) as a single point of
1186 failure.

1187 This means that if an attacker manages to compromise one vehicle, they must
1188 not be able to learn anything (any keys) which would allow them to compromise
1189 another vehicle with less effort.

1190 See [Tinkering vehicle owner on the network](#), [Tinkering vehicle owner on the](#)
1191 [boards](#).

1192 **Existing inter-domain communication systems**

1193 As this is quite a unique problem, we know of no directly comparable systems.
1194 More generally, this is an instance of a distributed system, and hence similar
1195 in some respects to a number of existing remote procedure call systems or dis-
1196 tributed middleware systems.

1197 If comparisons with specific systems would be beneficial, they can be included
1198 in a future revision of this document.

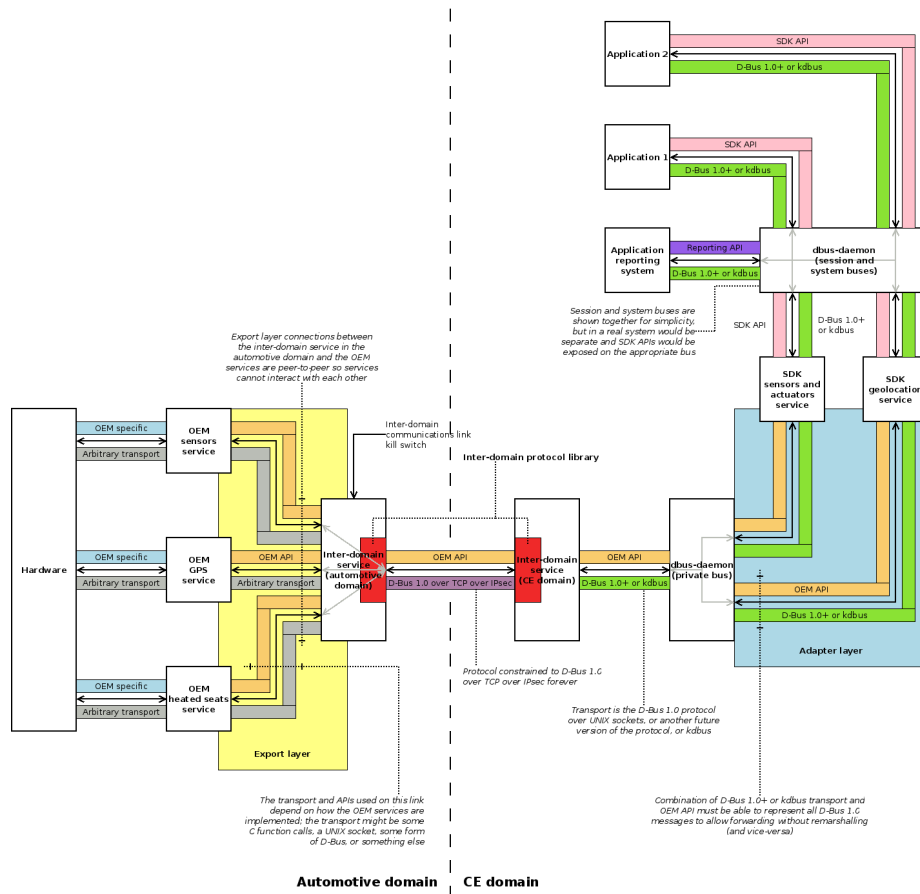
1199 **Open question:** Are there any relevant existing systems to compare against?

1200 **Approach**

1201 Based on the [above research][Existing domain communications system] and
1202 [Requirements](#), we recommend the following approach as an initial sketch of an
1203 inter-domain communication system.

1204 **Overall architecture**

1205 In the following figure, each box represents a process, and hence each connection
1206 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

1224 D-Bus session bus, to be used by applications ([Flow for a given SDK API call](#)).

1225 The export layer and adapter layer provide abstraction of the APIs from the
1226 automotive domain: the export layer converts them from C APIs, QNX message
1227 passing, or however they are implemented in the automotive OS, to a D-Bus API
1228 which is specific to that OEM, but which has stability guarantees through use
1229 of API versioning ([Interaction of the export and adapter layers](#)). The adapter
1230 layer converts from this D-Bus API to the current version of the Apertis SDK
1231 APIs. Both layers are OEM-specific.

1232 The use of the D-Bus protocol throughout the system means that between the
1233 export layer and the adapter layer, message contents do not need to be re-
1234 marshalled —messages only need their headers to be changed before they are
1235 forwarded. This should eliminate a common cause of poor performance (re-mar-
1236 shalling).

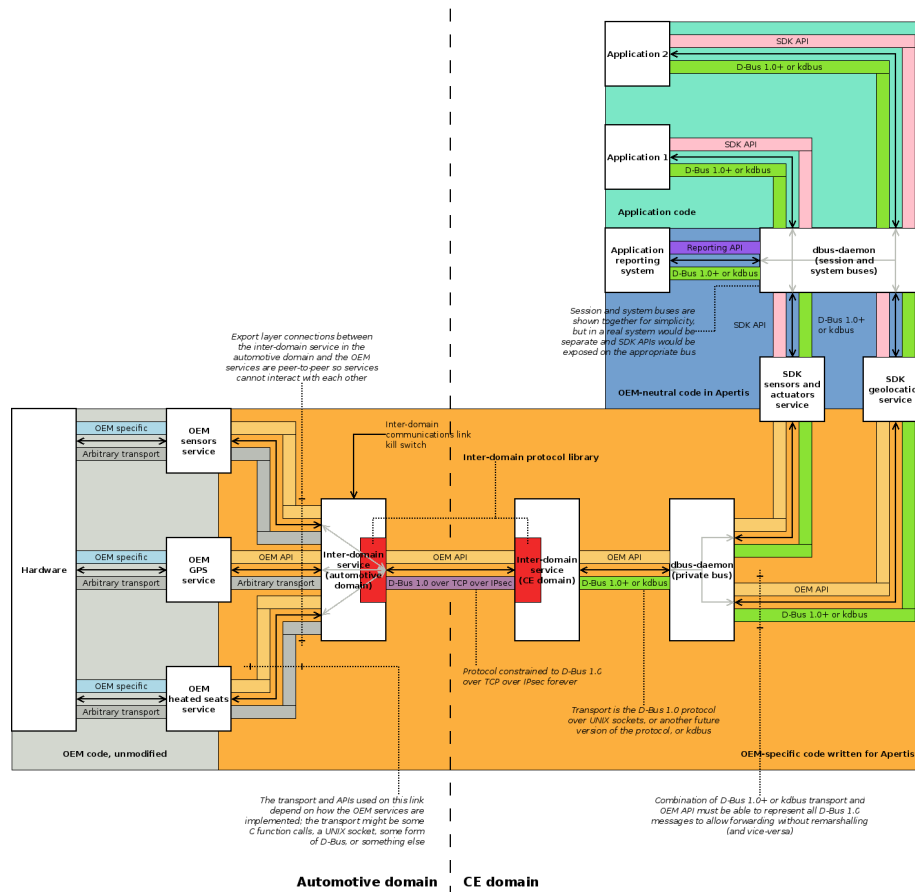
1237 High-bandwidth [Data connections](#) are provided in parallel with the *control con-*
1238 *nection* which runs this D-Bus protocol ([Control protocol](#)). They use TCP,
1239 UDP or Unix sockets, and are opened between the two inter-domain services on
1240 request. Applications and services must define their own protocols for commu-
1241 nicating over these links, which are appropriate to the data being transferred
1242 (for example, audio data or a Bluetooth file transfer).

1243 Authentication, confidentiality and integrity of all inter-domain communications
1244 (the control connection and data connections) are provided by using IPsec as
1245 the bottom layer of the protocol stack ([Encryption](#)). The same protocol stack
1246 is used for all configurations of the two domains (from a standalone CE domain
1247 through to multiple CE domains on a shared network with an automotive do-
1248 main), to ensure that the same code path is used for all configurations and hence
1249 is widely tested ([Configuration designs](#)).

1250 Addressing and discovery of domains, before the initial connection between them,
1251 is provided by IPv6 neighbour discovery ([Traffic control](#)).

1252 Traffic control is implemented in the CE domain using standard Linux kernel
1253 traffic control mechanisms, with the policy specified by the inter-domain ser-
1254 vice (section 8.4). It is applied for the control connection and for each data
1255 connection separately, as they are all separate TCP or UDP connections.

1256 The only exception from the above is [Linux container setup](#) which uses Unix
1257 Domain Sockets as a trusted and reliable bottom transport layer instead of IPsec.
1258 In this case, there is no need for traffic control. Addressing and discovery of
1259 local domains in [Linux container setup](#) is based on common directories created
1260 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.

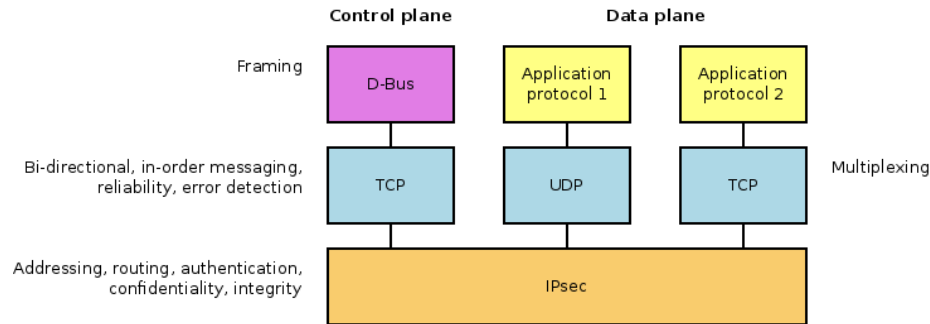
The data plane is used for high bandwidth data, such as video or audio streams, or Wi-Fi, 4G or Bluetooth downloads. The latency requirements are variable, but all connections are high bandwidth. The inter-domain communication system provides a plain stream for each data plane connection, and services 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>

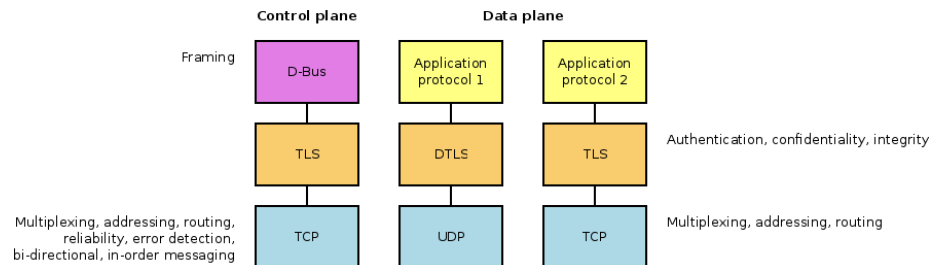
¹²https://en.wikipedia.org/wiki/Transport_Layer_Security



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 protocol¹³ identified a number of problems with it. However, since then, it has been updated by RFC 4301¹⁴, RFC 6040¹⁵ and RFC 7619¹⁶. 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.

¹³https://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>

1333 **Open question:** What is the security of the IPsec protocol in its current (2015)
1334 state?

1335 Performance-wise, TLS requires a handshake for each new connection, which
1336 imposes connection latency of at least one round trip (assuming use of [TLS ses-](#)
1337 [sion resumption](#)¹⁷) for each new connection (on top of other latency such as the
1338 TCP handshake). It is not possible to use a single TLS session and multiplex
1339 connections within it, as this puts the protocol reliability (TCP retransmission)
1340 below the multiplexing in the protocol stack, which makes the multiplexed con-
1341 nection prone to [head of line blocking](#)¹⁸, which seriously impacts performance,
1342 and allows one connection to perform a denial of service attack on all others it
1343 is multiplexed with. IPsec has the advantage of not requiring this handshake
1344 for each connection, which significantly reduces the latency of creating new con-
1345 nections, but does not affect their overall bandwidth once they have reached a
1346 steady state.

1347 **Open question:** What is the performance of TCP and UDP over IPsec, TLS
1348 over TCP and DTLS over UDP on the Apertis reference hardware?

1349 Overall, we recommend using IPsec if it is expected to be supported by all
1350 automotive domain operating systems which will be used with Apertis systems.
1351 Otherwise, if an AD OS might not support IPsec, we recommend using TLS
1352 over TCP and DTLS over UDP for *all* configurations. We do *not* recommend
1353 providing a choice for OEMs between IPsec and TLS, as this doubles the possible
1354 configurations (and hence testing) of a part of the system which is both complex
1355 and security critical.

1356 The remainder of this document assumes that IPsec is chosen. Throughout,
1357 please read ‘IPsec’ as meaning ‘the IPsec protocol stack or the TLS protocol
1358 stack’.

1359 **Configuration designs** The physical links available between the domains dif-
1360 fer between configurations of the domains, as do their properties. For some con-
1361 figurations ([Standalone setup](#), [Basic virtualised setup](#), [Linux container setup](#))
1362 confidentiality and integrity of the inter-domain communications protocol are
1363 not strictly necessary, as the physical link itself cannot be observed by an at-
1364 tacker. However, for the other configurations, these two properties are impor-
1365 tant.

1366 Since the first two configurations are the ones which are typically used for devel-
1367 opment, we suggest implementing confidentiality and integrity for them anyway,
1368 regardless of the fact it’s not strictly necessary. This avoids the situation where
1369 the code running on production configurations is vastly different from that run-
1370 ning on development configurations. Such a situation often leads to inadequate
1371 testing of the production code.

¹⁷<https://tools.ietf.org/html/rfc5077>

¹⁸https://en.wikipedia.org/wiki/Head-of-line_blocking

1372 This should be weighed against the potential performance gains from eliminating
1373 encryption from those connections, and the potential gains in debuggability
1374 (for the **Standalone setup** and **Linux container setup**) by being able to inspect
1375 network traffic without needing to extract the encryption key.

1376 **Open question:** What trade-off do we want between performance and testa-
1377 bility for the different transport layer configurations?

1378 Standalone setup

1379 IPsec running on a **loopback interface**¹⁹ to a service running in the SDK which
1380 mocks up the inter-domain service running in the AD. The security properties it
1381 provides are technically not needed, as the standalone setup is for development
1382 and is ignored by the security model.

1383 Even though there are only two peers communicating, they will both have and
1384 use a full addressing scheme (**Addressing and peer discovery**).

1385 Basic virtualised setup

1386 A virtio-net connection must be set up in the CE and AD virtual guests, using
1387 a private network containing those two peers. If the AD cannot be modified to
1388 enable a virtio-net connection, a normal virtualised Ethernet connection must
1389 be used.

1390 Virtio-net is the name of the KVM paravirtualised network driver
1391 (<http://www.linux-kvm.org/page/Virtio>). Similar paravirtualised
1392 drivers exist for most hypervisors; so an appropriate one for the
1393 hypervisor should be used. For simplicity, this document will use
1394 ‘virtio-net’ to refer to them all.

1395 In either case, the transport layer will use IPsec between the two. The security
1396 properties it provides are technically not needed for a virtualised configuration,
1397 as the security model guarantees that the hypervisor maintains confidentiality
1398 and integrity of the connection.

1399 Even though there are only two peers on the network, they will both have and
1400 use a full addressing scheme (**Addressing and peer discovery**).

1401 Separate CPUs setup

1402 A normal Ethernet connection must be used to connect the AD and CE on a
1403 private network. IPsec will be used over this Ethernet link, providing all the
1404 necessary transport layer properties.

1405 Even though there are only two peers on the network, they will both have and
1406 use a full addressing scheme, described below.

1407 Separate boards setup

1408 Same as for the separate CPUs setup.

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

1409 Separate boards setup with other devices

1410 Same as for the separate CPUs setup.

1411 Multiple CE domains setup

1412 Same as for the separate CPUs setup. Each domain's address must be unique,
1413 and the use of addressing in this configuration becomes important.

1414 Linux container setup

1415 The communication is based on Unix Domain Sockets (UDS) shared between
1416 the counterpart domains; this means that a common directory must be shared
1417 for each pair of communicating domains. This directory must be writable by at
1418 least one container, such that its gateway layer or adapter layer can create the
1419 named unix domain socket file and listen on it, and must be readable on the
1420 other container, which will connect to the shared named unix domain socket
1421 file. The dedicated shared directory for communication may support space
1422 limits for writing and inodes creation, for example: dedicated `tmpfs` mount or
1423 `btrfs` subvolume quota, to prevent denials of service due to filesystem space
1424 exhaustion.

1425 The container manager is responsible for the actions below when each container
1426 is started or stopped:

- 1427 • a shared storage space (a size-constrained `tmpfs` mount or `btrfs` subvol-
1428 ume) must be defined for each pair of containers on the host system, for
1429 instance `${IDC_HOST_DIR}/automotive-connectivity` for the link connecting
1430 the `automotive` and `connectivity` domains
- 1431 • the shared storage must be mounted by the container manager with
1432 read/write permissions on the first domain of the pair, for instance as
1433 `${IDC_DIR}/connectivity` in the `automotive` domain
- 1434 • the same shared storage must be mounted by the container manager
1435 with read permissions on the second domain of the pair, for instance as
1436 `${IDC_DIR}/automotive` in the `connectivity` domain
- 1437 • when the container is stopped, the shared storage and mounts associated
1438 with the container must be unmounted

1439 The variables `${IDC_HOST_DIR}` and `${IDC_DIR}` mentioned above represent the
1440 paths where the shared spaces are mapped on the host and containers filesys-
1441 tems respectively. By default, both variables `${IDC_HOST_DIR}` and `${IDC_DIR}`
1442 are defined in a common manner as `/var/lib/idc/`. OEM or developer's setup
1443 may require to redefine these paths for the customised environment.

1444 Addressing and peer discovery

1445 **Network addressing and peer discovery** Each domain will be identified
1446 by its IPv6 address, and domains will be discovered using the IPv6 protocol's

1447 secure [neighbour discovery](#)²⁰ protocol. As domains do not need to be human-
1448 addressable (indeed, the users of the vehicle need never know that it has multiple
1449 domains running in it), there is no need to use DNS or mDNS for addressing.

1450 The neighbour discovery protocol includes a feature called neighbour unreach-
1451 ability detection, which should be used as one method of determining that one
1452 of the domains has crashed. When a domain crashes, the other domain should
1453 poll for its existence on the network at a constant frequency (for example, at
1454 2Hz) until it reappears at the same address as before. This frequency of polling
1455 is a trade-off between not flooding the network with connectivity checks, but
1456 also detecting reappearance of the domain rapidly.

1457 When reconnecting to a restarted domain, the normal authentication process
1458 should be followed, as if both domains were starting up normally. There is no
1459 state to restore for the inter-domain link itself but, for example, SDK services
1460 may wish to re-query the automotive domain for the current vehicle state after
1461 reconnecting. They should do this after receiving an error response from the
1462 AD for an inter-domain communication which indicated that the other domain
1463 had crashed. Such behaviour is up to the implementers of each SDK service,
1464 and is not specified in this design.

1465 **Container-based addressing and peer discovery** Each container must be
1466 assigned an unique name on the filesystem to be used as domain identifier for
1467 addressing and peer discovery purposes.

1468 The `${IDC_DIR}` directory in the container contains a directory entry for each
1469 associated domain to be connected through the inter-domain communication
1470 mechanism. As described in [Linux container setup](#), the container manager is
1471 responsible for mounting a dedicated shared space to host the socket for the
1472 container pairs.

1473 The name of mount point for the shared directory in the container should be the
1474 same as the name of counterpart peer. For example, to connect an `automotive`
1475 and a `connectivity` domain, the shared space must be mounted in the `automotive`
1476 container on the `${IDC_DIR}/connectivity/` path and must be mounted in the
1477 `connectivity` container on the `${IDC_DIR}/automotive/` path.

1478 On startup, each container in the pair must try to `unlink()` any stale file in the
1479 shared spaces and then create a Unix Domain Socket named `socket` there. Since
1480 the shared directory is mounted with write permissions only on a single domain,
1481 the `unlink()` and `bind()` calls on the unix socket file will fail on the other domain,
1482 which only has read permissions.

1483 Once it has removed any stale file and successfully created the socket, the first
1484 container in the pair must then `listen()` on it: for instance the `automotive`
1485 domain must listen on the `${IDC_DIR}/connectivity/socket` unix socket. The
1486 second container in the pair must instead wait for the `socket` file to be available

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

1487 and must connect to it as soon it is created: for instance the `connectivity` must
1488 wait for the `${IDC_DIR}/automotive/socket` file to appear and connect to it.

1489 **Encryption** The confidentiality, integrity and authentication of the inter-
1490 domain communications link is provided by IPsec in transport mode for net-
1491 worked setups, and by kernel-provided Unix Domain Sockets on [container-based
1492 setups][Linux container setup].

1493 **Open question:** What more detailed configuration options can we specify for
1494 setting up IPsec? For example, disabling various optional features which are
1495 not needed, to reduce the attack surface. What IKE service should be used?

1496 The system should use an IPsec security policy which drops traffic between
1497 the CE and AD unless IPsec is in use. The security policy should not specify
1498 behaviour for communications with other peers.

1499 Each domain must have an X.509 certificate (essentially, a public and private
1500 key pair), which are used for automatic keying for the IPsec connections. The
1501 certificates installed in the automotive domain must be signed by a certificate
1502 authority (CA) specific to the automotive domain and possibly the OEM. The
1503 certificates installed in the CE domain must be signed by a CA specific to the
1504 CE domain and possibly the OEM.

1505 A domain (automotive or CE) which is in developer mode must use a certificate
1506 which is signed by a developer mode CA, not the production mode CA. This
1507 allows a production mode domain to prevent connections from a developer mode
1508 domain.

1509 See [Appendix: Software versus hardware encryption](#) for a comparison of soft-
1510 ware and hardware encryption.

1511 In order to maintain confidentiality of the connection, the keys for the IPsec
1512 connection must be kept confidential, which means they must be stored in mem-
1513 ory which is not accessible to an attacker who has physical access to the system
1514 (see [Tamper evidence and hardware encryption](#)); or they must be encrypted
1515 under a key which is stored confidentially (a key-encrypting key, KEK). Such a
1516 confidential key store should be provided by the Secure Boot design —if avail-
1517 able, confidentiality of the inter-domain communications can be guaranteed. If
1518 not available, inter-domain communications will not be confidential if an at-
1519 tacker can extract the boot keys for the system and use them to extract the
1520 inter-domain communications keys.

1521 As of February 2016, the Secure Boot design is still forthcoming

1522 See section 8.15 for further discussion of the hardware base for confidentiality
1523 and integrity of the system.

1524 **Open question:** A lot of business logic for control over OEM licencing can
1525 be implemented by the choice of the CA hierarchy used by the inter-domain
1526 communication system. What business logic should be possible to implement?

1527 **Open question:** Consider key control, revocation, protocol obsolescence, and
1528 various extensions for pinning keys and protocols.

1529 **Open question:** What can be done in the automotive domain to reduce the
1530 possibility of exploits like [Heartbleed](#)²¹ affecting the inter-domain communica-
1531 tions link? This is a trade-off between the stability of AD updates (high; rarely
1532 released) and the pace of IPsec and TLS security research and updates and the
1533 need for crypto-agility (fast). Heartbleed was a bug in a bad implementation of
1534 an optional and not-very-useful TLS extension.

1535 **Control protocol** The control protocol provides push and pull method call
1536 semantics and a type system for marshalling method call parameters and return
1537 values—but it does not prescribe a specific set of APIs which it will transport.
1538 It must be flexible in the set of APIs which it transports.

1539 We suggest using D-Bus over TCP as the control protocol, using a private bus
1540 between the automotive domain and the consumer–electronics domain. For mul-
1541 tiple CE domain configurations, each automotive–consumer–electronics domain
1542 pair would have its own private bus.

1543 The transport should be implemented using D-Bus’TCP [socket transport](#)²²
1544 mechanism. Authentication, confidentiality and integrity are provided by the
1545 underlying IPsec connection. D-Bus implements its own datagram framing on
1546 top of the TCP stream.

1547 On this bus, APIs from the automotive domain would be exposed as services;
1548 the CE domain can then call methods on those services, or receive signals from
1549 them.

1550 D-Bus was chosen as it implements the necessary functionality, reuses a lot of
1551 the technologies already in use in Apertis, is stable, and is familiar to Apertis
1552 developers. Note that we suggest D-Bus the *protocol*, not necessarily dbus-
1553 daemon the *message bus daemon* or libdbus the reference *protocol library*. D-Bus
1554 the protocol provides:

- 1555 • Method calls (pull semantics) with exactly one reply, supporting timeouts
- 1556 • Error responses
- 1557 • Signals (push semantics)
- 1558 • Properties
- 1559 • Strong type system
- 1560 • Introspection

1561 There are several important points here: introspection means that the D-Bus
1562 services on the AD can send their API definitions to the CE at runtime if needed,

²¹<https://en.wikipedia.org/wiki/Heartbleed>

²²<http://dbus.freedesktop.org/doc/dbus-specification.html#transports-tcp-sockets>

1563 so that the CE does not have to have access to header files (or similar) from the
1564 AD. It also means the API definition can change without needing to recompile
1565 things—for example, an update to the AD could expose new APIs to the CE
1566 without needing to update header files on the CE. Finally, method calls support
1567 ‘in’ and ‘out’ parameters (multiple return values) which allows for bi-directional
1568 communication in the control protocol.

1569 **Open question:** How should the multiple CE configuration (**Configuration de-**
1570 **signs** interact with D-Bus signals? Can the adapter layer perform the broadcast
1571 to all subscribers?

1572 The D-Bus protocol is stable, and has maintained backwards compatibility with
1573 all previous versions since 2006²³. If changes to the D-Bus protocol are intro-
1574 duced in future, they will be introduced as extensions which are used optionally,
1575 if supported by both peers on the bus. Hence backwards compatibility is main-
1576 tained.

1577 **Data connections** If a service wishes to send high-bandwidth data between
1578 the domains, it must open a new data connection. Data connections are created
1579 on demand, and are subject to traffic control, so the AD may, for example, reject
1580 a connection request or throttle its bandwidth in order to maintain quality of
1581 service for existing connections.

1582 The inter-domain communication protocol provides two types of data connec-
1583 tion: TCP-like and UDP-like. These are implemented as TCP or UDP connec-
1584 tions between the two domains, running over IPsec. IPsec provides the necessary
1585 authentication, confidentiality and integrity of the data; TCP or UDP provide
1586 the multiplexing between connections (see the IPsec protocol stacks figure in
1587 **IPSec versus TLS**). For **Linux container setup** a Unix domain socket is used as
1588 the IDC link; the local kernel provides the needed authentication, confidential-
1589 ity and integrity of the data. Services must implement their own application-
1590 specific protocols on top of the TCP or UDP connection they are provided. For
1591 example, a video service may use a lossy synchronised audio/video protocol over
1592 UDP for sending video data together with synchronised audio; while a down-
1593 load service may use HTTP over TCP for sending downloads between domains.
1594 (See [here][Appendix: Audio and video decoding] for a discussion of options for
1595 implementing video and audio decoding.) Such protocols are not defined as part
1596 of this design—they are the responsibility of the services themselves to design
1597 and implement.

1598 Data connections are opened by sending a request to one of the inter-domain
1599 services (**Protocol library and inter-domain services**), specifying desired charac-
1600 teristics for the connection, such as whether it should be TCP-like or UDP-like,
1601 its bandwidth and latency requirements, etc. The connection will be opened
1602 and a unique identifier and file descriptor for it returned to the requesting ser-
1603 vice. This service must then send the identifier over the control connection so

²³<http://dbus.freedesktop.org/doc/dbus-specification.html#stability>

1604 that the corresponding service in the other domain can request a file descriptor
1605 for the other end of the connection from its inter-domain service.

1606 **Open question:** Could this be simplified by using D-Bus's support for file de-
1607 scriptor passing? D-Bus's TCP transport currently explicitly does not support
1608 file descriptor passing, so implementing it that way without introducing incompatibilities
1609 requires planning.

1610 It is tempting to extend D-Bus's support for file descriptor (FD) passing so that
1611 it operates over TCP to provide these data connections. However, that would
1612 effectively be a fork of the D-Bus protocol, which we do not want to maintain
1613 as part of this system. Secondly, due to the way FD passing works, with the
1614 peer passing an FD to the dbus-daemon and asking for it to be forwarded —this
1615 would mean that the peer (i.e. an SDK or OEM service) has the responsibility
1616 for opening the data connection within the IPsec tunnel, which would be very
1617 complex.

1618 Instead, we recommend a custom API provided by the inter-domain service
1619 which an SDK or OEM service can call to open a new data connection, passing
1620 in the parameters for the connection (such as TCP/UDP, quality of service
1621 requirements, etc.). The inter-domain service would communicate over a private
1622 control API with the other inter-domain service to open and authenticate the
1623 connection at both ends, and return a file descriptor and cryptographic nonce
1624 (securely random value at least 256 bits long) to the original SDK or OEM
1625 service. This service can use that file descriptor as the data connection, and
1626 should pass the nonce over its own control protocol to the corresponding OEM or
1627 SDK service. This service should then pass the nonce to its inter-domain service
1628 and will receive the file descriptor for the other end of the data connection in
1629 reply.

1630 Both inter-domain services should retain their file descriptors (which they have
1631 shared with the OEM and SDK services) for the data connection, so that if the
1632 kill switch (**Disabling the CE domain**) is enabled, they can call shutdown() on
1633 the data connection to forcibly close it.

1634 The inter-domain services must reserve all well-known names starting
1635 with `org.apertis.InterDomain` (for example, `org.apertis.InterDomain1` or
1636 `org.apertis.InterDomain1.DataConnections`), and similarly all D-Bus interface
1637 names. This means they must not allow these names to be used as part of the
1638 OEM API shared between the export and adapter layers (**Interaction of the**
1639 **export and adapter layers**).

1640 A data connection cannot exist without an associated control connection
1641 (though one control connection may be associated with many data connections).
1642 As data connections are opened and controlled through APIs defined on the
1643 inter-domain services, there is no need for standard network-style service

1644 discovery using protocols like [DNS-SD](#)²⁴ or [SSDP](#)²⁵.

1645 **Time synchronization** As a distributed system, the inter-domain services
1646 may require a shared clock across the domains. Time synchronization is critical
1647 to correlate events and this is specially important when playing audio and video
1648 streams, for example. If those streams are decoded on the CE and needs to
1649 played by the AD, the AD and the CE should agree on the meaning of the
1650 timestamps embedded in the streams.

1651 For the synchronization, there are two suitable protocols:

- 1652 • [NTP](#)²⁶ is a well-known protocol to synchronise time among remote sys-
1653 tems. It provides millisecond or sub-millisecond accuracy over the Internet
1654 or local area networks respectively;
- 1655 • [PTP](#)²⁷ provides microsecond or sub-microsecond accuracy and is designed
1656 for local area networks.

1657 In terms of latency calculation, both protocols satisfy the requirements, but we
1658 recommends PTP for the following reasons:

- 1659 • NTP uses hierarchical time sources, whereas PTP has a simpler mas-
1660 ter/slave model. That means any system that is even untrusted domain
1661 in a network is able to be taken by the other CE domain as a NTP source;
- 1662 • PTP supports hardware assisted timestamps to improve accuracy. Un-
1663 der Linux, the PTP hardware clock (PHC) subsystem is used to produce
1664 timestamps on supported network devices.

1665 **Audio streams** To share audio streams [RTP](#)²⁸ and its companion protocol
1666 [RTCP](#)²⁹ are recommended both on networked and container-based setups, for
1667 encoded and decoded streams.

1668 They provide jitter compensation, out-of-sequence handling and synchronization
1669 across multiple different streams.

1670 In particular [multiplexed RTP/RCTP][Appendix: Multiplexing RTP and
1671 RTCP] can be used to multiplex both protocols over the kind of data
1672 connections described above.

1673 **Decoded video streams** A fully decoded video stream consumes large quan-
1674 tities of bandwidth and sharing it between domains using the same approach
1675 used by audio (RTP) can only work for very small resolutions (see [Memory](#)

²⁴https://en.wikipedia.org/wiki/Zero-configuration_networking#DNS-SD

²⁵https://en.wikipedia.org/wiki/Simple_Service_Discovery_Protocol

²⁶https://en.wikipedia.org/wiki/Network_Time_Protocol

²⁷https://en.wikipedia.org/wiki/Precision_Time_Protocol

²⁸https://en.wikipedia.org/wiki/Real-time_Transport_Protocol

²⁹https://en.wikipedia.org/wiki/RTP_Control_Protocol

bandwidth 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 bandwidth 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

³⁰<https://manpages.debian.org/stretch/manpages-dev/link.2.en.html>

³¹https://manpages.debian.org/stretch/manpages-dev/copy_file_range.2.en.html

1715 contents in the most efficient way that the kernel can provide.

1716 A different mechanism can be defined where the sender stores the contents in
1717 a private file and passes a file descriptor pointing to it to the inter-domain
1718 communication subsystem. The receiving side then uses the `copy_file_range()`
1719 syscall to get a copy of the data that cannot be altered by the sender and then
1720 validates the data. On filesystems that supports reflinks, `copy_file_range()` will
1721 automatically use them to provide fast copy-on-write clones of the original file:
1722 this would make the operation nearly-instantaneous regardless of the amount of
1723 data, and would avoid doubling the storage requirements. When reflinks can-
1724 not be used, `copy_file_range()` will do an in-kernel copy, avoiding unnecessary
1725 context-switches over normal user-space copy operations. Such approach can
1726 be used on container-based setups or when a cluster file system is shared across
1727 networked domains. Not many filesystems can handle reflinks, but Btrfs and
1728 the OCFS2 cluster filesystem support them.

1729 On systems set up such that reflinks can be used, this solution is much more
1730 efficient than the alternatives, but imposes constraints on the whole system
1731 that may not be acceptable, such as requiring filesystems that support reflinks
1732 (such as Btrfs or OCFS2) on all the domains and ensuring that the appropriate
1733 shared filesystem mounts are available to SDK services. For this reason, the
1734 socket-based approach is recommended in the general case.

1735 **Data connections API** This section defines the draft for a proposed D-
1736 Bus API that SDK services could use to request the creation of data channels
1737 separated from the control plane connection.

1738 The gateway and adapter layers are responsible for the creation and initialization
1739 of those channels, while other services and applications must not be able to
1740 directly create them.

1741 The gateway and adapter layers use instead file descriptors passing to share the
1742 channel endpoints with the requesting services and applications.

1743 The API drafted here is meant to only provide a very rough guideline for those
1744 implementing any real data channel API and it's not meant to be normative: real
1745 implementations can diverge from the interfaces described here and the actual
1746 API to be used by SDK services must be documented in a separate specification.

```
1747 /* The interface exported by the adapter/gateway to SDK services to initiate channel creation. */
1748 interface org.apertis.InterDomain.DataConnection1 {
1749     /* @id: the app-specific unique token used to to identify and authorize the channel
1750      * @destination: the bus name of the service which should be at the other end of the channel
1751      * @type: the kind of data and the protocol to be used for the data exchange.
1752      *      Use 'audio-rtp' for multiplexed RTP/RFC5761.
1753      * @metadata_in: a dictionary of extra information that can be used to authorize/validate the transfer
1754      * @metadata_out: the @metadata_in dictionary with additional information
1755      * @fd: the file descriptor for the actual data exchange using the protocol specified by @type */
```

```

1756     method CreateChannel (in s id,
1757                             in s destination,
1758                             in s type,
1759                             in a{sv} metadata_in,
1760                             out a{sv} metadata_out,
1761                             out h fd)
1762
1763     /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
1764     *
1765     * If the receiver was not able to validate the channel, the `org.apertis.InterDomain.ChannelError`
1766     * error is raised. */
1767     method CommitChannel(in s id)
1768
1769     /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel() */
1770     method AbortChannel(in s id)
1771
1772     /* @refclk: the reference to the IDC shared clock, in the format of defined
1773     * by the `clksrc` production of RFC7273 for the `ts-refclk:` parameter */
1774     method GetClockReference(out s refclk)
1775 }
1776
1777 /* The interface to be exported by services that can handle incoming channels.
1778 * Domains that do not use a local dbus-daemon can implement a similar mechanism
1779 * with the native IPC system. */
1780 interface org.apertis.InterDomain.DataConnectionClient1 {
1781     /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
1782     * @sender: the bus name of the service which initiated the channel creation
1783     * @type, @metadata_in, @metadata_out: see org.apertis.InterDomain.DataConnection1.CreateChannel()
1784     * @proceed: true if the channel should be set up, false if it should be refused */
1785     method ChannelRequested(in s id,
1786                             in s sender,
1787                             in s type,
1788                             in a{sv} metadata_in,
1789                             out a{sv} metadata_out,
1790                             out b proceed)
1791
1792     /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
1793     * @success: whether the connection has been successfully set up and @fd is usable
1794     * @fd: the file descriptor from which to read the incoming data with the
1795           previously agreed protocol
1796     method ChannelCreated(in s id,
1797                             in b success,
1798                             in h fd)
1799 }
1800
1801 /* The interface private to gateway/adaptor services to cross the domain boundary. */

```

```

1802 interface org.apertis.InterDomain.DataConnectionInternal1 {
1803     /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
1804     * @sender: see org.apertis.InterDomain.DataConnectionClient1.ChannelRequested()
1805     * @destination, @type, @metadata_in, @metadata_out: see org.apertis.InterDomain.DataConnection1.CreateChannel()
1806     * @proceed: see org.apertis.InterDomain.DataConnectionClient1.ChannelRequested()
1807     * @nonce: a one-time value used to authenticate the socket
1808     * @socket_addr: the proto:addr:port string to be used to connect to the remote service
1809     method RequestChannel(in s id,
1810                           in s sender,
1811                           in s destination,
1812                           in s type,
1813                           in a{sv} metadata_in,
1814                           out a{sv} metadata_out,
1815                           out b proceed,
1816                           out s nonce,
1817                           out s socket_addr)
1818
1819     /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
1820     * @sender: see org.apertis.InterDomain.DataConnectionClient1.ChannelRequested()
1821     * @destination: see org.apertis.InterDomain.DataConnection1.CreateChannel()
1822     *
1823     * If the receiver was not able to validate the channel, the `org.apertis.InterDomain.ChannelError`
1824     * error is raised. */
1825     /*
1826     method CommitChannel(in s id,
1827                           in s sender,
1828                           in s destination)
1829
1830     /* @id: see org.apertis.InterDomain.DataConnection1.CreateChannel()
1831     * @sender: see org.apertis.InterDomain.DataConnectionClient1.ChannelRequested()
1832     * @destination: see org.apertis.InterDomain.DataConnection1.CreateChannel()
1833     */
1834     method AbortChannel(in s id,
1835                           in s sender,
1836                           in s destination)
1837 }

```

1838 Data channel API flow example for a media player streaming audio

1839 A possible use-case of the API is a Media Player frontend hosted on the AD
1840 with the backend on the CE. The frontend requests the backend to decode a
1841 specific stream using an application specific API and passing a token with the
1842 request.



```

1846      o ----- Play() -----o-----|-----o-----
1847 ----> o
1848                                     |           o <-- CreateChannel() -- o
1849                                     o <-- RequestChannel() -- o
1850      o <-- ChannelRequested() -- o           |
1851      o -- ChannelRequested() --> o           |
1852      reply                                |
1853                                     o -- RequestChannel() --> o
1854      reply
1855      o <- connect and nonce -- o
1856      o <-- ChannelCreated() ---- o           |           o -- CreateChannel() --> o
1857      |                                     |           reply
1858      o <----- data channel -----
1859 ---- o

```

1860 The Media Player frontend initially calls the application-specific `Play()` method
1861 on its backend, with the IDC system transparently proxying the request across
1862 domains. This call must also carry an application-specific token that will be
1863 used to identify the request during the channel creation procedure.

1864 Once the Media Player backend has gathered some metadata about the stream
1865 to be played, it requests the creation of an `audio-rtp` channel directed to the Me-
1866 dia Player frontend by calling the `org.apertis.InterDomain.DataConnection1.CreateChannel()`
1867 on the local adapter service.

1868 The adapter service will then access the inter-domain link by calling the
1869 `org.apertis.InterDomain.DataConnectionInternal1.RequestChannel()` method of
1870 the remote gateway peer.

1871 The gateway service on the AD notifies the Media Player frontend that a channel
1872 has been requested, passing the request token and other application-specific
1873 metadata. If the token matches and the metadata is acceptable, the Media
1874 Player frontend replies to the gateway service telling it to proceed.

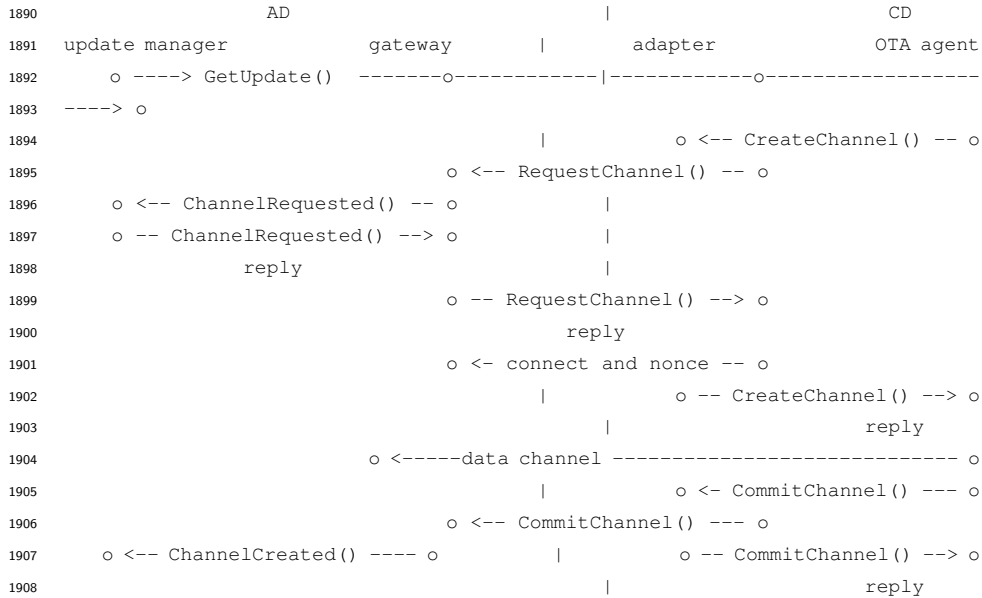
1875 Once the request has been accepted by the destination, the gateway service
1876 creates a listening socket for the requested channel type and returns the infor-
1877 mation needed to connect to it to the remote adapter peer, including a nonce
1878 to authenticate the connection.

1879 As soon as the adapter gets the socket information it connects to it and sends
1880 the nonce over it. On the other side the gateway will read the nonce and if does
1881 not matches it immediately closes the connection.

1882 Once the connection has been set up and the nonce has been successfully shared,
1883 the adapter and gateway services will hand over the file descriptors of the sockets
1884 that have been set up.

1885 **Data channel API flow example for an update manager sharing**
1886 **firmware images** The bulk data transfer API is meant to be useful for

1887 update managers where an agent in the Connectivity Domain fetches firmware
 1888 images from the Internet and shares them with the update manager in the AD
 1889 which has access to the devices to be updated.



1909 The update manager calls the `GetUpdate()` method of the agent, with a token identifying the request. The OTA agent retrieves the metadata of the
 1910 file to be shared, in particular the size and a set of cryptographic hashes.
 1911 With that information, it requests the creation of a bulk-data channel with
 1912 the `org.apertis.InterDomain.DataConnection1.CreateChannel()` method of the local
 1913 adapter service. The OTA agent must specify the `size` parameter and a
 1914 known cryptographic hash such as `sha512` in the `metadata_in` parameter. It must
 1915 then check in the `metadata_out` for the `offset` parameter to figure out if it must
 1916 resume an interrupted download.
 1917

1918 The adapter service accesses the inter-domain link by calling the `org.apertis.InterDomain.DataConnectionInternal`
 1919 method of the remote gateway peer.

1920 The flow is analogous to the one in the [streaming media player case][Data
 1921 channel API flow example for a media player streaming audio] until the point
 1922 where the inter-domain socket is created: while the receiving end of the socket
 1923 in the streaming case is meant to be used by the receiving service, in the bulk
 1924 data case it is used directly by the gateway, which stores the received data in a
 1925 local file.

1926 While it sends data through the socket, the OTA agent is expected to perform
 1927 on-the-fly data validation by computing cryptographic hashes on the streamed
 1928 contents: once it has sent all the data the agent can close the socket and call
 1929 `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 control³² 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.

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.

³²https://en.wikipedia.org/wiki/Network_traffic_control

³³<http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html>

1968 Furthermore, this allows the protocol code to be reused between the inter-
1969 domain service in the automotive domain, and the inter-domain service in the
1970 CE domain.

1971 The main advantage of implementing the protocol as a library is the flexibility
1972 this provides for integrating it into different automotive domain implementa-
1973 tions—it can be integrated into an existing system service (bearing in mind the
1974 suggestion to keep it in a separate trust domain, [Security domains](#)), or could be
1975 used as a stand-alone service daemon.

1976 A reference implementation of such a stand-alone inter-domain service program
1977 should be provided with the protocol library. This should provide the necessary
1978 systemd service file and AppArmor profile to allow itself to be strictly confined
1979 if the automotive domain OS supports this.

1980 As the inter-domain communications protocol uses D-Bus, the protocol library
1981 must contain an implementation of the D-Bus protocol. Note that this is *not*
1982 a D-Bus daemon; it is a D-Bus library, like libdbus or GDBus. See [Appendix:
1983 D-Bus components and licensing](#) for details about the different components in
1984 D-Bus and their licensing.

1985 Apart from its D-Bus library dependency, the protocol library should be de-
1986 signed with minimal dependencies in order to be easily integratable into a va-
1987 riety of automotive domain operating systems (from Linux through to other
1988 Unixes, QNX or Autosar). If the chosen D-Bus library is available as part of
1989 the automotive OS (which is more likely for libdbus than for other D-Bus li-
1990 braries), it could be linked against; otherwise, it could be statically linked into
1991 the protocol library.

1992 libdbus itself is already quite portable, having been known to work on Linux,
1993 Windows, OS X, NetBSD and QNX. It should not be difficult to port to other
1994 POSIX-compliant operating systems.

1995 [Rate limiting on control messages](#) should be implemented in the protocol li-
1996 brary, so that the same functionality is present in both the automotive and CE
1997 domains.

1998 The protocol library should expose the encryption keys for the IPsec connection
1999 used in the inter-domain communications link, including signals for when those
2000 keys change (due to cookie renegotiation on the link). The keys must only be
2001 exposed in development builds of the protocol library. See [Debuggability](#) for
2002 more details.

2003 **Non Linux-based domains**

2004 The suggested implementation uses D-Bus the *protocol*, not necessarily dbus-
2005 daemon the *message bus daemon* or libdbus the *protocol library*.

2006 This means that for inter-domain communications purposes, only the serial-
2007 ization format of D-Bus is used as a well defined RPC protocol. There's no

2008 requirement that domains run `dbus-daemon` or that they use a specific D-Bus
2009 implementation to talk to other domains.

2010 Several implementations of the D-Bus serialization format exists and their use
2011 is strongly encouraged rather than reimplementing the protocol from scratch:

- 2012 • [GDBus](#)³⁴ is a GTK+/GNOME oriented implementation of the D-Bus pro-
2013 tocol in GLib
- 2014 • [QtDBus](#)³⁵ is Qt module that implements the D-Bus protocol
- 2015 • [node-dbus](#)³⁶ is a D-Bus protocol implementation for NodeJS written in
2016 pure JavaScript
- 2017 • [libdbus](#)³⁷ is the reference implementation of the D-Bus protocol
- 2018 • [dbus-sharp](#)³⁸ is a C#/.net/Mono implementation of the D-Bus protocol
- 2019 • [pydbus](#)³⁹ is a python implementation of the D-Bus protocol

2020 On networked setups the D-Bus-based protocol is transported over TCP, relying
2021 on IPSec for authentication, confidentiality and reliability.

2022 If IPSec nor TLS are available, those properties cannot be guaranteed, and thus
2023 such setup is strongly discouraged. In that case every input should be treated
2024 as potentially malicious: the trusted domains must export only a very reduced
2025 set of interfaces, which must be conceived in a way that any kind of misuse does
2026 not lead to harm.

2027 Service discovery

2028 Accordingly to the use of the D-Bus serialization protocol, each service
2029 exported over the inter-domain communication channels is identified
2030 by a well-known name subject [specific constraints](#)⁴⁰, starting with the
2031 reversed DNS domain name of the author of the service (for instance,
2032 `com.collabora.CarOS.ClimateControl1` for a potential service written by [Collab-](#)
2033 [ora](#)⁴¹).

2034 Only one service at a time can own such names on each domain, but the owner-
2035 ship is not tracked across domains and collision may happen due to a transitional
2036 state during an upgrade or other causes: each setup is thus responsible to define
2037 a deterministic collision resolution procedure should two domains export the
2038 same service name.

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>

⁴⁰<https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-bus>

⁴¹<https://collabora.com>

2039 The adapter layer is responsible to inspect on which channel each service is
2040 available. The `NameOwnerChanged` signal⁴² must be used by the adapter layer to
2041 track the availability of services on each connection and to detect when a service
2042 is no longer available or changed ownership (for example because it has been
2043 restarted). The `org.freedesktop.DBus.ListActivatableNames()`⁴³ message can be
2044 used to gather the initial list of available services.

2045 After an upgrade a domain may stop providing a specific service and
2046 another domain may start providing it instead: both the old and new
2047 domains must trigger the `NameOwnerChanged` signal⁴⁴ in response to the
2048 `org.freedesktop.DBus.ReleaseName()`⁴⁵ and `org.freedesktop.DBus.RequestName()`⁴⁶
2049 calls. No specific ordering is required and thus the service may be temporarily
2050 unavailable or the two domains may export the same service name at the same
2051 time: the collision resolution procedure must choose the one on the connection
2052 with the highest priority.

2053 In the simplest case, each domain must be given an unique priority with the
2054 AD having the highest priority. The relative priority between the CE domains
2055 is used to provide deterministic service access when a service name exists on
2056 multiple connections. As a result, the priority list must be static and the priority
2057 of CE domains can be assigned arbitrarily for each specific setup.

2058 When accessing a service name that exists on more than one connection, the
2059 service that exists on the connection with the highest priority must be given
2060 precedence by the adapter layer.

2061 CE domains should not be able to spoof trusted services exported by the AD:
2062 for this reason a static list of services meant to be exported only by the AD
2063 must be defined and the adapter layer must ignore matching services exported
2064 by other connections, even if the service is not currently available on the AD
2065 connection itself.

2066 Particular care must be taken to ensure each domain can be fully booted with-
2067 out blocking on services hosted on other domains, to avoid untracked circular
2068 dependencies.

2069 SDK services must access the above service names through the private bus
2070 instance exported by the adapter layer, which proxies them from all the inter-
2071 domain channels, abstracting the complexities of inter-domain communications.
2072 SDK services are not aware of the fact that the services are hosted on different
2073 domains.

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>

2074 Automotive domain export layer

2075 To integrate the inter-domain communications system into an automotive do-
2076 main operating system, the APIs to be shared must be exported as objects on
2077 the D-Bus connection provided by the inter-domain service. This is done as an
2078 *export layer* in the inter-domain service in the automotive domain, customised
2079 for the OEM and their specific APIs. The export layer could be implemented
2080 as pure C calls from within the same process (no protocol at all), or D-Bus, or
2081 kdbus, or QNX message passing, or something else entirely. If D-Bus bus is
2082 used, a D-Bus daemon would need to be running on the automotive domain;
2083 otherwise, no D-Bus daemon would be needed.

2084 For example, if the automotive domain provides the APIs which are to be ex-
2085 posed over the inter-domain connection as:

- 2086 • C APIs in headers —the inter-domain service would call those APIs directly,
2087 and the export layer would essentially be those C calls;
- 2088 • daemons with UNIX socket connections —the inter-domain service would
2089 connect to those sockets and run whatever protocol is specified by the
2090 daemons, and the export layer would essentially be the socket connections
2091 and protocol implementations;
- 2092 • D-Bus services —the inter-domain service would connect to a D-Bus dae-
2093 mon on the automotive domain and translate the services' D-Bus APIs into
2094 an API to expose on the inter-domain communications link (see below),
2095 and the export layer would be the D-Bus daemon, D-Bus library in the
2096 inter-domain service, and the code to translate between the two D-Bus
2097 APIs.

2098 The APIs must be exported under [well-known names](#)⁴⁷ formatted as reverse-
2099 DNS names owned by the OEM. For example, if the AD operating system
2100 was written by Collabora, APIs would be exported using well-known names
2101 starting with com.collabora, such as com.collabora.CarOS.EngineManagement1
2102 or com.collabora.CarOS.ClimateControl1.

2103 The API formed by these exported D-Bus objects is vendor-specific, but should
2104 maintain its own stability guarantees —for every backwards-incompatible change
2105 to this API, there must be a corresponding update to the CE domain to handle
2106 it. Consequently, we recommend [versioning the exported D-Bus APIs](#)⁴⁸.

2107 APIs which the OEM does not want to make available on the inter-domain
2108 communications link (for example, because they are not able to handle untrusted
2109 data, or are too powerful to expose) must not be exported onto the D-Bus
2110 connection. This effectively forms a whitelist of exposed services.

2111 For each piece of functionality exposed by the AD, suitable safety limits must be
2112 applied ([Safety limits on AD APIs](#)). If the implementation of that functionality

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>

2113 already applies the safety limits, nothing more needs to be done. Otherwise,
2114 the safety limits must be enforced in the interface code which exports that
2115 functionality onto the inter-domain D-Bus connection.

2116 Similarly, for each piece of functionality exposed by the AD, if it fails to respond
2117 to a call by the inter-domain service, the service must return an error to the
2118 CE over the inter-domain D-Bus connection, rather than timing out. This is
2119 especially important in systems where the export layer is a set of C calls —
2120 the implementation must take care to ensure those calls cannot block the inter-
2121 domain service.

2122 If the vendor wants to implement per-API kill switches for services exported
2123 by the automotive domain, these must be implemented in the export layer (see
2124 [Disabling the CE domain](#)).

2125 **Consumer-electronics domain adapter layer**

2126 Paired with the OEM-specific API export code in the automotive domain is an
2127 *adapter layer* in the CE domain. This adapts the API exported by the services
2128 on the automotive domain to the stable SDK APIs used by applications in the
2129 CE domain. The layer has an implementation in each of the SDK services in
2130 the CE domain.

2131 This adapter layer does not have a trust boundary —each part of it lies within
2132 the trust domain of the relevant SDK service.

2133 These adapters connect to a private D-Bus bus, which the inter-domain service
2134 in the CE domain is also connected to. The inter-domain service exports the
2135 OEM APIs from the automotive domain on this bus, and the adapters consume
2136 them.

2137 The private bus could be implemented either by running dbus-daemon with a
2138 custom bus configuration, or by implementing it directly in the inter-domain
2139 service, and having all adapters connect directly to the service. In both cases,
2140 the trust boundary between the adapters (within the trust domains of the SDK
2141 services) and the inter-domain service are enforced.

2142 **Interaction of the export and adapter layers**

2143 The interaction between the export and adapter layers is important in main-
2144 taining compatibility between different versions of the AD and CE as they are
2145 upgraded separately. The CE is typically upgraded much more frequently than
2146 the AD. Both are customised to the OEM.

2147 **Initial deployment** The OEM develops both layers, and stabilises an initial
2148 version of their inter-domain API, using a version number (for example, 1). The
2149 export layer exports objects from the automotive domain, and the adapter layer
2150 imports those same objects. There may be functionality exposed on the objects

2151 which the SDK APIs currently do not support—in which case, the adapter layer
2152 ignores that functionality.

2153 **CE is upgraded, AD remains unchanged** A new release of Apertis is
2154 made, which expands the SDK APIs to support more functionality. The OEM
2155 integrates this release of Apertis and updates their adapter layer to tie the new
2156 SDK APIs to previously-unused objects from the inter-domain link.

2157 The version number of the inter-domain API remains at 1.

2158 **AD is upgraded, CE remains unchanged** The automotive domain OS
2159 is upgraded, and more vehicle functionality becomes available to expose on the
2160 inter-domain connection. The OEM chooses to expose most of this functionality
2161 using the inter-domain service. For some objects, this results in no API changes.
2162 For other objects, it results in new methods being added, but no old ones are
2163 changed. For some objects, it results in some old methods being removed or
2164 their semantics changed. For these objects, the OEM now exports *two* interfaces
2165 on the inter-domain service: one at version 1, exporting the old API; and one
2166 at version 2, exporting the new API. The version number of other inter-domain
2167 APIs remains at 1.

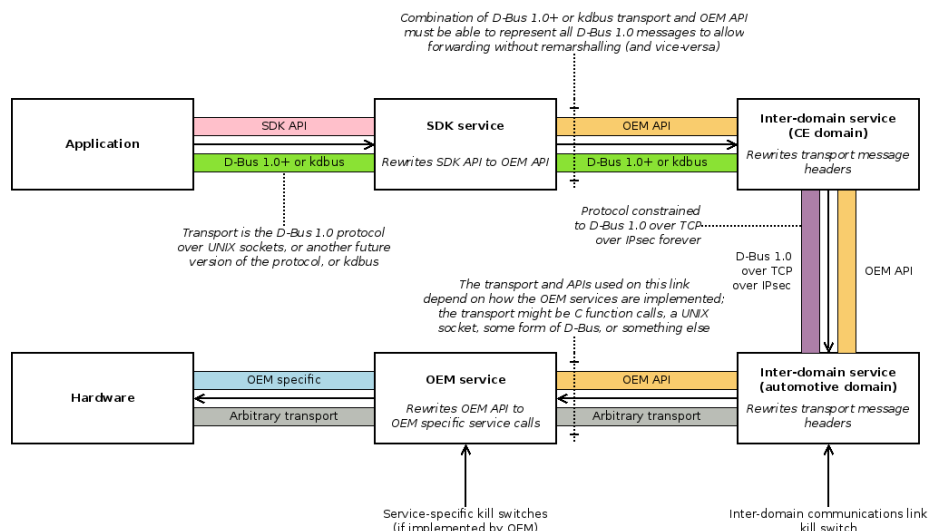
2168 The CE domain software remains unchanged, which means it continues to use
2169 the version 1 APIs. This continues to work because all objects on the inter-
2170 domain API continue to export version 1 APIs (in addition to some version 2
2171 APIs).

2172 **CE is upgraded again** The next time the CE domain is upgraded, its adapter
2173 layer can be modified by the OEM to use the new version 2 APIs for some of
2174 the services. If this updated version of the CE domain is guaranteed to only
2175 be used with new versions of the AD, the adapter layer can drop support for
2176 version 1 APIs. If the updated CE domain may be used with old versions of the
2177 AD, it must support version 1 and version 2 (or just version 1) APIs, and use
2178 whichever it prefers.

2179 **Flow for a given SDK API call**

2180 In the following figure, particular attention should be paid to the restrictions on
2181 the protocols in use for each link. For the links between the application and the
2182 inter-domain service in the CE domain, any version of the D-Bus protocol can be
2183 used, including kdbus or another future version. This depends only on the dbus-
2184 daemon and D-Bus libraries available in the CE domain. For the link between
2185 the two inter-domain services, the protocol must always be at least D-Bus 1.0
2186 over TCP over IPsec. If both peers support a later version of the protocol,
2187 they may use it—but both must always support D-Bus 1.0 over TCP over
2188 IPsec. For the link between the inter-domain service in the automotive domain
2189 and the OEM service, whatever protocol the OEM finds most appropriate for

2190 implementing their export layer should be used. This could be pure C calls
 2191 from within the same process (no protocol at all), or D-Bus, or kdbus, or QNX
 2192 message passing, or something else entirely.



2193

2194 Apertis IDC message flow, following a message being sent from ap-
 2195 plication to hardware; the message flow is the same in reverse for
 2196 message replies from the hardware

2197 Trusted path to the AD

2198 Providing a trusted input and output path between the user and the automo-
 2199 tive domain is out of scope for this design —it is a problem to be solved by
 2200 the graphics sharing and input handling designs. However, it is worth noting
 2201 that the solution must not involve communication (unauthenticated, or authen-
 2202 ticated via the CE domain) over the inter-domain link. If it did, a compromised
 2203 CE domain could be used to forge this communication and gain control of the
 2204 trusted path to the AD —which likely results in a large privilege escalation.

2205 A trusted path should be implemented by direct communication between the
 2206 input and output devices and the automotive domain, or mediating such com-
 2207 munication through the hypervisor, which is trusted.

2208 Developer mode

2209 In order to support connecting the CE domain from an SDK on a developer's
 2210 laptop to the automotive domain in a development vehicle, the 'separate boards
 2211 setup with other devices' configuration must be used, with the CE domain and
 2212 the automotive domain connected to the developer's network (which might have
 2213 other devices on it).

2214 In order to allow the SDK to connect, the vehicle must be in a ‘developer mode’
2215 . This is because the CE domain is entirely untrusted when it is provided by
2216 the SDK, because the developer may choose to disable security features in it
2217 (indeed, they may be working on those security features).

2218 **Open question:** What cryptography should be used to implement this authen-
2219 tication, and the division of trust between development and production devices?
2220 A likely solution is to only have the AD accept the CE connection if it connects
2221 with a ‘production’key signed by the vehicle OEM.

2222 Mock SDK implementation

2223 In order to allow applications to be developed against the Apertis SDK, imple-
2224 mentations of all the SDK APIs need to be provided as part of the official SDK
2225 virtual machine distribution. These implementations need to be fully featured,
2226 otherwise application developers cannot develop against the unimplemented fea-
2227 tures.

2228 There are two implementation options:

- 2229 1. Have an Apertis SDK adapter layer which provides the mock implemen-
2230 tations, and which does not use an inter-domain service or mock up any
2231 of the automotive domain.
- 2232 2. Write the mock implementations as stand-alone services which are log-
2233 ically part of the automotive domain (even though there is no domain
2234 separation in the SDK). Expose these services on the inter-domain link
2235 using an Apertis SDK export layer; and adapt the services to the actual
2236 SDK APIs using an Apertis SDK adapter layer.
2237 The inter-domain services would be running in the same domain (the
2238 SDK) and would communicate over a loopback TCP socket (see **Stan-**
2239 **dalone setup**).

2240 Option #1 has a much simpler implementation, but option #2 means that the
2241 inter-domain communications code paths are tested by all application develop-
2242 ers. Similarly, option #1 introduces the possibility for behavioural differences
2243 between the mock adapter layer and the production inter-domain communica-
2244 tion system, which could affect how application developers write their applica-
2245 tions; option #2 reduces the potential for that considerably.

2246 As option #2 uses the inter-domain service in the CE domain, it also allows for
2247 the possibility of connecting the CE domain to a different automotive domain
2248 —rather than the mock one provided by the SDK, a developer could connect to
2249 the automotive domain in a development vehicle (**Developer mode**).

2250 Hence, our recommendation is for option #2.

2251 Debuggability

2252 The debuggability of the inter-domain communications link is important for
2253 many reasons, from integrating two domains to bringing up a new automotive
2254 domain (with its export and adapter layers) to developing a new SDK API.

2255 Referring to the figure in [Overall architecture](#), debugging of:

- 2256 • *applications and the SDK services* happens using normal tools and meth-
2257 ods described in the [Debug and Logging design](#)⁴⁹;
- 2258 • *communications between the dbus-daemon (private bus) and inter-domain*
2259 *service (CE domain)* happens using normal D-Bus monitoring tools (such
2260 as [Bustle](#)⁵⁰ or [dbus-monitor](#)⁵¹), though this requires the developer to gain
2261 access to the private bus's socket;
- 2262 • *communications between the inter-domain services* happens using a special
2263 debug option in the services (see below);
- 2264 • *the export layer and OEM services* happens using tools and methods spe-
2265 cific to how the OEM has implemented the export layer.

2266 If possible, all debugging should happen on the SDK side, in the adapter layer
2267 or above, as this allows the greatest flexibility in debugging techniques —none of
2268 the communications at that level are encrypted, so are accessible to a developer
2269 user with the appropriate elevated permissions.

2270 If the connection between the inter-domain services (the TCP/IPsec link be-
2271 tween domains) needs to be debugged, it can be complex, as any debugging
2272 tool needs to be able to decrypt the IPsec encryption. Wireshark is [able to do](#)
2273 [this](#)⁵², if given the encryption key in use by the IPsec connection. This key may
2274 change over the lifetime of a connection (as the connection cookie is refreshed),
2275 and hence needs to be exported dynamically by the inter-domain service. In
2276 order to allow debugging both ends of the connection, it should be implemented
2277 in the protocol library ([Protocol library and inter-domain services](#)). In the CE
2278 domain, it should be exposed as a D-Bus interface on the private bus which is
2279 part of the adapter layer. This limits its access to developers who have access
2280 to that bus.

```
2281 Interface org.apertis.InterDomainConnection.Debug1 {  
2282     /* Mapping from IKEv1 initiator cookie to encryption key. */  
2283     readonly property a{ss} Ike1Keys;  
2284     /* Mapping from IKEv2 tuple of (initiator SPI, responder SPI) to tuple  
2285      * of (SK_ei, SK_er, encryption algorithm, SK_ai, SK_ar, integrity  
2286      * algorithm). Algorithms are enumerated types, with values to be
```

⁴⁹<https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application/debug-and-logging/>

⁵⁰<http://willthompson.co.uk/bustle/>

⁵¹<http://dbus.freedesktop.org/doc/dbus-monitor.1.html>

⁵²<https://ask.wireshark.org/questions/12019/how-can-i-decrypt-ikev1-and-or-esp-packets>

```

2287     * documented by the implementation. Other parameters are provided as
2288     * hexadecimal strings to allow for varying key lengths. */
2289     readonly property a((ss)(ssssussu)) Ike2Keys;
2290 }

```

2291 A new Lua plugin⁵³ in Wireshark could connect to this interface and listen for
 2292 signals of updates to the connection's keys, and use those to update Wireshark's
 2293 IKE decryption table. Wireshark is the suggested debugging tool to use, as it
 2294 is a mature network analysis tool which is well suited to analysing the protocols
 2295 being sent over the inter-domain connection.

2296 In the automotive domain, the key information provided by the protocol library
 2297 should be exposed in a manner which best fits the debugging infrastructure and
 2298 tools available for the automotive operating system.

2299 In both domains, this interface must only be exposed in developer builds of the
 2300 inter-domain services. It must not be available in production, even to a user with
 2301 elevated privileges. To expose it would allow all inter-domain communications
 2302 to be decrypted.

2303 External watchdog

2304 There must be an external watchdog system which watches both the automotive
 2305 and consumer-electronics domains, and which restarts either of them if they
 2306 crash and fail to restart themselves.

2307 In order to prevent one compromised domain from preventing a restart of the
 2308 other domain (a denial of service attack), each domain must only be able to send
 2309 heartbeats to its own watchdog, and not the watchdog of the other domain.

2310 The implementation of the watchdog depends on the configuration:

- 2311 • Standalone setup: No watchdog is necessary, as the configuration is not
 2312 safety critical.
- 2313 • Basic virtualised setup: The watchdog should be a software component in
 2314 the hypervisor, exposed as virtualised watchdog hardware in the guests.
- 2315 • Separate CPUs setup: A hardware watchdog on the board should be used,
 2316 connected to both domains. As an exception to the general principle that
 2317 the CE domain should not be allowed to access hardware, it must be able
 2318 to access its own watchdog (and must not be able to access the automotive
 2319 domain's watchdog).
- 2320 • Separate boards setup: A hardware watchdog on each board should be
 2321 used, connected to the domain on that board.
- 2322 • Separate boards setup with other devices: Same as the separate boards
 2323 setup.

⁵³<https://ask.wireshark.org/questions/44562/update-decryption-table-from-lua>

- 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.

⁵⁴https://en.wikipedia.org/wiki/Security_seal

⁵⁵https://en.wikipedia.org/wiki/Trusted_Platform_Module

2360 **Disabling the CE domain**

2361 The automotive domain must be able to disable the power supply to the CE
2362 domain (or otherwise prevent it from booting), and must be able to prevent
2363 inter-domain communications at the same time.

2364 Preventing inter-domain communications should be implemented by having the
2365 automotive domain inter-domain service read a 'kill switch' setting. If this is set,
2366 it should close any open inter-domain communication links, and refuse to accept
2367 new ones while the setting is still set.

2368 Preventing the CE domain from booting can be done in a variety of ways,
2369 depending on the hardware functionality available. For example, it could be
2370 done by controlling a solid-state relay on the CE domain's power supply. Or,
2371 if the CE domain implements secure boot, the boot process could require the
2372 automotive domain to decrypt part of the CE domain bootloader using a key
2373 known only to the automotive domain —if the kill switch is set, this key would
2374 be unavailable.

2375 **Open question:** What hardware provisions are available for controlling the
2376 power supply or boot process of the CE domain? How should this integrate
2377 with the secure boot design?

2378 The kill switch is intentionally kept simple, controlling whether *all* inter-domain
2379 communications are enabled or disabled, and providing no finer granularity.
2380 This is intended to make it completely robust —if support were added for selec-
2381 tively killing some of the control APIs or data connections on the inter-domain
2382 communications link, but not others, there would be much greater scope for
2383 bugs in the kill switch which could be exploited to circumvent it.

2384 If the OEM wants to provide finer grained kill switches for different APIs in
2385 the automotive domain, they must implement them as part of those services, or
2386 as part of the export layer which connects those services to the inter-domain
2387 service.

2388 **Reporting malicious applications**

2389 There are three options for reporting malicious behaviour of applications to the
2390 Apertis store:

- 2391 1. Report from the inter-domain service in the automotive domain, based on
2392 error responses from the OEM APIs.
- 2393 2. Report from the inter-domain service in the CE domain, based on error
2394 responses from the automotive domain.
- 2395 3. Report from the SDK API adapter layers, based on error responses from
2396 the automotive domain.

2397 They are presented in decreasing order of reliability, and increasing order of
2398 helpfulness.

2399 Option #1 is reliable (an attacker can only prevent a detected malicious action
2400 from being reported by compromising the automotive domain), but not helpful
2401 (the automotive domain does not have contextual information about the access,
2402 such as the application bundle which originally made the request —bundle iden-
2403 tifiers cannot be sent across the inter-domain link as that would mean partially
2404 defining the OEM APIs). This option has the additional disadvantage that it
2405 requires the AD to communicate directly with the Apertis store without going
2406 via the CE, which likely means the AD is on the Internet and could potentially
2407 be compromised by a Heartbleed-style vulnerability in a communication path
2408 that was intended to be secure. Options #2 and #3 do not have this disadvan-
2409 tage, because in those options it is the CE that needs to communicate on the
2410 Internet.

2411 Option #3 is unreliable (an attacker can prevent a detected malicious action
2412 from being reported by compromising that SDK service in the CE domain),
2413 but most helpful (the CE domain knows all contextual information about the
2414 access, including the application bundle identifier, parameters sent to the SDK
2415 API by the application, and the output of the adapter layer which was sent to
2416 the inter-domain link).

2417 We recommend option #3 as it is the most helpful, and we believe that the
2418 additional contextual information it provides outweighs the potential loss of
2419 reports from most severely compromised vehicles. This is one part of many
2420 which contribute to the security of the system.

2421 An alternative would be to implement two or all of the options, and leave it up
2422 to the Apertis store software to combine or deduplicate the reports.

2423 Suggested roadmap

2424 One the design has been reviewed, it can be compared to the existing state of
2425 the inter-domain communication system, and a roadmap produced for how to
2426 reconcile the differences (if there are any).

2427 **Open question:** How does this design compare to the existing state of the
2428 inter-domain communication system?

2429 Requirements

2430 **Open question:** Once the design is finalised a little more, it can be related
2431 back to the requirements to ensure they are all satisfied.

2432 Open questions

- 2433 • **Existing inter-domain communication systems:** Are there any relevant
2434 existing systems to compare against?
- 2435 • **IPSec versus TLS:** What is the security of the IPsec protocol in its current
2436 (2015) state?

- 2437 • **IPSec versus TLS:** What is the performance of TCP and UDP over IPsec,
2438 TLS over TCP and DTLS over UDP on the Apertis reference hardware?
- 2439 • **Configuration designs:** What trade-off do we want between performance
2440 and testability for the different transport layer configurations?
- 2441 • **Configuration designs:** What more detailed configuration options can we
2442 specify for setting up IPsec? For example, disabling various optional fea-
2443 tures which are not needed, to reduce the attack surface. What IKE
2444 service should be used?
- 2445 • **Configuration designs:** A lot of business logic for control over OEM li-
2446 cencing can be implemented by the choice of the CA hierarchy used by
2447 the inter-domain communication system. What business logic should be
2448 possible to implement?
- 2449 • **Configuration designs:** Consider key control, revocation, protocol obsoles-
2450 cence, and various extensions for pinning keys and protocols.
- 2451 • **Configuration designs:** What can be done in the automotive domain to
2452 reduce the possibility of exploits like Heartbleed affecting the inter-domain
2453 communications link? This is a trade-off between the stability of AD
2454 updates (high; rarely released) and the pace of IPsec and TLS security
2455 research and updates and the need for crypto-agility (fast). Heartbleed
2456 was a bug in a bad implementation of an optional and not-very-useful TLS
2457 extension.
- 2458 • **Control protocol:** How should the multiple CE configuration (section 8.3.2)
2459 interact with D-Bus signals? Can the adapter layer perform the broadcast
2460 to all subscribers?
- 2461 • **Developer mode:** What cryptography should be used to implement this
2462 authentication, and the division of trust between development and pro-
2463 duction devices? A likely solution is to only have the AD accept the
2464 CE connection if it connects with a 'production'key signed by the vehicle
2465 OEM.
- 2466 • **Disabling the CE domain:** What hardware provisions are available for
2467 controlling the power supply or boot process of the CE domain? How
2468 should this integrate with the secure boot design?
- 2469 • **Suggested roadmap:** How does this design compare to the existing state
2470 of the inter-domain communication system?
- 2471 • **Requirements:** Once the design is finalised a little more, it can be related
2472 back to the requirements to ensure they are all satisfied.

2473 Summary of recommendations

2474 **Open question:** Once the design is finalised a little more, and a suggested
2475 roadmap has been produced (**Suggested roadmap**), it can be summarised here.

2476 Appendix: D-Bus components and licensing

2477 The terminology around D-Bus can sometimes be confusing; here are some
2478 details of its components and their licensing.

- 2479 • *D-Bus* is a [protocol](#)⁵⁶ which defines an on-the-wire format for marshalling
2480 and passing messages between peers, a type system for structuring those
2481 messages, an authentication protocol for connecting peers, a set of trans-
2482 ports for sending messages over different underlying connection media, and
2483 a series of high-level APIs for implementing common API design patterns
2484 such as properties and object enumeration. It has a reference implemen-
2485 tation (libdbus and dbus-daemon), but these are by no means the only
2486 implementations. The protocol has had full backwards compatibility since
2487 [2006](#)⁵⁷.
- 2488 • A *D-Bus daemon* (for example: dbus-daemon, kdbus) is a process which
2489 arbitrates communication between D-Bus peers, implementing multicast
2490 communications (such as signals) without requiring all peers to connect to
2491 each other. Different D-Bus daemons have different performance charac-
2492 teristics and licensing. For example, kdbus runs in the kernel to improve
2493 performance by reducing context switching overhead, at the cost of some
2494 features; dbus-daemon runs in user space with more overhead, but is still
2495 quite performant.
- 2496 • A *D-Bus library* (for example: libdbus, GDBus) is a set of code which
2497 implements the D-Bus protocol for one peer, converting high-level D-Bus
2498 API calls into on-the-wire messages to send to another peer or a D-Bus
2499 daemon to send to other peers. Different D-Bus libraries have different
2500 performance characteristics and licensing.

2501 Licensing

- 2502 • The D-Bus Specification is freely licensed and has no restrictions on who
2503 may implement it or how those implementations are licensed.
- 2504 • libdbus and dbus-daemon are both licensed under your choice of the
2505 [AFLv2.1](#)⁵⁸, or the [GPLv2](#)⁵⁹ (or later versions).
 - 2506 – Hence, if the AFL license is chosen, libdbus and dbus-daemon may
2507 be used in non-open-source products.
- 2508 • GDBus is part of GLib, and hence is licensed under the [LGPLv2.0](#)⁶⁰ (or
2509 later versions).

⁵⁶<http://dbus.freedesktop.org/doc/dbus-specification.html>

⁵⁷<http://dbus.freedesktop.org/doc/dbus-specification.html#stability>

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+>

2510 Appendix: D-Bus performance

2511 libdbus and dbus-daemon are reasonably performant, having been used in vari-
2512 ous low-resource products (such as mobile phones) over the years. There have
2513 not been any quantitative evaluations of their performance in terms of latency
2514 or memory usage recently, but some have been done in⁶¹ the⁶² past⁶³.

2515 As indicative numbers *only*, D-Bus (using [dbus-python](#)⁶⁴ and dbus-daemon, not
2516 kdbus) gives performance of roughly:

- 2517 • 20,000 messages per second throughput
- 2518 • 130MB per second bandwidth
- 2519 • 0.1s end-to-end latency between peers for a given message
 - 2520 – This is likely an overestimate, as ping-pong tests written in C have
 - 2521 given latency of 200µs
- 2522 • 2.5MB memory footprint (RSS) for dbus-daemon in a desktop configura-
2523 tion
 - 2524 – So this could likely be reduced if needed —the amount of message
 - 2525 buffering dbus-daemon provides is configurable

2526 Note that these numbers are from performance evaluations on various versions of
2527 dbus-daemon, so should be considered indicative of an order of magnitude only.
2528 As with all performance measurements, accurate values can only be measured
2529 on the target system in the target configuration.

2530 The most commonly accepted disadvantage of using D-Bus with dbus-daemon
2531 is the end-to-end latency needed to send a message from one peer, through the
2532 kernel, to the dbus-daemon, then through the kernel again, to the receiving
2533 peer. This can be reduced by using kdbus, which halves the number of context
2534 switches needed by implementing the D-Bus daemon in [kernel space](#)⁶⁵. However,
2535 kdbus has not yet been accepted into the upstream kernel, and (as of February
2536 2016) there is some concern that this might not happen due to kernel politics.
2537 It can be integrated into distributions as a kernel module, although it relies on a
2538 few features only available in kernel version 4.0 or newer. This means it should
2539 be straightforward to integrate in the CE, but potentially not in the AD (and
2540 certainly not if the AD doesn't run Linux —in such cases, dbus-daemon can be
2541 used).

2542 Overall, the performance of a D-Bus API depends strongly on the API design.
2543 Good [D-Bus API design] eliminates redundant round trips (which have a high

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/>

⁶⁵<http://www.freedesktop.org/wiki/Software/systemd/kdbus/>

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 handles⁶⁶.

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 set⁶⁷, CLMUL⁶⁸ or the ARM cryptography extensions⁶⁹. 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 devices 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 FIPS⁷²-certified, but not all. GnuTLS has been FIPS certified in various devices, but is not

⁶⁶<http://dbus.freedesktop.org/doc/dbus-specification.html#idp9446907251>

⁶⁷https://en.wikipedia.org/wiki/AES_instruction_set

⁶⁸https://en.wikipedia.org/wiki/CLMUL_instruction_set

⁶⁹<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0514g/index.html>

⁷⁰https://en.wikipedia.org/wiki/Secure_cryptoprocessor

⁷¹https://en.wikipedia.org/wiki/Hardware_security_module

⁷²https://en.wikipedia.org/wiki/FIPS_140-2

2577 [routinely certified](#)⁷³. OpenSSL is not routinely certified, but provides a
2578 OpenSSL FIPS Object Module which *is* [certified](#)⁷⁴ as a drop-in replace-
2579 ment for OpenSSL, provided that it's used unmodified. The Linux kernel's
2580 IPsec support has been certified in Red Hat Enterprise Linux 6, but is
2581 not [routinely certified](#)⁷⁵.

- 2582 • Cheaper than hardware.
- 2583 • Provides the possibility of upgrading to use different encryption algorithms
2584 in future.
- 2585 • Possible to check the software implementation for backdoors, although it's
2586 a lot of work. Some of this work is being done by [other users of open](#)
2587 [source encryption software](#)⁷⁶.

2588 **Software encryption (with encryption acceleration instructions)**

- 2589 • Same advantages and disadvantages as software encryption without en-
2590 cryption acceleration instructions, except that the use of acceleration gives
2591 a higher encryption bandwidth (on the order of a factor of 10 improve-
2592 ment).
- 2593 • Same software interface as without acceleration.
- 2594 • Both TLS and IPsec provide various cipher suite options, at least some of
2595 which would benefit from hardware acceleration —both use [AES-GCM](#)⁷⁷
2596 for data encryption, which benefits from AES instructions.

2597 **Secure cryptoprocessor**

- 2598 • Higher encryption bandwidth.
- 2599 • Reduced attack surface area, as keys and in-progress encryption values are
2600 handled within the encryption hardware, rather than in general memory,
2601 and hence cannot be accessed by an attacker with physical access. Keys
2602 may still leave the cryptoprocessor, which gives some attack surface.
- 2603 • Typical secure cryptoprocessors have tamper evidence features in the hard-
2604 ware.
- 2605 • Typically hardware is FIPS-certified.
- 2606 • More expensive than software.

⁷³http://www.gnutls.org/manual/html_node/Certification.html

⁷⁴<https://www.openssl.org/docs/fips.html>

⁷⁵https://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/>

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

- 2607 • Provides a limited set of encryption algorithms, with no option to upgrade
2608 them once it's fixed in silicon.
- 2609 • No possibility to audit the hardware implementation to check for back-
2610 doors, so you have to trust that the hardware vendor has not been secretly
2611 required to provide a backdoor by some government.
- 2612 • Typical cryptoprocessors originate from mobile or embedded networking
2613 hardware, both of which need to support TLS, and hence cryptoprocessors
2614 typically have support for AES, DES, 3DES and SHA. This is sufficient
2615 for accelerating the common cipher suites in TLS and IPsec.
- 2616 • Have to be supported by the Linux kernel crypto API (`/dev/crypto`) in
2617 order to be usable from software.

2618 Hardware security module

- 2619 • Highest encryption bandwidth.
- 2620 • Minimal attack surface area, with keys never leaving the HSM.
- 2621 • All hardware is tamper-proof and tamper-evident, and typically can de-
2622 stroy stored keys automatically if tampering is detected.
- 2623 • Hardware is almost universally FIPS-certified.
- 2624 • Most expensive.
- 2625 • Provides a range of encryption algorithms, but with no option to upgrade
2626 them.
- 2627 • No possibility to audit the hardware implementation to check for back-
2628 doors, so you have to trust that the hardware vendor has not been secretly
2629 required to provide a backdoor by some government.
- 2630 • Some modules can handle encryption of network streams transparently,
2631 taking a plaintext network stream as input and handling all TLS or IPsec
2632 operations for it with peers.

2633 Conclusion

2634 According to [one evaluation](#)⁷⁸, using encryption acceleration instructions should
2635 reduce the number of cycles per byte for AES encryption from 28 to 3.5. Assum-
2636 ing the inter-domain connection is being used to transmit a HD video at 250kB·
2637 s⁻¹, that means encryption requires 7MHz of CPU compute without acceleration,
2638 and 875kHz with it. Performing symmetric encryption on a data stream doesn't
2639 significantly increase the required memory bandwidth compared to copying the
2640 stream around without encryption.

⁷⁸https://en.wikipedia.org/wiki/AES_instruction_set#Performance

Hence, overall, if we assume a peak bandwidth requirement on the inter-domain communications link on the order of $250\text{kB}\cdot\text{s}^{-1}$ 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](https://en.wikipedia.org/wiki/AES67)⁷⁹: The AES67 standard combines PTP and RTP using PTP clock source signalling ([RFC7273](https://tools.ietf.org/html/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](http://www.videoservicesforum.org/)⁸¹ (VFS). The TR-03 standard is similar to AES67 in terms of using PTP for clock synchronization, but it extends AES67 to cover other kinds of uncompressed streams, including video and metadata.
- [AVB](https://en.wikipedia.org/wiki/Audio_Video_Bridging)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 bandwidth 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

⁷⁹<https://en.wikipedia.org/wiki/AES67>

⁸⁰<https://tools.ietf.org/html/rfc7273>

⁸¹<http://www.videoservicesforum.org/>

⁸²https://en.wikipedia.org/wiki/Audio_Video_Bridging

2668 Note that VFS TR-03 has no explicit open source implementation, but as it
 2669 combines RTP for transport and PTP for clock synchronization, it is generally
 2670 supported by GStreamer.

2671 **Appendix: Multiplexing RTP and RTCP**

2672 RTP requires the RTP Control Protocol (RTCP) to exchange control packets
 2673 and timing information such as latency and QoS. Usually RTP and RTCP use
 2674 two different channels on different network ports, but it is also possible to use
 2675 a single port for both protocols using the [RFC 5761](https://tools.ietf.org/html/rfc5761)⁸³ standard, supported by
 2676 the GStreamer `funnel` element.

2677 The following diagram shows how a RFC 5761 pipeline can be set up in
 2678 GStreamer:

```

2679 /-----\      /-----\      /-----\      /-----\      /-
2680 -----\
2681 | audio | == | audio | == | rtpbin | = rtp = | rtp payloader | = rtp = |      | /-
2682 -----\
2683 | source |      | convert |      |      |      | \-----
2684 /          | funnel | == | udp sink |
2685 \-----/      \-----/      |      | ===== rtcp = |      | \-
2686 -----/
2687                                \-----/
  
```

2688 **Appendix: Audio and video decoding**

2689 As a system which handles a lot of multimedia, deciding where to perform audio
 2690 and video decoding is important. There are two major considerations:

- 2691 • minimising the amount of raw communications bandwidth which is needed
 2692 to transmit audio or video data between the domains; and
- 2693 • ensuring that an exploit does not give access to arbitrary memory from
 2694 either domain (especially not the automotive domain).

2695 The discussion below refers to video encoding and decoding, but the same con-
 2696 siderations apply equally well to audio.

2697 Software encoding is a large CPU burden, and introduces quality loss into
 2698 videos —so decoding and re-encoding videos in one domain to check their well-
 2699 formedness is not a viable option. If decoding is being performed, the decoded
 2700 output might as well be used in that form, rather than being re-encoded to be
 2701 sent to the other domain.

2702 In order to avoid spending a lot of CPU time and CPU-memory bandwidth on
 2703 video decoding, it should be performed by hardware. However, this hardware
 2704 does not necessarily have to be in the domain where the encoded video originates.

⁸³<https://tools.ietf.org/html/rfc5761>

2705 For example, it is entirely possible for videos to be sent from the CE to be
2706 decoded in the AD.

2707 The original designs which were discussed in combination with the GPU video
2708 sharing design planned to create a GStreamer plugin in the CE which treats the
2709 AD as a hardware video decoder which accepts encoded video, decodes it, and
2710 returns a handle which can be passed to the GL scene being output by the CE,
2711 via a GL extension (similar to [EXT_image_dma_buf_import](#)⁸⁴). This is the
2712 same model as used for ‘normal’ hardware decoders, and ensures that decoded
2713 video data remains within the AD, rather than being sent back over the inter-
2714 domain communications link (which would incur a very high bandwidth cost,
2715 which for uncompressed 1080p video in YUV 422 format at 60fps amounts to
2716 $16 \text{ bits/pixel} \times (1920 \times 1080) \text{ pixels/frame} \times 60 \text{ frames/s} = 1898 \text{ Mbit/s} = 237$
2717 MB/s).

2718 Regarding security, a hardware decoder is typically a [DMA](#)⁸⁵-capable peripheral
2719 which means that, unless constrained by an [IOMMU](#)⁸⁶, it can access all areas
2720 of physical memory. The threat here is that a malicious or corrupt video could
2721 trigger the decoder into reading or writing to areas of memory which it shouldn’t,
2722 which could allow it to overwrite parts of the (hypervisor) operating system or
2723 running applications. This concern exists regardless of which domain is driving
2724 the decoder. We highly recommend that hardware is chosen which uses an
2725 IOMMU to restrict the access a video decoder has to physical memory.

2726 Note that the same security threat applies to the GPU, which has direct access
2727 to physical memory (if shared with the CPU —some systems use dedicated
2728 memory for the GPU, in which case the issue isn’t present). GPUs have a much
2729 larger attack surface, as they have to handle complex GL commands which are
2730 provided from untrusted sources, such as WebGL.

2731 We recommend investigating the hardening and security applied to video de-
2732 coders on the particular hardware platforms in use, but there is not much which
2733 can be done by software to improve their security if it is lacking —the perfor-
2734 mance cost is too high.

2735 Memory bandwidth usage on the i.MX6 Sabrelite

2736 This section refers to some benchmarks evaluating the available memory band-
2737 width on the i.MX6 Sabrelite platform used in the reference hardware for Apertis.
2738 This data is very system dependent, but the order of magnitude should provide
2739 a general guide for evaluating approaches.

2740 The [iMX6 memory bandwidth usage benchmark](#)⁸⁷ describes some tools that

⁸⁴https://www.khronos.org/registry/egl/extensions/EXT/EGL_EXT_image_dma_buf_import.txt

⁸⁵https://en.wikipedia.org/wiki/Direct_memory_access

⁸⁶https://en.wikipedia.org/wiki/Input-output_memory_management_unit

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

2741 can be used to measure how memory is used, and reports that a [1080p @ 60fps](#)
2742 [loopback pipeline](#)⁸⁸ using GStreamer requires up to 1744.46 MB/s of memory
2743 bandwidth.

2744 Another useful benchmark is the one evaluating [the cost of memory copies](#)⁸⁹
2745 done with the `memcpy()` function. The effective usable memory bandwidth mea-
2746 sured with this test amounts to roughly 800 MB/s.

2747 Security Vulnerabilities in GStreamer

2748 To list vulnerabilities by type we can refer to the statistics available from the
2749 [CVE](#)⁹⁰ data source.

2750 According to the [CVE Details](#)⁹¹ website, a third party that provides summaries
2751 of CVE vulnerabilities, GStreamer had [total 17 vulnerabilities](#)⁹² since 2009.

2752 Examining the DoS and Code Execution vulnerability types, the statistics
2753 showed different characteristics for demuxers and decoders. There have been
2754 12 DoS vulnerabilities affecting demuxers, but only one issue could lead to
2755 Code Execution. For decoders, all the the 5 DoS issues which were found can
2756 be escalated to Code Execution as well.

2757 This report indicates that demuxers might have a smaller attack surface than de-
2758 coders from the arbitrary code execution viewpoint. However, it is also possible
2759 to have a security hole similar to [Video or audio decoder bugs](#).

2760 Both demuxing and possibly even decoding in the CE can help to mitigate the
2761 described attacks. If the CE is responsible of demuxing the AD does not need
2762 to deal with content detection and container formats, and the CE provides a
2763 kind of partial verification of the stream even without decoding it.

2764 Decoding in the CE poses some challenges in terms of bandwidth, as the amount
2765 of data generated by fully decoded video streams is very high. It's not going to
2766 be a viable solution on ethernet-based setups, and advanced zero-copy mecha-
2767 nisms to transfer frames are recommended on single board setups (virtualised
2768 or container-based).

⁸⁸https://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>