Intra-Vehicle Information Security Framework

Hagai Bar-El

Rev. 1.2

Abstract

This paper presents an internal information security services framework for vehicular environments. The framework consists of a logical *toolbox* — a set of logical modules that are installed in a variety of embodiments (e.g., controllers) and which provide security functionality that vehicular applications require. The framework also includes several *enablers*, which are higher-level security functions that are integrated into vehicular applications. These enablers use the aforementioned tools to provide for many typical use-cases, such as secure logging, secure code update, and secure feature activation.

The purpose of the toolbox is to provide some of the common security functions at the highest effective abstraction level, and to implement these functions securely in well suited embodiments. This detachment of security functions from the applications that use them shall allow developers to develop secure applications without requiring extensive security know-how, as well as to reduce the attack surface of their applications.

1 Introduction

1.1 The Motivation

Vehicles are increasingly utilizing, and becoming reliant on, information technologies. Digital technologies of signal and information processing are becoming common in luxury cars today, and are foreseen to become even more common in additional vehicles during the coming years. Information technologies are common in car multimedia devices today, as well as in GPS navigation devices. Some car models also deploy digital circuits that collect data that is later used for maintenance and for troubleshooting. Information systems technologies are planned to be introduced to other aspects of the vehicle's functionality. For example, vehicles are planned to communicate with each other and with roadside infrastructure components for various purposes ranging from safety to commerce.

As soon as information technologies are involved in the processing of assets of any kind, security risks arise. Information technology components in vehicles need not only be extremely reliable and bug-free, but they also need to be extremely resistant to malicious attacks. The magnitude of assets that may be involved in a vehicular information technology system, as well as the fact that the owner of the vehicle, who naturally has unlimited physical access to it, may in some cases form part of the threat, make protection of the assets involved even more complex. Concerns of 'cyber-terrorism', as well as the fact that the conformance of a vehicle with its perceived behaviour attributes directly to the life-safety of its passengers, further magnifies the risks, and thus the level of protection required.

The risks involved, along with the fact that a vehicle has a long typical lifetime (in information technology terms), implies that information security assurance has to be done properly, and hopefully — done properly from the start.

This paper presents an intra-vehicular information security framework. A discussion of the motivation for deploying information security facilities in a vehicle, beyond what is written above, is strictly outside the scope of this document. This paper assumes that the reader appreciates the need for information security-assuring functions in a vehicle. Accordingly, the paper hereafter focuses on the 'how' rather than on the 'why'.

1.2 Internal vs. External Security Aspects

There are essentially two major domains in which information security concerns need to be addressed: *inter-vehicle* and *intra-vehicle*, or: *external* and *internal* domains, respectively. These domains, as explained below, consider the vehicle information system as a single entity, although in practice it is likely to consist of many detached components.

The *external domain* involves the connectivity of the vehicular components with entities that are external to the vehicle. This domain deals with the possible attack vectors that involve exploitation of *protocol flaws*, either at the network layer or at the application layer. This domain disregards local attack vectors that involve access which is not by the known and approved communication interfaces with entities outside the vehicle. An example of an attack that is addressed as part of the *external domain* of security is an attack that uses a protocol flaw to introduce fake hazard messages that appear to come from a roadside beacon, while they actually come from an attacker's home network; an attacker who prefers traffic to not be routed through his neighborhood. Also

covered by this domain are DoS[1] attacks over the network interfaces.

The *internal domain* of information security considers the robustness of the in-vehicle components against local attacks. Local attacks are such that subject the attacked components to actions that are not (only) in the form of communication to and from their external ports. Among these attacks are physical attacks against the various embodiments of the information technology components (both invasive and non-invasive), and attacks on the interfaces between trusted components within the domain of the vehicle. API attacks also fall within this domain, as long as they involve exploiting interfaces between components *inside* the vehicle.

Distinction as described above is clearly not the only way to view external and internal aspects of security. For example, when treating each component of the vehicular system as an independent entity, attacks against it by other in-vehicle components through its API may be considered as external attacks; whereas internal attacks would only consist of attacks against the physical implementation of the component. Nevertheless, our approach of seeing the entire vehicle as a single entity, composed of a collection of mutually-trusting units that trust each other but not their connectivity, allows us to classify security considerations in a more pragmatic way, by allowing a clearer separation of responsibilities between an intra-vehicle framework and outward-facing applications.

1.3 What We Aim to Provide

Both the *internal domain* and the *external domain* of information security concerns have to be addressed. In this paper, which describes an intra-vehicle information security framework, we focus on the *internal domain*.

Security of the *external domain* relies on the protocols used by the vehicle to communicate with external entities. These protocols range from the network-level protocols (such as TCP/IP), to application-specific protocols for particular C2I and C2C[2] interactions. No substantive security functionality can be introduced that is not directed by the exact specification of these protocols. Since agreement on these protocols is required for interoperability, these protocols are defined and adopted based on strong market forces. Unfortunately, the author at the time of writing is not positioned to determine the design of these protocols and their selection. On the other hand, securing internal aspects of the vehicular system involves design decisions that are within the scope of a vehicle manufacturer to make. Consequently, in the *internal domain* it is more likely for the author's contribution to make a difference.

In this paper, we will attempt to help securing the vehicular information system from the *internal domain* perspective, by introducing a *toolbox*. This toolbox is aimed at providing several discrete functions ('tools'), publishing well-defined APIs, in a way that relieves client applications from the need to address some of the security concerns involved with the applications, to the extent that it is feasible (see Section 2.1 for a discussion of the goals of the toolbox.) The tools provide functions that also enable securing the client-side modules of C2C and C2I systems, from the *internal domain* perspective, as will become evident from reading their descriptions below.

The architecture presented in this paper is part of a larger effort made to develop a complete intra-vehicle information system security solution, which consists of:

- low level enablers (the toolbox), which provide core security functionality that is specific for vehicular environments;
- embedded applications that utilize components of the toolbox to provide for common vehicular use-cases; and
- the API required for third party providers to use the functionality offered by the toolbox for other applications as well.

This paper presents some of the components of the security toolbox (in Section 3), and a few applications that utilize these components (in Section 4).

Lastly, this paper discusses a design that is to a large extent still "work in progress". The reader is encouraged to treat implementation details, such as proposed protocols and structures, with due care.

2 Design Goals

2.1 Toolbox Design Goals

Our goal for the design of the toolbox, which is detailed in Section 3 (pp. 3), is to provide a set of security functions that are specifically tailored for a vehicular information system. The purpose of the toolbox is not just to offload security-related efforts from the developers of vehicular applications, but mainly it is to reduce the need for security assurance (and possibly — compliance) for as many components as possible, and to the extent feasible, while (and as a result of) decreasing the attack surface of the applications.

The toolbox differs from typical "crypto-modules" by the types of functions it provides. These functions are not typical low-level cryptographic functions such as the ones provided, e.g., by PKCS#11 [11] or by smartcards. Rather, these are higher-level functions that are tailored to vehicular systems. The provision of higher-level functions sacrifices being general-purpose in return for providing better vertical coverage in a given environment.

[1] Denial of Service
[2] "Car to Infrastructure" and "Car to Car", respectively.

The tools that form the toolbox are not as useful as other cryptographic modules for non-vehicular environments, but in vehicular environments they provide a higher level of abstraction, implying better coverage and more effective offloading of security functionality.[3]

The toolbox is aimed at allowing vendors to produce secure applications without having to deal (as extensively) with developing core security functionality as part of the applications. When defining the functions that the toolbox provides, we sought the set of highest common denominators of security functions that are used by a typical set of vehicular security-aware applications. For example, if many vehicular applications need to store data objects securely, we prefer to provide a tool that securely stores opaque data objects over providing a general-purpose encryption service that the application shall use to facilitate its own secure storage. For more information on this example see the *Secure Registry Tool* in Section 3.5 (pp. 7).

By offloading (any) functionality from the client application, we reduce its development cost; but there is much more to it. We *aim* towards the asymptotically-reachable (but never completely reachable) situation in which proper integration with the tools of the toolbox is the only security-related design effort the vendor of an application needs to take to make his application secure. Obviously, this target to its fullest extent is unreachable. Not only that tools for all purposes cannot possibly be conceived, but *some* security considerations (derived from possible security lapses) will ordinarily sneak into *every* design of a security-related application. Even an application that processes secure data using the most secure and comprehensive tools, might inadvertently leak data that *it* processes, due to security flaws.

Notwithstanding, by striving to find the highest common denominators of security-related functionality, that is, by designing the toolbox to carry out as much of the security-related functionality as possible, and by employing all security practices in the implementation of this toolbox, we effectively reduce the attack surface of the client application; a worthwhile goal in its own.

A secondary design goal that dictated the selection of tools is the preference to implement security functions that also involve interoperability between controllers in the vehicle. Whereas some of the tools are incorporated directly into the controllers that use them, there are other tools that are hosted by external embodiments, and that serve a plurality of controllers. In such cases, the advantage of having the tool is not limited to the offloading of security concerns from the application, but also include the provision of new features that rely on interaction between multiple controllers, made possible through the tool. For example, by providing a tool for centralized secure storage, as described in Section 3.5 (pp. 7), rather than having each controller store its own data securely, we obtain an important benefit: the ability of controllers to *share* data securely among them.

[3]As a side note, higher levels of abstraction for cryptographic services also imply easier compliance with restrictions on import, export, and usage of cryptography. This is due to the product being non-general-purpose. Notwithstanding, the addressing of such compliance issues is not a design goal of the toolbox.

2.2 Application Enablers Design Goals

Our goal for the design of the application enablers detailed in Section 4 is to provide a useful set of security applications that utilize components of the toolbox.

When determining the set of applications to provide, we try to foresee as many of the security related applications as possible that vehicle makers may require. This set of applications may be seen as a sample set, demonstrating the usefulness of the toolbox.

Obviously, we design the applications to be as robust as possible against the foreseen and documented types of threats that these applications may be subject to. We design the applications securely from the ground up, beyond basing them on components of the toolbox.

3 Architecture of the Security Toolbox

3.1 Overview

The toolbox components presented in this section are designed to meet the goals described in Section 2.1. Subsection 3.2 briefly describes the hardware cores that are used to implement the functionality of the toolbox components. Subsequent subsections describe some of the particular tools that are offered by the toolbox: key distribution for interconnection between controllers in Section 3.3 (pp. 4), a higher-level secure messaging function in Section 3.4 (pp. 6), a secure data registry in Section 3.5 (pp. 7), a code authentication tool in Section 3.6 (pp. 9), a trusted time tool in Section 3.7 (pp. 10), and a key provisioning tool in Section 3.8 (pp. 13).

3.2 Underlying Core

The 'tools', as referred to in this paper, are logical functions that are implemented by various physical embodiments.

The main embodiment that is used to implement some of the tools, as well as some of the application enablers, is a self-contained security module utilizing hardware encryption engines, a local processor, RAM, ROM, connected mass non-volatile storage, and on-chip non-volatile storage, such as in the form of EEPROM. The embodiment itself is physically protected against physical tampering, and against the exploitation of physical side channel attacks. The ciphers are designed to be resistant to logical and physical side-channel attacks, such

as timing attacks, power analysis attacks, and fault attacks. The processor is used to execute locally-stored instructions that implement the functions consisted by the tools. These functions publish the specified API over the system bus on which the embodiment is installed.

Some of the tools are implemented as much smaller IP cores that are incorporated into existing controllers in the vehicle. These IP cores implement functions that have to be provided by the same controller that runs the application that uses these functions. For example, for the *Code Authentication Tool* of Section 3.6 (pp. 9) to be effective, it must perform its main function of code verification on the same controller that runs the verified code. As another example, when keys are securely delivered to a controller using the *Key Distribution Tool* of Section 3.3 (pp. 4), their delivery must utilize cryptographic assets that are securely maintained on that same destination controller.

Some of the tools incorporate both a 'server-side' that runs on a 'main embodiment' as described above, and a lightweight 'client-side' that runs on the controllers on which the client applications run.

3.3 Key Distribution Tool

3.3.1 Purpose

The first tool that is provided by the toolbox is a key distribution function. The purpose of this function is to allow any two controllers that share a key with a *Master Controller* (such as the one presented as the 'main embodiment' in Section 3.2, playing the 'server-side') to have a session-specific key that is available to both of them and which is neither known to, nor obtainable by, any other component in or out of the vehicle.

The necessity of such a tool is clear: a vehicle may host a large set of controllers (more than 80 electronic control units per vehicle of certain types [3], or between 30 and 300 controllers [9]). At least some of these controllers need to communicate securely with each other, and some of these controllers may arrive from different vendors. Regardless of the type of inter-controller communication, direct and secure communication between two controllers requires pre-shared key material.[4] The objective of the *Key Distribution Tool* is to provide the participating controllers with this shared key material, in runtime and upon need.

The use of keys that are valid for a session, rather than permanent keys, also has its rationalé. It allows to easily and centrally revoke the ability of a controller to communicate with other controllers in a vehicle at any time during the vehicle's (long) lifetime. Also, there is a cryptographic need for short-term session-keys in light of the amount of data that is likely to be involved in inter-controller communication throughout the lifetime of the vehicle.

The *Key Distribution Tool* assumes that every controller that uses it has one 256-bit shared symmetric key with the aforementioned *Master Controller*, where the 'server-side' of the tool is installed. Initial provisioning of this key can be done using the *Provisioning Tool* specified in Section 3.8 (pp. 13). The *Provisioning Tool* allows controllers to be enrolled with their keys either at a manufacturer's facility or later, also over untrusted links.

Once the two controllers obtained the key material using the *Key Distribution Tool*, they may use it to communicate securely. It is up to the client application to actually use the key it received from the tool to encrypt and/or sign data, decrypt and/or verify it, etc. A second tool, the *Secure Messaging Tool* presented in Section 3.4 (pp. 6), is provided for client applications that prefer the message processing to be carried out by the toolbox. The *Secure Messaging Tool* utilizes the *Key Distribution Tool* and implements message security over it.

The *Key Distribution Tool* is a fundamental one and is therefore detailed in the following subsection more extensively than other tools.

3.3.2 Method

The *Key Distribution Tool* carries out a well defined and specified key generation protocol on the *Master Controller* and uses the pre-provisioned key each controller has with the *Master Controller* to deliver the generated keys securely. Keys are delivered upon request of the controllers that need to communicate with other controllers; just before communication, or upon boot.

3.3.2.1 Embodiment The *Key Distribution Tool* is implemented in two parts. One part implements the *Master Controller* (the 'server-side') and is installed on one or more embodiments (for load balancing and redundancy). The other part is smaller in size and is required to be installed as embedded logic on each controller that uses this tool. This second part (the 'client-side') contains the minimal secure storage for storing the single shared key with the *Master Controller*, as well as the simple logic required for the execution of the protocol.

3.3.2.2 Flow Key generation and distribution by the tool is accomplished by steps as described herein and illustrated in Figure 1:

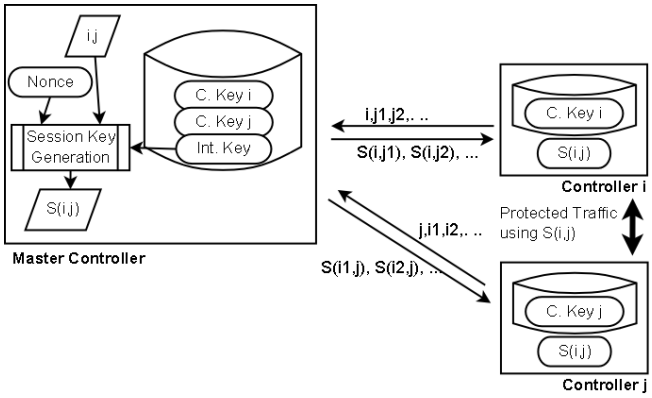
1. When a controller, i, needs to communicate with another controller, j, for the first time since system

[4] It is possible to establish secure communication using public key cryptography, which does not require pre-established key material between the two corresponding controllers. However, the use of PKI (based on RSA, at least) for inter-controller communication is often considered impractical in vehicular environments, due to the short required response time (typically up to 1 ms, according to [9]).

boot, it sends a request to the *Master Controller* for a *shared session-key* $S_{i,j}$.

2. Alternatively, controller i may perform the above request for key $S_{i,j}$ for a given j before the need occurs, e.g., based on a probable future need. For example, controller i may request keys $S_{i,j}$ for any j identifying a controller it may ever need to communicate with, upon system boot.[5]
3. Upon need to communicate with controller j, and given that controller i possesses *shared session-key* $S_{i,j}$, controller i uses it as the shared key with controller j, in accordance with the communication protocol defined by the client application. Controller j, if not yet possessing $S_{i,j}$, requests it from the *Master Controller* in a similar way.
4. Both controller i and controller j may cache the value of *shared session-key* $S_{i,j}$ at most until the system is powered off. The *shared session-key* $S_{i,j}$ for given i and j remains constant throughout the power cycle. If the *Master Controller* is asked for $S_{i,j}$ once again before power-down, the same value will be returned. It is therefore possible for one controller to cache the key while the other has no cache and requests the key each time it is needed. In this case, the controller that does not cache the key shall consider the effect this behavior may have on its response time.

Figure 1: Session-key distribution flow



The request for $S_{i,j}$ is done using the protocol specified in Protocol 1, where C is the requesting controller, identified by i, and M is the *Master Controller*. j denotes the identity of the controller for which $S_{i,j}$ is requested. Multiple values of j may be included in the request, which will lead to multiple key responses. $E(k, x)$ denotes the *authenticated encryption* of message x using symmetric key k. Authenticated encryption may be achieved using the AES block cipher [5] in supporting modes, such as: CCM [4], CWC [7], EAX [1], OCB [10], or Galois/Counter [8]. K_i denotes the permanent *controller key* that is shared between controller i and the *Master Controller* (see assumptions in Section 3.3.1). This key can be initially provisioned using the *Provisioning Tool* specified in Section 3.8 (pp. 13). The value of n is a nonce value generated by C, once per boot.

[5] Although the total number of controllers in a system may reach the hundreds, it is not likely that any single controller i (other than the *Master Controller*) is ever to communicate with all (or even with most) other controllers. Therefore, a scenario in which each controller, upon system boot, requests keys for all controllers it may communicate with, is considered feasible and likely.

Protocol 1 Shared session-key acquisition protocol

$$C \rightarrow M \quad \texttt{'REQ.C.SACQ.V1.00'}, i, n, [j_1, [j_2, \ldots]]$$
$$C \leftarrow M \quad \texttt{'RESP.M.SACQ.V1.00'},\ E\left(K_i, \left(i, n, [(j_1, S_{i,j_1}), [(j_2, S_{i,j_2}), \ldots]]\right)\right)$$

3.3.2.3 Key generation and delivery The *shared session-key* $S_{i,j}$ is computed by the *Master Controller* based on an *internal secret key* of the *Master Controller*, a *session nonce*, and the values i and j.

The acquisition protocol in Protocol 1 is built to allow only controller i to be able to obtain the value of $S_{i,j}$ (for any j), possibly for more than one instance of j. A controller cannot request for arbitrary values of $S_{i,j}$, because the result will always be encrypted using K_i (the *controller key*) of the same i for which $S_{i,j}$ was computed. The value of $S_{i,j}$ is computed in a way that $\forall i, j\ S_{i,j} = S_{j,i}$ so that when controller j makes the request for S, the same value of S will be returned as was returned to i.

To determine the value of $S_{i,j}$, given the values of i, j, s (an optional 256-bit *internal secret key* held by the *Master Controller*), and b (a random 256-bit *session nonce* generated upon boot and kept by the *Master Controller*), the following computation is done.

$$T \leftarrow \begin{cases} H(j,i,s,b) & i > j \\ H(i,j,s,b) & i \leq j \end{cases}$$

$$S_{i,j} \leftarrow \mathrm{MSB}_{256}(T)$$

The function $H(x)$ denotes a secure hash function, running with input x. Input x is represented as a set of comma-separated elements. The hash function is run on a concatenation of structures that contain the listed data elements.

3.3.2.4 Implementation performance considerations The functionality that is carried out by the *Master Controller* (and more so — by the other controllers) for computing S was designed to be simple enough to be implemented by hardware logic, without the need for a processor or for software. It is foreseen that this operation may be done frequently during run-time, and might form a bottleneck on the performance of the controllers in the system if executed too slowly. Moreover,

the functionality of the *Master Controller* is defined in a way that makes it easily distributed among two or more identical *Master Controller*s, without the need for mutual locks or semaphores.

3.4 Secure (Intra-Vehicle) Messaging Tool

3.4.1 Purpose

The *Key Distribution Tool* presented in Section 3.3 allows two controllers to use a shared key each of them has with a *Master Controller* to establish a session-key that is shared between them. Once this key is obtained, the controllers can use it with cryptographic schemes of their choice to secure messages sent between them. As an alternative, the controllers can use the *Secure Messaging Tool* presented in this section to facilitate secure messaging based on the agreed key. The *Secure Messaging Tool* uses the *Key Distribution Tool* to obtain a *shared session-key* for secure messaging. If the vehicle already uses another scheme to disseminate keys to controllers, such as the one presented in [9], the *Secure Messaging Tool* can utilize the resulting key material for secure messaging.

The purpose of the *Secure Messaging Tool* is to take a shared symmetric key between two controllers, and provide for secure message interaction between these two controllers.

It is important to point out that the *Secure Messaging Tool* does not implement the communication sockets. It builds on the existing message (e.g., packet) transmission mechanism that the interacting controllers already utilize. It is not the intent of the *Secure Messaging Tool* to replace the existing intra-vehicle network infrastructure. The tool adds *security assurances* to the existing message-based delivery that is assumed to already be available.

The tool is also agnostic to the contents of the messages it processes. It is designed for adding a security wrap to messages that applications send normally between physically separate controllers.

3.4.2 Method

The tool provides for message security by publishing a library of message processing functions to the controller it is installed on. The tool processes the messages to provide:

- assurance that the messages cannot feasibly be deciphered by any but the intended destination controller/s;
- assurance that the messages cannot feasibly be modified in transit without this fact being detected upon message arrival;
- assurance that messages cannot feasibly be replayed, or otherwise taken out of context, without this fact being detected upon message arrival; and
- assurance of the time at which messages were sent, when used in conjunction with the *Trusted Time Tool* of Section 3.7 (pp. 10).

All the above security properties are provided to all messages that the tool processes. Allowing the ability to switch security features off, or to determine protection strength on a per-message basis, may offer small performance increases, but risks security faults due to misuse. The paradigm of "there is only one mode, and it is 'secure'." is desired whenever the cost of observing it is not too high.[6]

3.4.2.1 Embodiment The tool is embodied as embedded IP in each controller that sends and/or receives secure messages to and/or from other controllers. The tool publishes a library to applications on the controller that allows applications to provide plain-text outgoing messages for encryption, as well as to provide incoming enciphered messages for decryption and verification.

3.4.2.2 Interface The interface that the tool provides is straightforward, and primarily consists of the following functions:

`SetTimeParams`
: If the *Trusted Time Tool* described in Section 3.7 (pp. 10) is also installed, allows it to signal the current time. This function also allows setting of freshness threshold parameters.

`PeerInit`
: Accepts an ID of a destination (or source) controller, and initiates a context against this controller, including the obtainment of a session-key, e.g., a *shared session-key* from the *Key Distribution Tool* specified in Section 3.3 (pp. 4).

`SendMessage`
: Accepts a plain-text message and an ID of a destination controller (that shall match the ID used in `PeerInit`), and returns a *protected message structure*. Optionally, if the *Trusted Time Tool* is also installed, a secure time-stamp can be included in the *protected message structure*.

`ReceiveMessage`
: Accepts a *protected message structure* that was originated by a `SendMessage` call on another controller, and returns the plain-text message, as well as a *message status code* (explained below). If the message also includes a secure time-stamp, the time included in it is also returned.

[6] One of the best examples for this paradigm in action is in *Skype*. The user of *Skype* does not need to do, or even know, anything about security. Yet, as long as his password is safe, he enjoys decent call privacy. (Details on the security scheme used in *Skype* can be found in [2].)

`PeerTerminate`
: Accepts an ID of a peer controller and removes its context. Using `SendMessage` and/or `ReceiveMessage` with this peer controller again will require execution of `PeerInit`.

The *message status code* that is returned by the tool along with each decrypted message holds at least the following indications:

1. The message is valid and contains a time-stamp.
2. The message is valid and does not contain a time-stamp.
3. The message could not be decrypted because it may be destined to another controller.
4. The message was decrypted, but its signature check failed, implying that the message was illegally modified in transit. (In this case, the plain-text message is *not* returned.)
5. The message was decrypted and the signature is valid; however, it may have been replayed or taken out of its context.
6. The message was decrypted, the signature is valid, and so is the context; however, the message has a secure time-stamp attached to it, which indicates that the message is too old to be considered valid.

The existence of a time-stamp is indicated by the *message status code* for successfully decrypted (and valid) messages to make it more intuitive for applications that require time-stamped input to reject messages without a time-stamp.

3.5 Secure Registry Tool

3.5.1 Purpose

The purpose of the *Secure Registry Tool* is to provide a central secure storage facility for all applications running on various controllers. This secure storage keeps any type of information that applications may need to store securely, and assures to a feasible extent that:

- data cannot be obtained by other applications, unless explicitly authorized;
- data cannot be changed by other applications, unless explicitly authorized;
- data can neither be obtained nor changed by physical access to the embodiment of the registry.

There is a vast number of vehicular applications that may require the secure registry service, some of which are described in Section 4 (pp. 14) presenting application enablers.

Data is provided to the *Secure Registry Tool* by applications, and is kept in *registry objects*. Access rights to certain *registry objects* may be granted to applications other than the application that initially stored the objects. Moreover, data can be maintained by entities that are *external* to the vehicle, as long as a local application is available to function as a conduit by interacting with the remote application on one end and with the *Secure Registry Tool* on the other. This feature allows external entities to *own* data in the vehicle, while providing local applications with restricted privileges on this data. Examples for such applications are applications for secure logging and for selective activation of features, presented in Sections 4.2 (pp. 15) and 4.5 (pp. 18), respectively. The *Code Authentication Tool* of Section 3.6 (pp. 9) also utilizes this registry. In the case of the *Code Authentication Tool*, the stored data is managed by the vehicle manufacturer, or by an entity on its behalf, outside the vehicle.

Other than the obvious advantage of having the secure storage facility being implemented once, as part of a secure embodiment that is properly verified, there is a second advantage. It allows applications to authorize other applications to access and possibly manipulate (by various methods described in Section 3.5.2.2) their data objects. Additionally, having a single secure storage implies that all data redundancy mechanisms can be implemented in one place, to provide cheaper and more reliable fail-safe storage.

3.5.2 Method

To implement access by client applications to the secure registry, the *Secure Registry Tool* performs two functions:

- authentication of client applications and establishment of secure sessions; and
- performance of data access operations on *registry objects*, per requests of the client applications.

Each *registry object* is cryptographically protected when stored by the tool in non-volatile storage, and access permissions are associated with it in the form of Access Control Lists (ACLs). Each *registry object* is also identified individually, to allow for applications to refer to it when interacting with the tool. Identification is done using an ID assigned to the *registry object* when it is created.

3.5.2.1 Embodiment The main part (a.k.a. 'server-side') of the *Secure Registry Tool* is implemented on one controller, such as the *Master Controller* discussed in Section 3.3 (pp. 4), or on what is referred to as the 'main embodiment' in Section 3.2 (pp. 3). This controller has to have access to non-volatile storage. This storage can be divided into bulk storage that is not necessarily protected against physical or logical tampering (e.g., flash memory that may be shared with other components), and a much smaller non-volatile

storage medium, which is part of the tamper-protected environment.

The server-side serves client applications that may be installed on other controllers. Each controller that hosts an application which is a client application, must have installed in it a 'client-side' component that provides functions for secure registry access. The client-side component contains the minimal secure storage for storing the single shared key with the server-side, as well as logic required for the execution of the protocol used by the tool.

3.5.2.2 Supported access control The set of permissions that are supported when enforcing access control to *registry object*s is richer than that of most ACLs, in order to support vehicular use-cases, and to do so in a way that positions as many of the security requirements as possible on the *Secure Registry Tool* rather than on the client application. The set of supported permissions is as follows:

`enumerate`
: Permission to know of the existence of the object. It allows the client application to see the object listed in a directory of objects, or to otherwise get indications of the objects existence. Lack of this permission causes unauthorized operations on an existing object to report as `Not Found`, rather than as `Permission Denied`.

`read`
: Permission to obtain the object in its decrypted form.

`write`
: Permission to replace the contents of an object with other contents, while retaining the same object ID and associated metadata. Unlike in other platforms that support ACLs, a `write` permission does not imply a `read` permission. Neither does it imply a `delete` permission.

`delete`
: Permission to remove the object, rendering it unavailable and freeing its ID. This permission implies no other permission.

`append`
: Permission to append data to the object. This permission does not imply a `read` permission, nor a `write` permission.

`increment`
: For numeric objects, permission to increase the value stored in an object by any positive value that does not cause an overflow. This permission implies no other permission.

`manage`
: Permission to perform all operations, and also to set permissions for the object. This permission is granted implicitly to the client application that created the object. In some cases, an application with a `manage` permission on an object may revoke its own `manage` permission. This is allowed to increase security in some scenarios.

3.5.2.3 Flow When a client application, running on any controller on which the 'client-side' is installed, desires to access data stored by the *Secure Registry Tool*, it initiates a session against the 'server-side' of the tool. The purpose of the session establishment is twofold:

1. to determine (securely) the identity of the client application; and
2. to form session-key material that will be used to protect the confidentiality and integrity of the data communicated between the server-side of the *Secure Registry Tool* and the client application.

Authentication between the *Secure Registry Tool* and the client application is carried out upon establishment of a session, using a shared key between each application and the tool. Key material may be provisioned using the *Provisioning Tool*, described in Section 3.8 (pp. 13). The keys may be equal to the keys that are provisioned for the *Key Distribution Tool* of Section 3.3.

A session between a component/application and the *Secure Registry Tool* consists of three phases: establishment, operation, and termination. The first phase allows the *Secure Registry Tool* to authenticate the client application, and to establish the key that will be used to protect the interaction between the client application and the *Secure Registry Tool* during that session. This first stage always occurs, unless a session is already in place between the client application and the *Secure Registry Tool*. The second phase is the phase at which the *Secure Registry Tool* serves the client application. This phase includes zero or more *transaction*s. Each such *transaction* is an operation in which one *registry object* is created, read, modified, erased, or otherwise managed. The third phase is the session termination. The termination process tears down the session that was established at the first phase.

Each *transaction* involves two messages: a request sent from the client application to the *Secure Registry Tool* (the 'server-side'), and a response returning from the *Secure Registry Tool* to the client application. The request contains an operation, optional operation data, and reference to a *registry object*, identified by its ID, if applicable. IDs of *registry object*s always start with the ID of their creator client application, so that client applications cannot generate objects that seem as if they were created by other client applications.

The response to the request may include information and always includes a status code. An established session can only carry out a single *transaction* at a time. Multiple sessions, however, may be established for the same client application at a given time.

3.5.2.4 Session security Each message is protected in terms of confidentiality and integrity using

session-specific key material that was agreed during the session establishment phase. Session-key material is generated by means similar to those used by the *Key Distribution Tool* in Section 3.3 (pp. 4). Message replay (within the scope of the session) is prevented by using running counters. The use of session-keys protects against longer-term message replay. Session-specific key material is valid until the session is terminated by the client application or until the embodiment of either the client application and/or the *Secure Registry Tool* are powered down, whichever comes first.

3.5.2.5 Lookup messages The *Secure Registry Tool* also supports *lookup message*s. These are short interactions that are tailored for use by the *Code Authentication Tool*, specified in Section 3.6 (pp. 9), and by applications such as the *Secure Feature Activation Enabler* of Section 4.5 (pp. 18). These short interactions are designed to allow quick and low-overhead queries to be sent to the registry in a secure fashion, and true/false ('found' or 'not found') responses delivered; all without the establishment of a session. Such interactions are only available for lookups for certain types of *registry object*s, and are suitable when the only relying (trusting) party in the operation is the entity issuing the query, and neither the registry itself nor any other component using the registry.

The protocol used in *lookup message*s is specified in Section 3.6 discussing the *Code Authentication Tool*.

3.6 Code Authentication Tool

3.6.1 Purpose

The *Code Authentication Tool* performs one of the most fundamental functions in vehicular information system security. It allows individual controllers to be assured, upon boot, that they are about to execute the trustworthy code that they are intended to execute. Obviously, no assurance can be granted if the 'valid' code is itself not sound and erroneously causes the execution of untrusted code. (This implies that aside from using a tool, it shall be verified that the 'valid' code is sound and cannot be manipulated into placing the controller into an untrusted state.) The *Code Authentication Tool* assures that the 'valid' code, as it was approved, is indeed the code that the controller starts executing upon boot.

Put in other words, the *Code Authentication Tool* helps to assure, upon controller boot, that the code that is stored in (or next to) the controller has not been tampered with. This verification is done before the controller actually branches into the verified code.

The importance of code authentication cannot be overstated. Controllers take part in various security-related activities, interacting with other controllers (see Sections 3.3 and 3.4), storing and retrieving secure data (see Section 3.5), and performing other critical operations. The security model of *all* these operations may break if the controller code is surreptitiously modified to make the controller not behave as expected by the overall system. For example, a controller that is supposed to retrieve an entry from the secure registry and determine engine behavior by the retrieved value, may have its code 'patched' to perform an operation regardless of the value that is retrieved from the secure registry. In this case, the robustness designed into the registry is nullified by an opponent that, while not being able to change the registry, is able to cause the value obtained from the registry to be ignored.

The *Code Authentication Tool* allows the automatic detection of off-line tampering with controller code, while allowing authorized updates to this code to be performed, as described in Section 4.1 (pp. 15). If tampering is detected, the controller may either refuse to operate, may operate without the ability to authenticate as itself against the secure registry and/or against other controllers, or may run in a *reduced functionality* mode[7].

3.6.2 Method

The *Code Authentication Tool* is based on a routine that computes a one-way secure hash of the entire code base of the controller, or at least of the code range covering the boot entry point, and compares it with reference values of approved code. The reference values are stored in the secure registry, specified in Section 3.5 (pp. 7). The detachment of the on-controller verification routine from the registry allows for code to be updated by an authorized entity.

3.6.2.1 Embodiment The *Code Authentication Tool* is installed on each controller that requires code authentication services. This includes all controllers that run security-critical code, other than controllers on which code modification is physically impossible, such as ones that execute all their code from ROM.

The implementation of the *Code Authentication Tool* is minimalistic, so as to allow its incorporation into essentially all controllers, including ones utilizing low-end processors. The *Code Authentication Tool* logic essentially reads and computes a SHA-2 [6] hash, sends it over to the *Secure Registry Tool* in a *lookup message*, and receives a simple response, based on which the controller either halts, branches into *reduced functionality* mode, or makes some key material unavailable until the next boot. Execution of this logic is done from ROM, as soon as the controller boots, and before branching into any modifiable code.

[7] Description of the *reduced functionality* mode is beyond the scope of this paper.

3.6.2.2 Flow Upon boot, the following steps take place in order:

1. The code that is stored in pre-configured address ranges is read into a SHA-2 [6] hash engine.
2. When read completes, the hash engine concludes the hash computation, and the tool generates a random (or sequential) nonce. The tool then sends its own ID, the hash, and the nonce, symmetrically signed using the controller's shared key with the *Secure Registry Tool*, in a specially formed *lookup message*.
3. The *Secure Registry Tool* verifies the signature based on the key associated with the ID in the *lookup message*, and looks up the hash in the registry. The hash shall appear in a specially formed *registry object* containing the hash and the ID of the controller from which the *lookup message* should have arrived.
4. If a match of the ID and hash with a *registry object* was found, and if the controller that issued the *lookup message* has a `read` permission on that *registry object*, a positive answer is returned, signed with the same key. The response message also includes the same nonce that was included in the *lookup message*. If a match is not found, a similarly formed negative response is sent.
5. Upon receipt of the response, the logic of the *Code Authentication Tool* verifies the signature on the response, as well as the nonce.
6. If the signature verification fails, the nonce is wrong, or if the response is negative, the code is treated as non-authenticated, and the processor either halts, prevents access to key material (e.g., disables the *Key Distribution Tool*), or branches to *reduced functionality* mode (if other conditions are met.)

The protocol by which a *lookup message* is sent and a response is returned is illustrated in Protocol 2. C denotes the client — the *Code Authentication Tool*; ID_C is the identity of the *Code Authentication Tool* (the controller on which it is installed); R denotes the 'server' — the *Secure Registry Tool*; h is the computed hash; n is the generated nonce; k is the shared key between the controller on which the *Code Authentication Tool* is installed and the *Secure Registry Tool*; v is the verification response (positive or negative); and SIG_k denotes a symmetric signature computed using symmetric key k.

Protocol 2 Lookup Protocol

$C \rightarrow R$	`'REQ.LMM.V1.00'`, $\mathrm{ID}_C, n, h,$ $\mathrm{SIG}_k(\mathrm{ID}_C, n, h)$
$C \leftarrow R$	`'RESP.LMM.V1.00'`, $v, \mathrm{SIG}_k(\mathrm{ID}_C, n, v)$

For a positive response to be issued by the *Secure Registry Tool*, (i) the queried hash needs to exist in a suitable *registry object*, (ii) the ID specified in that *registry object* has to match that of the querying controller, and (iii) the querying controller has to have a `read` permission on that object. The reason for the second requirement is so code that is allowed on one controller does not get implicitly and unintentionally approved on other controllers. The reason for the third requirement is to retain consistency with the data access policy of the *Secure Registry Tool*.

When updating code, the relevant *registry object* needs to be changed to contain the new hash. If a controller is designed to manage its own code update, then it shall have a `write` permission on its own relevant *registry object*. This does not introduce a security risk because the controller is presumably in a trusted state (after code authentication passed successfully at boot) *before* it can exercise its `write` permission to change its reference hash. In other words, a controller of which code was tampered, cannot not go through its boot cycle to reach a state in which it can exercise its `write` permission to introduce its own modified code as valid code.

The code integrity assurance provided by the *Code Authentication Tool* assumes not only that valid code is also sound, but also that code cannot be altered in runtime. Controllers that allow for their code to be modified by code that is not checked during boot, may require real-time integrity checking; a discussion of which is beyond the scope of this paper.

3.7 Trusted Time Tool

3.7.1 Purpose

The purpose of the *Trusted Time Tool* is to provide any requesting application on any controller with trustworthy universal time. The tool provides this time information from an embedded secure clock that is implemented as part of the tool's embodiment.

The term 'trusted time', or 'secure time', is frequently used in designs of secure systems to denote different types of assurance regarding the provided time information. The security properties that are deemed required by a 'trusted time' source, according to the purposes described and envisioned in this paper are:

- The reported time (as stored and as reported) cannot be modified other than by a trusted time source and by the natural progress of time.
- Determination of the natural progress of time is done by embedded logic that is part of the secure embodiment of the *Trusted Time Tool*.
- Time does *not* have to be synchronized. It may lapse by up to several minutes when compared to another trusted time source. This implies that the time provided by the *Trusted Time Tool* may not be used for protocol-level replay protection in some scenarios.
- Time should be protected both against roll-forward and against roll-back. For most use-cases, protection

against roll-back shall be more stringent than protection against roll-forward.
- Time that is reported by the tool may have one of a few possible *time trustworthiness levels*. It is up to the relying application to determine if the *time trustworthiness level* that is associated with the reported time is sufficient for its purposes. This feature is necessary because trusted time information has multiple customers in a vehicle, having different requirements on the trustworthiness of the time data they use.
- Trusted time may be unavailable, for example, if the *Trusted Time Tool* cannot assess it. Reliant applications shall always support a *failure mode* if they perform a critical function, such as a function that affects passenger safety.

3.7.2 Method

The *Trusted Time Tool* maintains the current time value securely, relying on secure hardware (see Section 3.7.2.1). It responds with the current value of the time register to any controller that requests this value, over a protocol that protects the integrity and freshness of the response. When responding with the current time, the *Trusted Time Tool* also reports the *time trustworthiness level* associated with the time it reports (as described below). Lastly, the tool also provides an interface for trusted parties to update the time register, either in local proximity or over a network.

3.7.2.1 Embodiment The *Trusted Time Tool* consists of a 'server-side' that is embodied in a physically secure component. It maintains its time register using a real-time clock that is protected to a reasonable extent against physical tampering and against tampering by subjecting it to extreme environmental conditions. It may also facilitate a connection to the *Secure Registry Tool* to facilitate additional protection against roll-back, as described below in Section 3.7.2.5.

The tool also consists of a simple 'client-side' that is installed on each controller that requires to obtain time information.

3.7.2.2 Time query When any controller requires to know the current time for a purpose that requires trustworthy time, its 'client-side' of the *Trusted Time Tool* obtains a shared session-key with the 'server-side' of the *Trusted Time Tool*. This can either be done using the *Key Distribution Tool*, or using the *Secure Messaging Tool*. The use of the *Key Distribution Tool* is recommended, because confidentiality protection is not required for the time acquisition session.

Once a shared session-key, k, is obtained, the time is queried according to the protocol specified in Protocol 3. The 'client-side' of the *Trusted Time Tool*, on the querying controller Q, sends a message containing an either random or sequential nonce, n, to the *Trusted Time Tool* 'server-side' T, along with its ID. A MAC[8] is computed using k as a key, and is appended to the message. Upon receipt and verification of the message, the *Trusted Time Tool* responds with a message containing the current UTC time, t, and the *time trustworthiness level*, l, of that time value. It also appends a MAC on those two values and the nonce n, using k as the MAC key. The 'client-side' verifies the MAC and the contained nonce, and notes the specified time along with its *time trustworthiness level*.

Protocol 3 Trusted Time Acquisition Protocol

$Q \rightarrow T$	$\texttt{'REQ.TT.V1.00'}, \mathrm{ID}_Q, n,$ $\mathrm{MAC}_k(\texttt{'REQ.TT.V1.00'}, \mathrm{ID}_Q, n)$
$Q \leftarrow T$	$\texttt{'RESP.TT.V1.00'}, \mathrm{ID}_Q, t, l,$ $\mathrm{MAC}_k(\texttt{'RESP.TT.V1.00'}, \mathrm{ID}_Q, n, t, l)$

The purpose of the nonce is to prevent an attack in which responses are swapped within the same session. If the querying controller never asks for the time more than once per each session, the nonce can be made a fixed part of the message. If the querying controller does not have a reliable random number generator, the nonce may be a monotonic counter. The counter may reuse values (i.e., may reset) across sessions.

It is the responsibility of the querying controller to measure the time it takes for the *Trusted Time Tool* 'server-side' to respond, and refuse to accept the response if it arrived after a predefined constant threshold. The 'client-side' *must not* use the incoming time value in the response message from the *Trusted Time Tool* to determine if the threshold was exceeded. It *may*, however, use an internal clock or counter for this purpose.

It is also within the responsibility of the querying controller to determine if the *time trustworthiness level*, as contained in the response message, is suitable for the purpose for which the time was obtained.

3.7.2.3 Time setting The *Trusted Time Tool* allows trusted entities to update the current time that is maintained by the tool. The user, or the vehicle owner, is never considered as such a trusted entity, both because he may be motivated to subvert the contents of the time register, and because the user is generally hard to authenticate.

The time may be updated, for example, by:

- the car service center; and/or
- the car manufacturer, or anyone on his behalf, over a network.

Time trustworthiness levels Multiple entities may be entitled to set the time, and they may be associated with different *time trustworthiness levels*. When time is set by an authorized entity, the *time trustworthiness level* of the current time is set to the level that

[8] Message Authentication Code

is associated with the entity that submitted the update. The *Trusted Time Tool* maintains the current time in the time register, along with the time at which it was obtained, and its *time trustworthiness level*.

Time trustworthiness erosion When a certain time-gap between the current time and the time at which time was last updated is reached, the *Trusted Time Tool* erodes (reduces) the *time trustworthiness level*, by a fixed value, to account for a possible *significant* drift of the internal clock. It is important to note that the *time trustworthiness level* does not reflect the *accuracy* of the clock, and thus does not need to constantly be reduced as the clock (presumably) drifts. Rather, it reflects *trustworthiness* of the information, and shall thus only be reduced when enough time passed since the last update to make the time less *reliable* even for purposes that do not require close synchronization.

Time setting protocol To update the time, a trusted entity connects to the *Trusted Time Tool* using the time update protocol. The entity that issues the update is authenticated using a symmetric key, or a public key, that is maintained by the 'server-side' *Trusted Time Tool* — using the *Secure Registry Tool*, or by the *Trusted Time Tool* itself. These authentication keys may be provisioned using the *Provisioning Tool*, specified in Section 3.8 (pp. 13).

The time is updated using the protocol specified in Protocol 4, demonstrated with a public key K of the update entity U. A copy of the public key is available for the *Trusted Time Tool*, T. The protocol is initiated by the tool, but may be triggered by the update entity in some installations. The protocol starts with the tool issuing a random nonce, n. The tool assures that a reasonable amount of time passes between the issuance of the `REQ` message and the arrival of the `RESP` message. Upon reception of the response, the tool verifies the nonce (against a stored copy), and the signature on the nonce and the time information, t, using the registered public key matching the identity of the update entity, ID_U. If verification passes, the response arrived in a timely manner, and the *time trustworthiness level* associated with the entity denoted as ID_U is higher than that of the currently known time, then the tool records the provided time as the current time, and the *time trustworthiness level* associated with ID_U as the level of the current time. The time at which the time information was obtained is also recorded to support time erosion as specified earlier in this section.

Protocol 4 Trusted Time Update Protocol

$U \rightarrow T$	`'TRIG.TTU.V1.00'`, ID_U
$U \leftarrow T$	`'REQ.TTU.V1.00'`, ID_U, n
$U \rightarrow T$	`'RESP.TTU.V1.00'`, $\mathrm{ID}_U, t, \mathrm{SIG}_K(n,t)$

If the trustworthiness level associated with the source is *lower* than the *time trustworthiness level* associated with the current time, then the *Trusted Time Tool* shall carry out the protocol, but not update the time, so not to erode the *time trustworthiness level* of the time information it has.

3.7.2.4 GPS signals Some trusted time systems utilize GPS signals to update their local time. GPS signals are convenient means and are often received for navigation purposes anyway. However, they are limited in their level of trustworthiness, because GPS signals can generally be spoofed [13, 12]. GPS signals can provide time measurements that are more trustworthy than time entered by the user, suitable, e.g., for low value DRM purposes, but are probably not trustworthy enough for other uses, such as for the collection of driving information that shall be usable as evidence in court.

The current architecture allows the utilization of GPS signals by assigning them with a lower *time trustworthiness level*, when compared to time that was obtained over a secure connection from a trusted source.

When using GPS signals to set the current time, the GPS feed may be assigned a *time trustworthiness level* of b, where time obtained over a trusted link from a more trusted source (e.g., a time service provided by the car manufacturer over a cellular connection) is assigned a level a, and where $a > b$.

As long as the *Trusted Time Tool* has a current time with *time trustworthiness level* of a, it will ignore the GPS time updates, having a lower level. Over time, the *time trustworthiness level* erodes. If a source of level a ever offers an updated time, the tool will obtain an updated time, and reset the *time trustworthiness level* of the current time back to a. If no update is available from a source of level a until the *time trustworthiness level* of the current time eroded to a level below b, a GPS signal will be used to update the time and assign it with a *time trustworthiness level* of b. The *Trusted Time Tool* will use the time it received from the GPS until a source of level a offers a more trustworthy time. Until a more trustworthy time is obtained, some applications in the vehicle that use the *Trusted Time Tool* may refuse to use the provided time (of level b), whereas less critical components will use the time provided by the tool in spite of its relatively low trustworthiness value.

3.7.2.5 Roll-back prevention The current time register is maintained using physical security means, as specified in Section 3.7.2.1 (pp. 11). Physical protection of the embodiment, along with the lack of an interface to change the time (other than by trusted parties, as specified in Section 3.7.2.3), prevents an opponent from being able to set the current time register, either to a value in the future or to a value in the past.

Notwithstanding, roll-back, which is often considered as a more significant threat than roll-forward, can be further prevented by relatively inexpensive means (rel-

atively inexpensive — in comparison to the cost of the hardware protection employed). This additional protection is achieved by noting the current time at certain intervals in a secure storage space that itself cannot be rolled-back, and which is protected from illegal modification. Such a storage space is offered by the *Secure Registry Tool* of Section 3.5 (pp. 7).

Each time the current time value is served by the tool, or once in a predefined interval, the *Trusted Time Tool* records the current time in a specified *registry object* for which the *Trusted Time Tool* has exclusive `read` and `write` permissions. Upon and before modifying this object (unless a time update is being performed), the *Trusted Time Tool* reads the contents of this object and verifies that the time currently stored in it is indeed previous or equal to the currently known time that is about to be written. If the already-recorded time is ahead of (that is, greater than) the current time, then a roll-back has taken place, and the current time value shall be set to '`Unavailable`'.

The current time changes away from an `Unavailable` value only by an authenticated update, according to Section 3.7.2.3.

3.8 Provisioning Tool

3.8.1 Purpose

All tools designed and presented so far require certain key material to be provisioned to them before they can be functional. For example:

- The *Key Distribution Tool* requires each instance of it (on each controller) to have a shared symmetric key with a *Master Controller*.
- The *Secure Registry Tool* requires the controller running each client application to have a shared symmetric key with the 'server-side' of the tool.
- The *Trusted Time Tool* authenticates update entities using pre-established key material.

The keys are provisioned to the controllers in structures that consist at least of (i) 256-bit key material, and (ii) identity associated with the key. Other than symmetric keys, in some cases reference public keys are also provisioned, such as for the *Trusted Time Tool* to authenticate time update entities.

3.8.2 Method

Provisioning is done by allocating *slot*s, somewhat similarly to the model used by PKCS#11 [11]. Each tool has a (usually fixed) number of slots that can be used to store symmetric or public keys, one per each *slot*. Key provisioning is done by providing the tool with an encrypted structure containing the key to be added, the associated information (such as the ID of the key, or of the controller the key is shared with), and the identity of the slot in which the key is to be placed. Slot identifiers consist of the *key type* and a sequential number.

Slots are filled with, and freed from, key material using *provisioning message*s, as described below. These messages arrive from authorized provisioning sources, also described below.

3.8.2.1 Embodiment The *Provisioning Tool* has to be installed on every controller that hosts any other tool that uses keys, unless the tool uses only pre-loaded keys that are provisioned during fabrication. This includes essentially all embodiments that utilize any of the tools mentioned in this document. Provisioned keys are required also on instances of the 'client-side' implementation of tools. For example, every controller that uses the *Secure Registry Tool*, and thus which has installed the client-side portion of the *Secure Registry Tool*, is required to have a local installation of the *Provisioning Tool*. Due to this requirement, the *Provisioning Tool* was designed to be deployed also on low-end controllers, and also on components lacking a processor.

3.8.2.2 Interface The *Provisioning Tool* interacts with its environment by receiving and responding to *provisioning message*s coming from authorized sources. These messages are used to manage the stored keys. There are at a minimum three types of *provisioning message*s that are supported:

enumerate
: Lists IDs of keys (including their *key type*s), stored in all slots. The keys themselves are not listed.

set
: Fills a given slot with a key, associated with an ID, if the slot is empty.

clear
: Deletes the contents of a given slot, making it available.

Messages are encrypted and authenticated. To simplify the design and implementation of the *Provisioning Tool*, and also due to the fact that there is no strong reason to have it otherwise, there is only one level of authorization for the submission of *provisioning message*s. A single authorization level grants permission to submit all three types of messages, as explained in the following subsection.

3.8.2.3 Message security It must be assured that only authorized entities can submit *provisioning message*s. Otherwise, opponents can bypass most security mechanisms simply by making controllers use keys of which they have a copy. It is also essential to protect message confidentiality, because the messages carry key material and may be communicated over untrusted bearers.[9]

[9] Untrusted bearers are not necessarily the Internet alone. The ability to compromise *provisioning message*s carries with it the

Provisioning keys and delegation The root of trust for provisioning is a single symmetric 256-bit key that is fabricated into the embodiment of each instance of the *Provisioning Tool*. This key is known only to the entity that fabricated the embodiment of the tool. This key is denoted as K_M.

Each *provisioning message* and its response have their confidentiality and integrity assured by a symmetric provisioning key, K_P. It is possible that $K_P = K_M$, but it is not required. It is often the case that the fabrication entity is not the entity that provisions the operational provisioning keys into the components it produces. It is therefore possible for the fabrication entity holding K_M to *delegate* provisioning authority to another key — K_P. The operational provisioning key K_P may be generated by the owner of K_M and delivered to the entity doing provisioning, or it may be generated by the provisioning entity and *authorized* by the owner of K_M. Once K_P is authorized by the owner of K_M to issue (certain) *provisioning message*s, K_M is no longer required for such provisioning. This authorization of K_P is irrevocable.

Indication of the authorization of K_P by the owner of K_M, as well as the value of K_P, are communicated to the *Provisioning Tool* by a *delegation structure*, which is part of each *provisioning message*. The *provisioning message* contains a *delegation structure* of K_P by K_M as a preamble, followed by the body of the message, which is then authorized (and signed) by K_P.

The owner of K_P may further delegate the authority to issue *provisioning message*s to $K_{P'}$ in the same form, and so forth. The preamble of the *provisioning message* will then include a chain of *delegation structure*s. The (one or more) proper *delegation structure*/s shall appear in *every provisioning message*. The *Provisioning Tool*, being suited also for low-end platforms, is not expected to retain provisioning keys other than K_M.

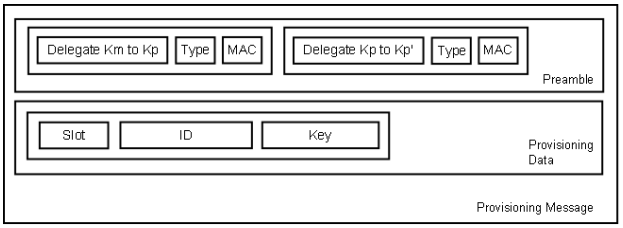
Second-order delegation, as described above, is among the most useful features of the *Provisioning Tool*. It is foreseen that not only that the fabricator of the controller running the tool will not do the provisioning, but the purchaser (often the device maker), will not provision all controllers either. It is possible that some of the provisioning may be done by the car dealership, or by other trusted parties.

Partial delegation Delegation may be partial. When the owner of K_M delegates the provisioning authority to the owner of K_P (or when the owner of K_P delegates to the owner of $K_{P'}$, and so forth), he may wish to delegate authority to provision only keys of certain *key type*s or keys for particular *slot*s.[10] To support this, the *delegation structure* consists of an indication of the *key type* that may be provisioned by the *provisioning message* in which it is enclosed. When a chain of multiple *delegation structure*s is sent, the indicated *key type* shall be identical in all components (levels) of the chain. This requirement implies that the owner of K_P cannot delegate authority to provision keys of *key type*s that he is not authorized to provision himself.

It is possible to delegate the authority to provision more than one *key type*. This is done by issuing multiple *delegation structure*s to the same entity. Each such structure covers one *key type*. The creator of the *provisioning message* is responsible for including the suitable *delegation structure* in the message.

3.8.2.4 The provisioning message Figure 2 shows the structure of a typical *provisioning message*, with two *delegation structure*s.

Figure 2: A sample *provisioning message*



To protect the confidentiality and integrity of *provisioning message*s, they are encrypted using an *authenticated encryption* mode of AES [5] using the 256-bit key K_P. The message includes *delegation structure*s suited for the *key type* of the key that is set, deleted, or queried by the message. The included *delegation structure*s chain from K_M to K_P. The main component of each *delegation structure* is the value of the lower-level key, e.g., K_P, encrypted using the higher-level key, e.g., K_M. The authenticity of the *delegation structure*s is verified by the *Provisioning Tool* independently from the authenticity of the *provisioning message* as a whole.

The response to the *provisioning message* is sent from the *Provisioning Tool*, confidentiality- and integrity-protected using the same K_P that was used to protect the *provisioning message*, unless the authentication phase failed either for the message or for any of the *delegation structure*s it contained. If such authentication errors occurred, an error is returned in plain-text.

4 Application Enablers Architecture

The purpose of the various tools detailed in Section 3 is to provide security functionality that serves security applications. This section details some of the security ap-

ability to bypass most security mechanisms employed by the various tools. Since the car dealership (or service center) is in some scenarios a possible opponent in the system, even the local wire connecting the maintenance terminal to the car at the service center is considered to be an untrusted bearer.

[10] Consider that due to the irrevocable nature of delegation, non-partial delegation of K_M to K_P is effectively similar to handing over a copy of K_M to the owner of K_P.

plication enablers that utilize components of the toolbox to provide for real use-cases that involve security considerations. Clearly, the application enablers described in this section are merely examples for security related applications that can benefit from the toolbox.

Each subsection of this section presents an application enabler, with subsections detailing its purpose, the method in which the application can run to fill its purpose, and the toolbox components that are used.

4.1 Secure Code Update Enabler

4.1.1 Purpose

The need to update the firmware of controllers is likely to occur at least a few times throughout the lifetime of the vehicle. The *Secure Code Update Enabler* is an application enabler that allows controller code to be updated securely, with authorization done through a central location in the vehicle. The enabler is responsible for obtaining signed code images and destination controllers, connecting to the controllers to be patched securely, providing them with the new code image or update information, and updating the registry accordingly, so that the controller boots properly next time.

For the enabler to work, it is required that the destination controller has an application installed which allows the reception and processing of new code information. It is not possible to update firmware code on controllers that do not support firmware update, other than by connecting to their firmware storage medium using other agents that run an update application and connect to that storage medium.

4.1.2 Method

Code update involves the following steps:

1. The *Secure Code Update Enabler* obtains the new code image (or update information) and the ID of the target controller.
2. The enabler obtains the relevant public key of the entity that is authorized to introduce code to that controller, from a *registry object* kept by the *Secure Registry Tool*.
3. The enabler checks an asymmetric digital signature on the code, its destination controller ID, and its version information, to assess its authenticity and applicability to the destination controller.
4. Following a positive verification, the enabler initiates a secure connection with the destination controller, either using the *Key Distribution Tool* (see Section 3.3) or using the *Secure Messaging Tool* (see Section 3.4).
5. Over the secure session with the destination controller, the enabler initiates the code update, and pushes the new code image, or update information, to the destination controller.
6. The controller receives the code update (as code chunks, deltas, or any other suitable format) over the secure connection, and updates its code accordingly.
7. The enabler, having a `write` permission over the relevant *registry object*, updates the object in the *Secure Registry Tool*. The update is necessary if the controller uses the *Code Authentication Tool* specified in Section 3.6 (pp. 9), so the modification of code does not cause the *Code Authentication Tool* to fail the boot process.

The concept of checking the validity of code updates at a central location (the *Secure Code Update Enabler*), rather than at the destination controller, serves the purpose of eliminating the requirement for every (possibly low-end) controller to be able to check asymmetric signatures and certificates, to be provisioned with public keys, and to handle update authorizations, separately. Instead, the controllers have a pre-provisioned trust relationship with the *Secure Code Update Enabler*, which makes update decisions for them.

4.1.2.1 Utilized tools This enabler makes use of the *Secure Registry Tool* to securely store public keys of approved update sources and to record valid code hashes that are later checked by the *Code Authentication Tool*. It also uses the *Key Distribution Tool* or the *Secure Messaging Tool* to deliver the approved code update to the destination controller.

4.2 Secure Logging Enabler

4.2.1 Purpose

The purpose of the *Secure Logging Enabler* is to collect, retain, and serve back transaction logs of all sorts. For example, the enabler can be used for:

- Drive recording (e.g., logging driving speed, durations, and routes);
- Service and maintenance logbook (e.g., times and odometer readouts at service sessions);
- Collection of troubleshooting information.

The log information has to be protected from tampering. In many cases, even the entity that generates logged transactions shall not be able to remove or modify such transactions after they were recorded.

4.2.2 Method

This enabler coordinates interaction with the *Secure Registry Tool*. It manages the interaction of the registry tool with entities of three types, each installed with a suitable component of the enabler depending on the type of the entity:

- **Transaction generation entities**: these entities are the ones that create the data that is logged. They are not necessarily privileged to read the log (which may also contain data from other transaction generation entities), or to remove transactions that they generated. Examples of such entities are GPS, odometers, and maintenance modules.
- **Log reading entities**: these entities are allowed to view certain logs, but not necessarily to change them. Examples are equipment at the service centers, or displays serving the owner of the vehicle.
- **Log clearing entities**: these entities are allowed to clear the logs. These are usually entities associated with the vehicle maker. In some cases, there are no such entities and logs are only truncated based on age of transactions. Such entities may in some rare cases be privileged to also *modify* log entries.

The enabler as a whole is configured to support a set of *log*s. Each *log* is logically defined by its types of data, the IDs of controllers that generate the data, and events that trigger the collection of this data into *log transaction*s. Upon a trigger event, the enabler component connects to each relevant controller and obtains relevant information. The data-generating controller may be initiating the delivery of data ('push' rather than 'pull') in some scenarios. The enabler component then composes a *log transaction* and connects to the *Secure Registry Tool* to have it recorded. This is done by exercising its `append` privilege on the *registry object* that is used to store the *log*. By restricting the privilege of the enabler component to `append`, it is prevented from reading *log transaction*s written by other components of the enabler, even if these *log transaction*s were written to the same log. It is also prevented from changing any items once they were recorded. Only components of the *Secure Logging Enabler* that serve "log clearing entities" (see above) require `write` permissions on the relevant *registry object*s.

For components of the *Secure Logging Enabler* that serve devices such as maintenance-related counters, such as odometers, the enabler component can run with `increment` privileges only. This will prevent the device from being able to roll back important counters, even if the device is itself completely compromised.

When serving entities that are allowed to read the log, the *Secure Logging Enabler* component exercises `read` privileges, to be able to convey log information.

Obviously, more than one instance of any of the the enabler components may be installed, each such instance having different privileges on the log data, depending on the type of entity it serves and the *log* it uses.

4.2.2.1 Utilized tools The *Secure Logging Enabler* components utilize the *Key Distribution Tool* or the *Secure Messaging Tool* to communicate with various entities that feed data to, or read data from, the log. They use the *Secure Registry Tool* to store *log transaction*s into *log*s in a way that prevents log data from being removed, or even read, by unauthorized entities. They may also unitize the *Trusted Time Tool* for obtainment of trustworthy time information to be associated with entries.

4.3 Part Authorization and Management Enabler

4.3.1 Purpose

It is often important to be able to tell if parts of the vehicular system are genuine and/or approved. Such determination is important for reasons of safety, liability, and maintenance. Even if detection of a counterfeit (or an otherwise unauthorized) part shall not result in its effective extraction from the vehicular system, it is often important to note the status of this part, so warranties can be restricted, other components can treat the non-genuine part accordingly, or so the service center is warned and may notify the driver. The driver may even be notified directly of a non-genuine part that was installed in his car.

4.3.2 Method

4.3.2.1 Authenticity implied by provisioning If a vendor can clone a part to the level that it behaves *exactly* like the original part, then its detection is impossible. However, if a genuine part contains a cryptographic component that is impractical to duplicate, then the existence of that component can often be verified and be seen as attesting for the authenticity of the part in which it is installed. To be effective, the cryptographic component shall be one that cannot be duplicated easily and one that can be authenticated by the authenticity verifier.

A cryptographic component with such properties is part of the implementation of the tools specified in Section 3. Specifically, it is part of the implementation of the *Key Distribution Tool*. Therefore, the ability to provision implementations of the *Key Distribution Tool* on original parts with key material that is not provisioned to non-genuine parts allows other controllers using the *Key Distribution Tool* to authenticate the original parts.

The *Key Distribution Tool* leverages on a pre-shared key between each approved controller and a *Master Controller* to provide shared session-keys. This implies that for a controller to be able to engage in a key agreement session using this tool, it needs to have been provisioned with a symmetric key that is shared with the *Master Controller* of the vehicle. To determine that a controller was provisioned with key material, the verifying application needs to initiate a brief interaction using a shared session-key issued by the *Key Distribu-*

tion Tool. Given a controller ID, a success in the establishment of a workable secure session with the controller holding that ID implies that the controller was properly provisioned with a key shared with the *Master Controller*. Proper provisioning of keys that are shared with the *Master Controller* presumably attests for the authenticity of the component.

This method puts the burden of *authorizing* parts on the entity that does the provisioning of key material. The entity that is responsible and authorized for provisioning is expected to only provision keys to controllers in authentic parts. This is a stringent requirement that is sometimes difficult to fulfill.

4.3.2.2 Authenticity as an attribute The key provisioning entity, while being trusted by the *Part Authorization Enabler* (as well as by probably all other components), may wish to provision key material to controllers of parts that may not be authentic, or that it cannot determine to be authentic. To support this case, it is made possible for the provisioning entity to create a *registry object* in the *Secure Registry Tool* that, for each provisioned controller identified by its ID, stores data that is readable by all relying controllers/applications in the vehicle. This object, hereafter referred to as the *attestation document*, contains indications of the authenticity and trustworthiness of the part in which the controller is installed, in any suitable language.

For example, an *attestation document* may note that the part which contains controller of ID 'XYZ' is okay to interact with but is not to be trusted to carry out particular operations, or which shall not obtain particular types of data. For example, proprietary components by non-approved vendors may interact with in-vehicle receivers and displays, but may not be entitled to modify security controls, or have access to certain types of information.

Permission to `write` to this object will be granted only to applications that are in a position to determine the authenticity, suitability, and/or trustworthiness of the component to which the *attestation document* belongs; not necessarily associated with the entity that did the key provisioning. These applications may reside outside the vehicle, and may be controlled, for example, by the vehicle maker.

In some scenarios, an application at the vehicle maker's facility may issue certificates for components, and these certificates can be verified by an application in the vehicle, which will in-turn update the various *attestation document*s accordingly, for all relying applications in the vehicle to consult.

The *Part Authorization Enabler* is thus not an independent application running on one controller, but rather it is a set of operations utilizing toolbox functions to obtain:

- the ability to distinguish between provisioned controllers (and hence, parts) and non-provisioned controllers;
- the ability to associate provisioned controllers (and hence, parts) with securely stored data that cannot be changed by the controllers themselves, and which indicates how these parts shall be treated by other applications in the vehicle; and
- ability for other applications to view the aforementioned data and determine their behavior when interacting with controllers according to this information.

4.3.2.3 Utilized tools To obtain the features listed above, relying applications utilize the *Key Distribution Tool* (to authenticate controllers and thus determine if they were provisioned properly), and the *Secure Registry Tool* to obtain information that indicates how the controllers are to be treated by the relying applications. Relying applications may also utilize the *Code Authentication Tool* to be assured that their firmware was not altered to cause them to treat unauthorized or non-genuine parts differently than intended.

4.4 Theft Prevention Enabler

4.4.1 Purpose

The motivation for preventing vehicle theft is obvious. The purpose of the *Theft Prevention Enabler* is to utilize the tools defined in Section 3 to securely accomplish conditional disablement of essential vehicle components in certain scenarios that may indicate that the vehicle is operated by an unauthorized individual.

4.4.2 Method

Two types of components are needed to implement theft prevention: (i) components that are necessary for vehicle operation, and (ii) components that can obtain information that allows to determine that the user of the vehicle is authorized to operate the vehicle. The more components of type-(i) available, the more robust the solution is. Type-(i) components consist of at least the ignition system and the gas pump [14], and may also contain the gear shifting logic and other essential circuits. Typical type-(ii) components are biometric readers and RFID receivers that challenge hand-held remote controls or key-fobs that are presumably carried only by the authorized user/s.

To accomplish theft prevention, the type-(ii) component shall retain, in volatile memory, the status of authorization, which indicates whether the user was challenged and responded properly or not. The type-(i) component/s is/are to consult the type-(ii) component securely, and disable the essential mechanism (at sensible timing) if the type-(ii) component indicates unauthorized use. This communication between the two types of components can be obtained by an open secure session on which driver authorization status is periodically

pushed by the type-(ii) component to the type-(i) components.

4.4.2.1 Utilized tools The *Theft Prevention Enabler* utilizes the *Key Distribution Tool* or the *Secure Messaging Tool* to establish trustworthy signalling between both types of components. Each component may also utilize the *Code Authentication Tool* to assure that a car thief did not tamper with the firmware to make the component ignore relevant state information, or to otherwise not operate as expected.

4.5 Secure Feature Activation Enabler

4.5.1 Purpose

Vehicles are sometimes manufactured with features that are not made available to all buyers. Car manufacturers sell cars that support different feature sets for variable prices, sometimes targeted at different customer groups. Since manufacturing cars of several profiles incurs significant added costs in comparison to a "one size fits all" design, some features are often designed to be enabled or disabled at the firmware (or configuration data) level, while the car hardware is identical. This not only reduces cost, but it also allows the user to purchase extended options for his vehicle post-purchase of the vehicle.

Premium features are worth money, and along with this worth come attempts to subvert the mechanism so it enables features in an unauthorized fashion. The threat model is more complex than a car owner attempting to enable features for which he did not pay. In some cases, a car dealership may desire to activate features on vehicles sold to end customers either for promotion or for pay, but without notifying (and paying) the car manufacturer. A secure system for feature activation shall on one hand allow a dealership to activate features on cars that it sells, but on the other hand assure that it cannot do so without either:

- the permission of the car manufacturer for such activation in advance; or
- authentic reporting on activation to the car manufacturer, after the fact.

The *Secure Feature Activation Enabler* is aimed at providing this feature.

4.5.2 Method

The main part of the *Secure Feature Activation Enabler* is an application running on a controller in the vehicle. This application, hereafter referred to as the *activation management application*, manages the activation of features. It connects to the *Secure Registry Tool*, where parameters of activated features are recorded, and to a back-end application that runs in the car dealership or service center. That back-end application is further connected to an application running at the facility of the car manufacturer.

Also in the vehicle are other controllers that actually carry out the operations involved in the optionally-activated features. These controllers utilize `read` permissions on the *Secure Registry Tool* to check the status of such may-be-activated features, before carrying out operations that depend on the activation status of these features. For example, an application that downloads map updates for the navigation system may check in the registry if such a download feature is activated, and has not expired. As another example, a controller associated with the operation of the motor and/or gas pump may consult the registry to learn the maximum horsepower that the motor is allowed to produce.

The *registry object* that contains information on the activation status of a particular feature is likely to have set a `write` permission for the aforementioned *activation management application*, and a `read` permission for the controller/s that act based on the activation status of the feature. The *activation management application* is responsible for updating the relevant *registry object*s based on requests of the back-end application at the service center or dealership, but only after the conditions involving permission of the car manufacturer were met.

Figure 3 illustrates the entities involved in feature activation.

4.5.2.1 Feature certificates To assure the approval of the car manufacturer, a request to activate a feature (or otherwise change its properties) is provided in a *feature certificate* to the *activation management application*. This certificate is a data object signed using a private key of the car manufacturer, and contains at least:

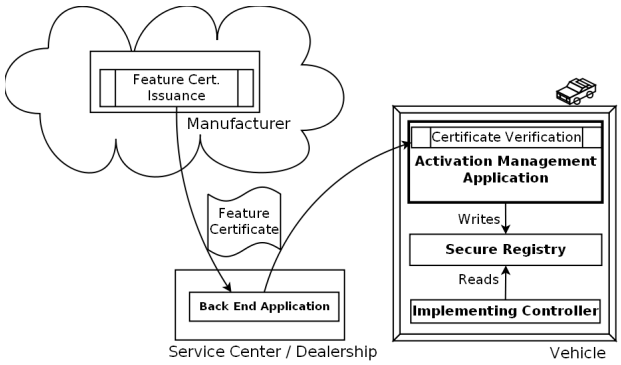
- the ID of the vehicle (e.g., in the form of an ID of the *activation management application* in the vehicle);
- the ID of the feature involved;
- the feature parameters, if applicable (activation status, activation parameters, such as expiry date, or terms in which the feature can be used);
- a time-stamp token for the certificate;
- a signature computed by an application running at the site of the car manufacturer, using the car manufacturer's private key.

The *activation management application* verifies this certificate and sets the relevant *registry object*s to align with the *feature certificate*, if the certificate was verified properly.

Since the *feature certificate* has to be signed by the car manufacturer, and since it holds the ID of the instance of the *activation management application*, the car manufacturer is always aware of features that are activated on individual vehicles. The car manufacturer issues the *feature certificate*s per request, logs and bills

the relevant entity, assuming the certificate was used to activate the feature.

Figure 3: Entities involved in feature activation



4.5.2.2 Feature certificate revocation It is possible that once a *feature certificate* was issued, it is never used. For example, it may be the case that the dealership sets to sell a certain number of cars with particular features enabled, but due to market conditions, it is driven to sell lower-end feature sets instead. The holder of a *feature certificate* may thus request a refund for the certificate, given that it was not used. To do so, the *feature certificate* is presented to the *activation management application* in the vehicle, not for consumption but for *revocation*. The *activation management application* may sign a *revocation attestation* that is delivered to the back-end application and from there on to the application at the car manufacturer's site. The car manufacturer, after verifying the *revocation attestation*, refunds the cost of the certificate (and feature) involved. The *revocation attestation*, signed by the *activation management application*, indicates that the *activation management application* in the vehicle saw the *feature certificate* specified in the *revocation attestation* and confirms to never accept this *feature certificate*. Shall the feature be required in the future, a new *feature certificate* will have to be issued.

4.5.2.3 Utilized tools The *activation management application* in the vehicle utilizes the *Secure Registry Tool* to record parameters of activated features, and to keep a public key of the car manufacturer, by which *feature certificate*s are verified. It may also use this registry to keep symmetric or private keys needed for *revocation attestation*s. This application enabler may also use the *Code Authentication Tool* to protect itself from tampering. Tampering with the code of the *activation management application* may lead to activation of features without authorization.

Other controllers that are set to operate in accordance with activation status of features may also utilize the *Code Authentication Tool* for a similar purpose.

References

[1] M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation: A two-pass authenticated-encryption scheme optimized for simplicity and efficiency. In *In Fast Software Encryption 2004*, 2004.

[2] Tom Berson. Skype security evaluation. Technical report, October 2005.

[3] Maureen C. Curran. 'auto show' of a different kind: The automotive software workshop 2006. *Newsroom — California Institute for Telecommunications and Information Technology.*

[4] Morris Dworkin. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality. In *National Institute of Standards and Technology, NIST Special Publication*, 2004.

[5] FIPS. *Advanced Encryption Standard (AES).* National Institute of Standards and Technology, November 2001.

[6] FIPS. *Secure Hash Standard.* National Institute of Standards and Technology, August 2002.

[7] Tadayoshi Kohno, John Viega, and Doug Whiting. CWC: A high-performance conventional authenticated encryption mode. In *Proceedings of FSE 2004, LNCS 3017*, pages 408–426. Springer-Verlag, 2004.

[8] D. A. McGrew and J. Viega. The galois/counter mode of operation (GCM). *NIST Modes Operation Symmetric Key Block Ciphers*, 2005.

[9] Hisashi Oguma, Akira Yoshioka, Makoto Nishikawa, Rie Shigetomi, Akira Otsuka, and Hideki Imai. New attestation based security architecture for in-vehicle communication, 2008.

[10] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, 2003.

[11] RSA Laboratories. PKCS #11: Cryptographic token interface standard, 2004.

[12] Bruce Schneier. GPS spoofing. *Schneier on Security*, September 2008.

[13] Jon S. Warner and Roger G. Johnston. GPS spoofing countermeasures. Technical report, December 2003.

[14] Wikipedia. Immobiliser — wikipedia, the free encyclopedia, 2009. [Online; accessed 1-March-2009].