Design, Implementation, and Evaluation of a Vehicular Hardware Security Module

Marko Wolf and Timo Gendrullis*

ESCRYPT GmbH, Embedded Security, Munich, Germany
{marko.wolf,timo.gendrullis}@escrypt.com

Abstract. Todays in-vehicle IT architectures are dominated by a large network of interactive, software driven digital microprocessors called electronic control units (ECU). However, ECUs relying on information received from open communication channels created by other ECUs or even other vehicles that are not under its control leaves the doors wide open for manipulations or misuse. Thus, especially safety-relevant ECUs need effective, automotive-capable security measures that protect the ECU and its communications efficiently and dependably. Based on a requirements engineering approach that incorporates all security-relevant automotive use cases and all distinctive automotive needs and constraints, we present an vehicular hardware security module (HSM) that enables a holistic protection of in-vehicle ECUs and their communications. We describe the hardware design, give technical details on the prototypical implementation, and provide a first evaluation on the performance and security while comparing our approach with HSMs already existing.

Keywords: hardware security module, automotive, in-vehicle, on-board.

1 Introduction and Motivation

Over the last two decades vehicles have silently but dramatically changed from rather "dumb" electro-mechanical devices into interactive, mobile information and communication systems already carrying dozens of digital microprocessors, various external radio interfaces, and several hundred megabytes of embedded software. In fact, information and communication technology is *the* driving force behind most innovations in the automotive industry, with perhaps 90% of all innovations in vehicles based on digital IT systems [18]. This "digital revolution" enables very sophisticated solutions considerably increasing flexibility, safety and efficiency of modern vehicles and vehicle traffic [28]. It further helps saving fuel, weight, and costs. Whereas in-vehicle IT safety (i.e., protection against [random] technical failures) is already a relatively well-established (if not necessarily well-understood) field, the protection of vehicular IT systems against systematic manipulations has only very recently started to emerge. In fact, automotive

* The authors thank Mirko Lange for his extensive help, Oliver Mischke for his valuable comments, the EC, and all partners involved in the EVITA project [15]. The work was done in scope of the European FP7 research project EVITA (FP7-ICT-224275).

H. Kim (Ed): ICISC 2011, LNCS 7259, pp. 302–318, 2012.
© Springer-Verlag Berlin Heidelberg 2012

IT systems were never designed with security in mind. But with the increasing application of digital software and various radio interfaces to the outside world (including the Internet), modern vehicles are becoming even more vulnerable to all kinds of malicious encroachments like hackers or malware [2]. This is especially noteworthy, since in contrast to most other IT systems, a successful malicious encroachment on a vehicle will not only endanger critical services or business models, but can also endanger human lives [26]. Thus strong security measures should be mandatory when developing vehicular IT systems. Today most vehicle manufacturer (hopefully) incorporate security as a design requirement. However, realizing dependable IT security solutions in a vehicular environment considerably differs from realizing IT security for typical desktop or server environments. In a typical vehicular attack scenario an attacker, for instance, has extended attack possibilities (i.e., insider attacks, offline attacks, physical attacks) and could have many different attack incentives and attack points (e.g., tachometer manipulations by the vehicle owner vs. theft of the vehicle components vs. industrial espionage). Thus, just porting "standard" security solutions to the, moreover, very heterogeneous IT environment usually will not work. However, there already exist some first automotive-capable (software) security solutions [29,32]. But, especially with regard to potential internal and physical attackers, these software solutions have to be protected against manipulations as well. In order to reliably enforce the security of software security mechanisms, the application of hardware security modules (HSM) is one effective countermeasure as HSMs:

- protect software security measures by acting as trusted security anchor,
- securely generate, store, and process security-critical material shielded from any potentially malicious software,
- restrict the possibilities of hardware tampering attacks by applying effective tamper-protection measures,
- accelerate security measures by applying specialized cryptographic hardware,
- reduce security costs on high volumes by applying highly optimized special circuitry instead of costly general purpose hardware.

Unfortunately, there are currently no automotive-capable HSMs available (cf. Section 2.2). Thus, the objective of this research was to design and prototype a standardized automotive-capable HSM for automotive on-board networks where security-relevant components are protected against tampering and sensitive data are protected against compromise. The HSM was especially designed for protecting e-safety applications such as emergency break based on communications between vehicles (V2V) or emergency call based on communications between vehicles and (traffic) infrastructures (V2I).

Our Contributions and Paper Outline. After this motivation, Section 2 gives a short introduction into our design rationale for a vehicular HSM including a short state-of-the-art review of the *related work*. Section 3 then presents the main objectives of this article, the design of a vehicular HSM concretely its system architecture, communication interface, security building blocks and security functionalities. Section 4 gives a detailed, technical overview of the HSM prototype implementation followed by a performance and security evaluation in Section 5 that compares our HSM approach with other HSMs currently available.

2 The Need For Efficient Hardware Security

In this section we provide an introduction into our design rationale for realizing a vehicular HSM. However, as this article focuses on the design, implementation, and evaluation of a vehicular HSM, this section mainly summarizes the work done before in [15] where the authors were involved, too. There, with a requirements engineering approach we identified all automotive use cases with a security impact [6], that means, all use cases involving a security-critical asset or allow for potential misuses. The security impacts identified are then transformed into high-level security objectives that could thwart these impacts accordingly. For fulfilling these security objectives, we then derived concrete (technical) security requirements [7] including all related functional (security) requirements using an appropriate security and trust model and reasonable "attack scenarios" [8].

2.1 Security and Functional Requirements Engineering

The security and functional requirements engineering is described in more detail in [6,7,9]. It has yielded to the following HSM security requisites (SR) and functional requisites (FR) as outlined below.

SR.1 Autonomous, strongly isolated security processing environment
SR.2 Minimal immutable trusted code to be executed prior to ECU processor
SR.3 Internal non-volatile memory for storing root security artifacts
SR.4 Non-detachable (tamper-protected) connection with ECU hardware
SR.5 Authentic, confidential, fresh comm. channel between HSM and ECU
SR.6 Autonomously controlled alert functionality (e.g., log entry, ECU halt)
SR.7 Only standardized, established security algorithms (e.g., NIST[1], BSI[2])

FR.1 Physical stress resistance to endure an automotive life-cycle of ≥20 years
FR.2 Bandwidth and latency performance that meets at least ISO 11898 [24]
FR.3 Compatibility with existing ECU security modules, i.e. with HIS-SHE [21]
FR.4 Compatibility with existing ECU microprocessor architectures
FR.5 Open, patent free specifications for cost-efficient OEM-wide application

2.2 Related Work

The vehicular HSM presented here is not the first security processor applied in the automotive domain. Hence, there already exist some proprietary and single-purpose HSM realizations used, for instance, by vehicle immobilizers, digital tachographs [19] or tolling solutions [34]. However, these are no general-purpose,

[1] US National Institute of Standards and Technology (www.nist.gov).
[2] German Federal Office for Information Security (www.bsi.bund.de).

private HSMs and hence cannot be reused by other vehicular security solutions. On the other hand, general-purpose HSMs that are currently available, for instance, the IBM 4758 cryptographic co-processor [4], the TCG Mobile/Trusted Platform Module [35], or typical cryptographic smartcards are not applicable for use within an automotive security context. They, for instance, lack of cost efficiency, performance, physical robustness, or security functionality. Solely, the secure hardware extension (SHE) as proposed by the HIS consortium [21] takes an exceptional position as it was explicitly designed for application in a automotive security context. However, the SHE module is mainly built for securing cryptographic key material against software attacks, but cannot be used, for instance, to protect V2X communications. An overview comparison of the HSM proposed in this work with the general-purpose HSMs is given later in Table 5.

3 Design

This section describes the hardware architecture and security functionality of our HSM. The corresponding full detailed HSM specification can be found in [9].

3.1 System Architecture

As shown in Figure 1, the hardware architecture design is based on a closed on-chip realization with a standard ECU application core and the HSM together on the same chip connected by an internal communication link. Hence, HSM commands are not explicitly and individually protected at hardware level. That means they are communicated in plain and without any replay and authenticity protection between HSM and application core. However, a TCG like command protection approach based on session keys and rotating nonces [35] is possible. The interface to the internal communication bus and external communication peripherals (if existing) are managed by the application core. The HSM consists of an internal processor (*secure processor*), some internal volatile and non-volatile memories (*secure memory*), a set of hardware security building blocks, hardware security functionality, and the interface to the application core. In order to enable a holistic but cost-efficient in-vehicle security architecture, there exist at least three different HSM variants – *full*, *medium*, and *light* – each focusing on

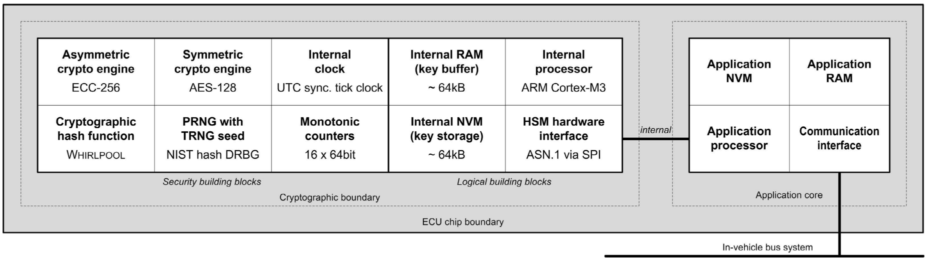

Fig. 1. Overall vehicular hardware security module architecture (full module)

different security use cases with different cost, functional and security requirements. However, these variants are no isolated developments, in fact, the light and medium module are proper subsets of the full module. The **full module** focuses on securing V2X communications and securely connecting the external vehicle interface(s) with the in-vehicle IT infrastructure. Due to its central importance and due to the strong V2X requirements, the full HSM provides the maximum level of functionality, security, and performance applying amongst others a very powerful hardware-accelerated asymmetric cryptographic building block, as shown in Figure 1. The **medium module** focuses on securing the in-vehicle communication. Hence the medium module has no dedicated hardware-accelerated asymmetric cryptographic building block, no dedicated hardware-accelerated hashing function, and a somewhat less performing internal processor. Even though, the medium HSM has no asymmetric cryptography in hardware, it is nonetheless able to perform some non-time-critical asymmetric cryptography operations (e.g., key exchange protocols) using the internal processor and firmware. As for efficiency and cost reasons virtually all internal communication protection is based on symmetric cryptographic algorithms, omitting the asymmetric cryptography and hashing hardware blocks is reasonable to save costs, size and power consumption. The **light module** focuses on securing the interaction of ECUs with sensors and actuators and is able to fulfill the strict cost and efficiency requirements typical there. In comparison with the medium module, the light module is again minimized and contains only a hardware-accelerated symmetric cryptography engine, a hardware random number generator, a UTC clock together with some very small optional volatile and non-volatile memories.

3.2 Hardware Interface in General

The HSM applies an asynchronous (i.e., non-blocking) hardware interface. The interface is multi-session capable (i.e., interruptible) for all security building blocks (cf. Section 3.3) that use a session identifier (e.g., encryptions, signatures, hash functions). Single session (i.e., non-interruptible) security building blocks, in contrast, are the random number generator, all counter functionality and all higher-level security functionality (cf. Section 3.4). Internally, the HSM is single-threaded in general, but provides limited multi-threading if different hardware functionality is accessed in parallel. Thus, for instance, one can invoke a hardware encryption in parallel with a random number generation, but not two encryptions with the same hardware encryption building block at the same time. Finally, as the HSM functionality is a proper superset of the HIS SHE functionality [21] and should be able to process all SHE commands.

3.3 Hardware Security Building Blocks

This section describes the (always) hardware-protected and (selected) hardware-accelerated security building blocks (SBB) available.

Asymmetric crypto engine enables creation and verification of digital signatures with the asymmetric signature algorithm specified on invocation using different hashing functions, different padding schemes and optional time-stamping. The full and medium prototype modules provide an ECC-256 (i.e., NIST P-256 [16] based) signature function that is hardware-accelerated at the full module. Full and medium prototype modules also provide the elliptic curve integrated asymmetric encryption scheme (ECIES) [1].

Symmetric crypto engine enables symmetric encryption and decryption with the cipher specified on invocation including different modes of operation (e.g., ECB, CBC, GCM) and different padding schemes (e.g., bit padding, PKCSx) – if available. As shown in Figure 1, all prototype modules provide at least the AES-128 block cipher [17].The symmetric crypto engine further enables generation and verification of message authentication codes (MACs) that optionally can get time-stamped using the internal tick or UTC clock (if synchronized). Thus, all prototype modules provide at least AES-128 CMAC [33] functionality.

Cryptographic hash function enables generation and verification of plain hash fingerprints and additionally HMACs (hash-based message authentication code) calculated with a secret key with the hashing algorithm specified on invocation. This SBB also provides optional time-stamping of the hashes/HMACs generated using the internal UTC clock (if synchronized). The full and medium prototype modules provide an ISO 10118 Whirlpool [23] hashing function that is hardware-accelerated at the full module.

Pseudo random number generator (PRNG) creates pseudo random numbers with a PRNG algorithm specified on invocation that can be seeded internally from a physical true random number generator (TRNG) or from an external TRNG during production in a controlled environment of the chip manufacturer. The latter case additionally requires a proper seed update protocol. All prototype modules provide at least an officially evaluated PRNG according to E.4 [31] (e.g., AES- or hash-based).

Internal clock serves as hardware-protected time reference that can be synchronized with UTC time. For further details see Section 3.4.

Monotonic counters servs as a simple secure clock alternative while providing at least 16 monotonically increasing 64-bit counters together with corresponding access control similar to TCG's monotonic counters [35].

A distinctive feature of the HSM is the possibility for very fine-grained application specific authorizations for the processing and the migration of internal security assets. Concretely, a key can have several individual authorizations that allow or forbid processing it in different SBBs specified by so-called `use_flags`. As shown in the example in Table 1, a symmetric key, for instance, can have a `use_flag` for using it for MAC verifications but has a different `use_flag` (or even *no* `use_flag`) for using it for the creation of MACs. Our HSM prototype supports key `use_flags` each with individual authorizations for processing and migration at least for signing, signature/MAC verification, data encryption and decryption, time stamping, secure boot, secure storage, clock synchronization, key creations and key transports. Moreover, these `use_flags` can have individual migration authorizations that specify their transport restrictions to locations outside the respective HSM (cf. Section 3.4). Thus, a `use_flag` can be restricted to be moved (i) to no other location (*internal*), (ii) only between HSMs identical in construction (*migratable*), (iii) only between HSMs and trusted OEM locations (*oem*), or freely to any other location (*external*). For example, the `use_flag` *verify* for signature verification of a certain key can be allowed to become migrated to another HSM (*migratable*), while the `use_flag` *sign* for signature creation of the same key cannot be moved to a location outside its local HSM (*internal*). Lastly, each `use_flags` can have also its individual authorizations required for each invocation that can be simple passwords, can be based on the individual ECU platform configuration as measured at ECU bootstrap (cf. Section 3.4), or could be even a combination of both (i.e., configuration and password).

Table 1. Internal key structure including # of instances per field

<table>
<tr><td colspan="2">key identifier [1]</td><td colspan="2">algorithm identifier [1..n]</td></tr>
<tr><td colspan="2">expiration date [1]</td><td colspan="2">key signatures [0..n]</td></tr>
<tr><td colspan="2">public key data [0..1]</td><td colspan="2">private key data [0..1]</td></tr>
<tr><td>use flag [1..i..n]</td><td>transport flag [i]</td><td>auth. flag [i]</td><td>auth. value [0..i]</td></tr>
<tr><td><i>sign</i></td><td><i>internal</i></td><td><i>password</i></td><td><i>Hash(“abc”)</i></td></tr>
<tr><td><i>verify</i></td><td><i>migratable</i></td><td><i>ecr</i></td><td><i>ECR(0;1;7) = 0x123</i></td></tr>
</table>

3.4 Hardware Security Logic and Functionality

This section gives a short description of some central HSM keys and the underlying key hierarchy used by our HSMs. The *manufacturer verification key* (MVK) is a key from the module manufacturer to verify the authenticity of other HSMs or to authenticate HSM firmware updates. The *device identity key* (IDK) enables global HSM identification and authentication. The IDK is unique per module, fixed for HSM lifetime, and signed by MVK. The *OEM verification key* (OVK) is a key from an OEM to have an additional OEM managed trust domain similar to the manufacturer trust domain controlled via MVK. The OVK is unique per OEM and is also fixed for HSM lifetime. The *clock synchronization key(s)* (CSK) are a verification keys from a trusted time reference (e.g., a certain GPS module trusted by the HSM manufacturer) accepted for synchronizing the internal tick counter to absolute UTC time. The CSK is signed by MVK. The *storage root key* (SRK) is the (symmetric) master parent key for securely swapping internally created keys to external storages similar to the SRK as introduced by the TCG [35]. The *stakeholder key(s)* (SxK) finally are all externally created symmetric (SSK) or asymmetric (SAK) keys for stakeholder individual usage such as authentication, secure feature activation, or content protection. To increase (external) trust into SxK and OVK, they *can* be signed by MVK as well.

Key management provides functionalities for internal key creation (using the internal RNG), key import and export. The HSM further provides hardware-protected functionality for Diffie-Hellman key agreements [3] and for symmetric key derivations, for instance, according to ISO 18033 [25]. The creator of a key has the possibility to set individual usage authorizations (`use_flags`) and transport authorizations (`trnsp_flags`) for each key usage as introduced in Section 3.3 and Table 1. Note that some key flags cannot always be set freely but are inherently set by the HSM (e.g., the *internal* transport flag). For moving keys between different HSMs, between HSMs and external (trusted) locations (if permitted), the HSM provides key import and export functionality that ensures confidentiality of private key internals via (symmetric or asymmetric) transport encryption as well as authenticity of all key data structures via (symmetric or asymmetric) so-called transport authenticity codes (i.e., a digital signature or a MAC). Strictly speaking, not the whole key itself is moved, but only individual key `use_flags` if they have proper transport authorizations (`trnsp_flags`). The `trnsp_flags` inherently define also the keys that can be used for transport encryption and authenticity enforcement. Hence, use flags of keys marked *internal* are only permitted to become swapped out to offline storage and imported again to the same HSM via the SRK. Use flags of keys marked *migratable* are additionally permitted to become moved between HSMs identical in construction. This is enforced by accepting only target IDKs for transport encryption that are signed by a trusted MVK (e.g., MVKs from the same manufacturer used for HSMs of at least equal physical security). A similar approach is foreseen for use flags of keys marked *oem* that accept for transport encryption only keys that are signed by a trusted OVK. This was introduced to support an OEM managed trusted domain that can be differentiated (but not enforced!) by the HSM in contrast to use flags of keys marked *external* that are fully out of the control of the HSM for enforcing any trust assumptions.

Secure boot and authenticated boot is realized using HSM-internal so-called *ECU configuration registers* (ECR) that are similar to TCG's platform configuration registers (PCR) [35]. In contrast to the TCG approach, our HSM is also acting as the – by all involved parties a priori trusted[3] – core root of trust (CRT) that initializes the chain of trust for ECU integrity and authenticity verification. Assuming a multi-stage bootstrap with the CRT executed first, the trust chain is built by a hierarchical step by step *measuring* of the program code of all upper layers i such as the boot ROM, the boot routine, the boot loader, and all consecutive layers that are part of the Trusted Computing Base(TCB)[4]. For the HSM, the corresponding measurement routine $m()$ that creates a small unique fingerprint f_i for each program code measured, is the Whirlpool one-way hash function for *full* and *medium* modules and the AES-MAC for *light* modules[5]. The fingerprint f_i in turn is saved to an individual ECR_n protected against manipulations inside the HSM before the next stage becomes executed. In order to prevent the overwriting of a previously saved ECR by layers executed later, ECRs in fact can only become *extended* via $\text{ECR}_n[\text{t}] = m(\text{ECR}_n[\text{t-1}], f_i)$ that connects the new fingerprint f_i cryptographically with the old $\text{ECR}_n[\text{t-1}]$ value and hence prevents overwriting. To detect and counteract possible vali-

[3] It is per definition impossible to self-verify the integrity of the *core root* of trust.

[4] The TCB means all code of the ECU that can be critical for ECU security.

[5] The bootstrap security functionality is optional for light HSMs.

dation failures, there exist at least two different approaches, usually known as *secure boot* and *authenticated boot*. In case the HSM can be deployed as active module (e.g., as ECU master) and hence having autonomous control to the corresponding ECU or an alarm function, the HSM can realize the active secure boot approach. Then the procedure executed at each step of the bootstrap uses the HSM to compare the actual value of an $\mathrm{ECR}_n[\mathrm{t}]$ with the corresponding reference value $\mathrm{ECR}_{n,ref}$ that can securely preset for each ECR (cf. [9] for further details). In case of a mismatch between $\mathrm{ECR}_n[\mathrm{t}]$ and $\mathrm{ECR}_{n,ref}$, secure boot automatically yields to an immediate response (e.g., ECU halt or alarm). Authenticated boot, in contrast, remains rather passive while only measuring the bootstrap without any direct interventions. It can be used if the HSM can be deployed as passive add-on module only having no autonomous control to the ECU or no alarm function. By using the authenticated boot approach, some essential key `use_flags`) (e.g., decryption) can become inaccessible afterwards in case the actual bootstrap measurements do not match the ECR reference individually linked for this particular `use_flag`) by the key owner. Our HSM therefore can enforce individual ECR references $\mathrm{ECR}_{n,ref}$ as additional `use_flag` invocation authorization (cf. Section 3.3) that makes the corresponding key `use_flag` inaccessible (and only this) in case of an ECR mismatch.

Secure clock is realized by a so-called *tick counter* t_c that is monotonically increasing during HSM runtime using a fixed `tick()` interval of at least 1 Hz. As shown in Table 2, the initial value of t_c on HSM's first initialization in life is $t_c = 0$ or $t_c = \mathrm{UTC}(t_0)$ that represents a UNIX/POSIX encoded default time value, for instance, the HSM release date. After any reset, the HSM continues increasing t_c unsynchronized (i.e., $t_c < \mathrm{UTC}(t)$), but starting with the last internally saved[6] value of t_c. This provides a simple relative secure clock that – even if seldom or never synchronized – never "runs slow". However, optionally t_c can be synchronized to absolute UTC time received from an external UTC source (e.g., in-vehicle GPS sensor) trusted by the HSM manufacturer (e.g., in-vehicle GPS sensor). Therefore, the HSM can be invoked with the actual UTC time and the HSM synchronization challenge (as requested from the HSM before) both signed with a trusted CSK (cf. Section 3.4).

Table 2. HSM clock with relative *tick time* optionally synchronized to absolute UTC

On clock event	t_c	t_c vs. $\mathrm{UTC}(t)$
First initialization in HSM life	0 or $\mathrm{UTC}(t_0)$	$t_c < \mathrm{UTC}(t)$
Internal `tick()` after reset	t_c++	$t_c < \mathrm{UTC}(t)$
External UTC synchronization	$\mathrm{UTC}(t)$	$t_c = \mathrm{UTC}(t)$
Internal `tick()` after sync	t_c++	$t_c = \mathrm{UTC}(t)$

[6] The internal saving interval for t_c depends on the rewrite capabilities of the HSM internal non-volatile memory and can vary from seconds up to hours or even days, but is always inherently invoked after a successful UTC synchronization.

Administration and audit functionality provides HSM status information, self tests, internal state backup, migration, and updates. Auditing the HSM operations can enhance the security by enabling an (external) auditor to retrace for instance critical operations (e.g., key migrations) or critical incidents that have occurred (e.g., authorization failures, resource exhaustions).

3.5 Driver Software and Software Security Framework

The HSM provides a set of basic hardware security functionalities on which a larger set of more sophisticated software security mechanisms can be built on. However, these higher-level software security mechanisms are not part of this work. But the corresponding HSM low-level software driver [12], an appropriate high-level software security framework [13] and proposals for implementing secure on-board protocols using the HSM as basis [10] are already available.

4 Implementation

The HSM architecture has been prototypically implemented [11] on a *Xilinx Virtex-5* FPGA (*XC5VFX70T-1FF1136C*). This FPGA has a total amount of 11,200 slices (i.e., 44,800 both flip-flops (FF) and 6-input look-up tables (LUT)), 128 dedicated digital signal processing (DSP) blocks, and 148 Block-RAMs (BRAMs) with 36 Kb of memory each. Additionally, the FPGA comprises an embedded hard-coded microprocessor block, that is, a PowerPC 440 (PPC440) with a maximum frequency of 550MHz. The FPGA comes on the *Virtex-5 FXT FPGA ML507 Evaluation Platform* with all necessary peripherals such as various interfaces, both volatile memory and non-volatile memory (NVM), and power supply. The *application processor* was implemented on an *Infineon TriCore TC1797* with all necessary peripherals and resources (i.e., application NVM, application RAM, communication interfaces) available on a corresponding development board. Both development boards are mechanically and electrically attached to each other via a *custom-made connection board* that provides access from the application core to the HSM. For this purpose the connection board wires the TC1797's external SPI bus with general purpose in-/output pins (GPIOs) of the FPGA. Figure 2 gives an overview of the layered approach of the HSM implementation. In the prototypical design, the HSM's *secure processor* is mapped to the embedded PPC440 of the FPGA that runs at 400 MHz with a standard Linux 2.6.34 kernel as operating system. This is the basis to host software implementations of the HSM firmware, the cryptographic library, and the Linux kernel drivers to access the hardware cores. The hardware cores implemented in the configurable logic of the FPGA are connected to the PPC440 via the 32 bit wide Xilinx specific processor local bus (PLB). Thus, the interface between software and hardware implementation has a theoretical throughput of 3 Gbit/s at the given PLB bus speed of 100 MHz.

First, all SBBs of Section 3.3 (also cf. Figure 1) and all HSM security functionalities of Section 3.4 were implemented in software (i.e., *HSM library*) as a reference for verification and performance analysis. The underlying cryptographic primitives of the SBBs were made available in a cryptographic software

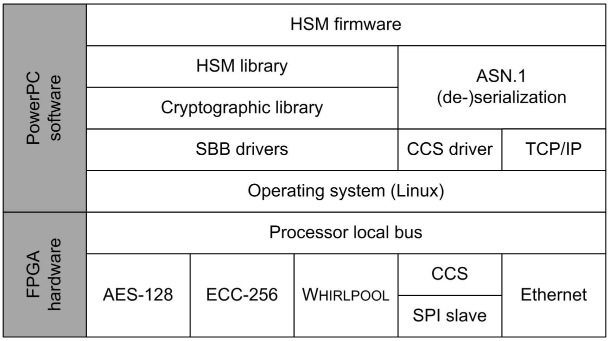

Fig. 2. Architectural overview of the prototypical HSM implementation

library (i.e., CycurLIB [5]). Afterwards, those cryptographic primitives involved in the vast majority of all SBBs were additionally implemented in hardware (i.e., AES-128, ECC-256, and WHIRLPOOL). In a straight-forward approach, they were wrapped with the PLB interface and cryptographic algorithms in software were enhanced by non-blocking hardware calls via custom SBB Linux kernel drivers. Different modes and protocols based on one of the three cryptographic primitives in hardware are also part of the cryptographic library using the hardware calls. SBBs complementary to these three algorithms remain available as pure software. The HSM firmware is the top level entity that handles all communication requests, that means, all requests from internal hardware cores and also all external requests (i.e., from the application processor). Access from the application processor to the secure processor of the HSM is provided via the configuration, control and status module (CCS). The CCS is also part of the HSM and its basic task is to make the communication interface independent (i.e., to abstract it) from the underlying physical communication medium. The CCS includes a shared message RAM to buffer issued commands and their answers. The instantiation of the CCS module in the HSM prototype utilizes an SPI slave module with a data rate of 22.5 Mbit/s to connect the HSM to the *TC1797*. However, because of the separate CCS abstraction layer, the design can be easily enhanced to physical communication mediums other than SPI (e.g., to a socket based communication using TCP/IP over Ethernet for demonstration purposes with standard PCs). Since a serial bus (i.e., SPI) connects the HSM with the application processor all data has to be serialized before transmitting and de-serialized after receiving. This is done by a separate (de-)serialization layer using abstract syntax notation number one (ASN.1 [22]). At the application processor side, commands are serialized accordingly into a binary encoding and return values are deserialized with the common ASN.1 representation of the HSM API. The SBBs implemented in hardware are briefly described in the following. Afterwards, Table 3 shows a comparison of the entire HSM system design and the single SBB hardware implementation results (time and area) after synthesis. The synthesis results for the SBBs are mentioned as well for the sole hardware core as for the core with hardware overhead for PLB integration.

AES-128 was implemented in ECB mode with a focus on a very low area, still providing a reasonable performance. To keep the design small, the implementation is comprised of four 8 bit S-boxes that are shared between all round functions and the key scheduling. With this architecture, encrypting one 128 bit block in ECB mode takes 53 clock-cycles. As the AES operates with a clock of 100 MHz, it has a theoretical throughput of 242 Mbit/s. For decryption, the AES needs another 53 clock-cycles to perform the key scheduling each time a new decryption key is used. Thus, in the worst case, the decryption data rate decreases to 50% compared to encryption. Due to the small hardware footprint, several instances could be mapped to the design in parallel to increase the performance. Since all remaining operation modes of AES (e.g., CBC) share the ECB mode as their common atomic operation, they are realized in software finally using hardware calls of the AES core in ECB mode.

WHIRLPOOL was implemented with a moderate time-area trade off. Both, the underlying 512 bit block cipher W (which is very similar to AES) and the enclosing *Miyaguchi-Preneel* compression function were implemented in hardware. In our implementation, the 512 bit state of W consists of eight 8 Byte blocks operating on eight 8 bit S-boxes. Both the round function and the key scheduling share all computational modules (i.e., S-boxes, ShiftColumns, MixRows). With this architecture, one 512 bit block operation takes 308 clock-cycles. To keep the number of different clock domains in the HSM design at a minimum, WHIRLPOOL operates at the same clock of 100 MHz as the AES-128 resulting in a theoretical throughput of 166 Mbit/s.

ECC-256 was implemented based on the work of Güneysu and Paar [20] with a strong emphasis on performance optimization. Only the point multiplication of the NIST P-256 [16] ECC operation is implemented in hardware while all remaining parts of the algorithm are still performed in software. Even though this is only a small portion of the algorithm it covers over 90% of the entire ECC computation time. The design is highly optimized for the Xilinx DSP architecture and uses a pipelined architecture with input and output queues. Operated at a clock of 200 MHz, one ECC point multiplication (for signature generation) with 303,450 clocks takes 1.52 ms on average (i.e., 659 OP/s) and two point multiplications plus one addition with 366,905 clocks takes 1.83 ms (i.e., 545 OP/s).

Table 3. Synthesis results for the prototypical HSM implementation

Module	FF	LUT	BRAM	DSP	Critical Path
AES standalone	279	1,137	0	0	9.878 ns
AES with PLB	744	1,399	0	0	9.860 ns
WHIRLPOOL standalone	2,656	2,932	0	0	6.364 ns
WHIRLPOOL with PLB	3,445	3,826	0	0	6.386 ns
ECC standalone	1,854	1,964	13	32	3.539 ns
ECC with PLB	2,102	2,095	13	32	4.426 ns
HSM system	16,273	18,664	37	45	—

5 Evaluation

This section provides an evaluation on HSM performance and HSM security based on the security requirements stated in Section 2 together with a comparison of our HSM approach with other relevant HSMs already existing.

5.1 Performance Analysis

After fully implementing and extensively testing the HSM hardware design, the implementation was finally deployed on the target platform. Table 4 shows a comparison of the hardware cores' theoretical throughput from the previous section and the SBB performances measured in both pure software and hardware accelerated versions. All measurements were performed on the internal PPC440 based on the cryptographic library CycurLIB [5] enhanced by drivers for hardware core access. On first sight, the AES-128 hardware accelerated implementation gains only between $5 - 40\%$ compared to the pure software solution. Moreover, it reaches only $20 - 41\%$ of the theoretical throughput of the hardware core depending on the mode of operation. This seemingly small yield of performance is mainly caused by the very high bidirectional data traffic over the PLB bus. Since the AES core operates only on single 128 bit blocks in ECB mode, each en-/decryption requires first sending and afterwards receiving a small block of 128 bit data. Additionally, all remaining modes (including MAC generation) have to perform their protocol overhead in software based on the ECB block operation in hardware. Together with (bus) latencies this explains the relatively small performance gain. With DMA (direct memory access) usage and slight modifications to the hardware implementation, e.g., enhancing wrappers for different operation modes, buffers for in-/output data or using pipelining, the throughput could be increased vastly. On the other hand, the Whirlpool implementation reaches 77% of its theoretical performance in hardware. Similar to the AES, it operates on single blocks (of 512 bit for Whirlpool) but the communication remains unidirectional except for the last block of the hash computation when the result is returned. This saves a vast amount of bus traffic (and latencies). While the hardware performance benefits from the larger block size of W (i.e., 512 bit), it makes computation in software much harder. Actually, the hardware accelerated Whirlpool outperforms the software version by a factor of 25. Optimizing the software implementation might reduce this gap slightly but will never close it entirely. From all SBBs, ECC-256 benefits the most from the hardware acceleration. The more complex ECDSA signature verification (two ECC point multiplications plus one addition vs. one ECC point multiplication only) is 29 times faster in hardware than in software. Since one ECC operation requires only sending 1024 bit and receiving 512 bit data, the requirements on communication bandwidth are very low. This allows the ECC core to reach up to 80% of its theoretical throughput although the complete ECDSA protocol (except for the point multiplication) is performed in software.

Table 4. Comparison of SBB Performance-Measurements in Software and Hardware

	Throughput		
SBB Mode	SW (measured)	HW (measured)	HW (theoretical)
AES-128, ECB encrypt	53 Mbit/s	76 Mbit/s	242 Mbit/s
AES-128, ECB decrypt	53 Mbit/s	58 Mbit/s	121 Mbit/s
AES-128, CBC encrypt	46 Mbit/s	68 Mbit/s	242 Mbit/s
AES-128, CBC decrypt	44 Mbit/s	46 Mbit/s	242 Mbit/s
AES-128, CMAC generate	44 Mbit/s	60 Mbit/s	242 Mbit/s
Whirlpool, hash generate	5 Mbit/s	128 Mbit/s	166 Mbit/s
ECC-256, ECDSA generate	30 sig/s	480 sig/s	659 OP/s
ECC-256, ECDSA verify	15 sig/s	436 sig/s	545 OP/s

5.2 Security Analysis

This section shortly analyses the fulfillment of the security requirements (SR) stated in Section 2 by our HSM approach. We fulfill *SR.1* (*isolated security processing environment*) as our HSM is realized as an autonomous microprocessor with its own independent memories (physically) isolated from the ECU main processor (cf. Section 3.1). The shielded execution environment can also be strengthened further against physical manipulations by applying appropriate physical tamper-protection measures [27]. We fulfill *SR.2* (*immutable core root of trust code*) by having foreseen an HSM deployment architecture that either assumes an active HSM (part) executed prior to the ECU main processor or an additional, small, immutable component that initializes the hierarchical bootstrap measurements (cf. Section 3.1). We fulfill *SR.3* (*internal non-volatile memory*) by having our HSM foreseen at least about 64kB non-volatile memory for shielded storing of security root artifacts (cf. Section 3.1 and Figure 1). We fulfill *SR.4* (*non-detachable connection with ECU hardware*) by assuming a system-on-chip (SoC) design, for instance, that ensures at least tamper evidence if someone tries to detach the HSM from the corresponding ECU. However, in the end, this requirement has to be fulfilled by the ASIC system designer, who can apply different powerful tamper protection measures (e.g., cf. [27]). We fulfill *SR.5* (*secure HSM-ECU communication channel*) by assuming a secure communication channel between HSM and ECU realized by appropriate physical tamper protection, for example, by assuming an SoC communication line. In case this is not possible, we propose to realize a secure communication channel by cryptographic means, for instance, as proposed by TCG's command transport encryption [35]. However, the additional cryptographic protection of HSM commands is not part of our current design. We fulfill *SR.6* (*autonomously controlled alert functionality*) by moving this alert functionality inside the HSM's cryptographic boundary (cf. Figure 1). The specification of the actual alert response (e.g., from autonomous log entry up to ECU full stop), however, should be defined by the respective ECU application (engineer). We finally also fulfill *SR.7* (*standardized, well-established security algorithms*) for all security building blocks, concretely by applying the NIST advance encryption standard [17] for symmetric encryption, the NIST digital signature standard [16] for asymmetric encryption, the ISO hash function standard [23] for cryptographic hashing, a BSI evaluated PRNG according to E.4 [31] for random number generation, and the ISO standardized ECDH [25] for key agreements.

5.3 Comparison with Other Hardware Security Modules

This section provides an overview comparison of our HSM with other hardware security modules currently available. Table 5 compares our three different HSM variants – full, medium, and light – with the Secure Hardware Extension (SHE) as proposed by HIS [21], with the Trusted Platform Module (TPM) as proposed by the TCG [35], and with a typical cryptographic smartcard (SmC). While the TPM and the smartcard were neither designed nor applicable for use within an automotive security context (as they lack, for instance, in cost efficiency, performance, physical robustness, or security functionality), SHE was explicitly designed for application in an automotive security context. It can be seen that the HSM is quite successful in merging the relevant security functionalities from SHE, from the TPM and from a typical smartcard while transferring them into the automotive domain with its strict cost efficiency and performance requirements. Beyond this, Table 5 clearly indicates the distinctive security features, for instance, the possibility for very fine-grained application-specific authorizations

Table 5. Comparison with other hardware security modules

	Full	Medium	Light	SHE	TPM	SmC
Cryptographic algorithms						
ECC/RSA	■/■	■/■	□/□	□/□	□/■	⊞/⊞
AES/DES	■/⊞	■/⊞	■/□	■/□	□/□	⊞/⊞
WHIRLPOOL/SHA	■/■	■/■	□/□	□/□	□/■	⊞/⊞
Hardware acceleration						
ECC/RSA	■/□	□/□	□/□	□/□	□/□	□/□
AES/DES	■/□	■/□	■/□	■/□	□/□	□/□
WHIRLPOOL/SHA	■/□	□/□	□/□	□/□	□/□	□/□
Security features						
Secure/authenticated boot	■/■	■/■	⊞/⊞	■/□	□/■	□/□
Key AC per use/bootstrap	■/■	■/■	■/⊞	□/■	⊞/■	□/□
PRNG with TRNG seed	■	■	■	■	■	■
Monotonic counters 32/64 bit	■/■	■/■	□/□	□/□	■/□	□/□
Tick/UTC-synced clock	■/■	■/■	■/■	□/□	□/□	□/□
Internal processing						
Programmable/preset CPU	■/⊞	■/⊞	□/⊞	□/■	□/■	⊞/⊞
Internal V/NV (key) memory	■/■	■/■	⊞/⊞	■/■	■/□	■/□
Asynchronous/parallel IF	■/⊞	■/□	■/□	■/□	□/□	□/□

Annotation: ■ = available, □ = not available, ⊞ = partly or optionally available for the processing and the migration of internal security assets or the possibility for a secure (UTC) time reference, that are exclusively available with this HSM.

6 Conclusion and Outlook

Based on a tight automotive-driven security requirements engineering, we have designed, implemented, and evaluated a vehicular hardware security module that enables a holistic protection of all relevant in-vehicle ECUs and their communications. Practical feasibility will be proven with first HSM-equipped passenger vehicle demonstrators [14] by end of 2011. Further large-scale field operational tests with hundreds of HSMs and HSM-equipped vehicles that amongst others address performance, scalability, and deployment of our HSM approach are already scheduled in a subsequent research project [30]. We are convinced that future interactive vehicles can be realized in dependable manner only based on an effective, efficient holistic in-vehicle security architecture, which in turn is based upon effective and efficient vehicular HSM as presented by this work.

References

1. Abdalla, M., Bellare, M., Rogaway, P.: DHAES: An encryption scheme based on the Diffie-Hellman problem. Submission to P1363a: Standard Specifications for Public-Key Cryptography, Additional Techniques 5 (2000)
2. Checkoway, S., et al.: Comprehensive Experimental Analyses of Automotive Attack Surfaces. National Academy of Sciences Committee on Electronic Vehicle Controls and Unintended Acceleration (2011)
3. Diffie, W., Hellman, M.: New Directions in Cryptography. IEEE Transactions on Information Theory 22(6) (1976)
4. Dyer, J., Lindemann, M., Perez, R., Sailer, R., Van Doorn, L., Smith, S., Weingart, S.: Building the IBM 4758 Secure Coprocessor. IEEE Computer 34(10) (2001)
5. escrypt GmbH – Embedded Security: CycurLIB - Cryptographic Software Library (2011), http://www.escrypt.com/products/cycurlib/overview/
6. EVITA: Deliverable 2.1: Specification and Evaluation of E-Security Relevant Use Cases (2008)
7. EVITA: Deliverable 2.3: Security Requirements for Automotive On-Board Networks Based on Dark-Side Scenarios (2009)
8. EVITA: Deliverable 3.1.2: Security and Trust Model (2009)
9. EVITA: Deliverable 3.2: Secure On-board Architecture Specification (2010)
10. EVITA: Deliverable 3.3: Secure On-Board Protocols Specification (2010)
11. EVITA: Deliverable 4.1.3: Security Hardware FPGA Prototype (2011)
12. EVITA: Deliverable 4.2.2: Basic Software (2011)
13. EVITA: Deliverable 4.3.2: Implementation of Software Framework (2011)
14. EVITA: Deliverable 5.1.2: On-board Communication Demonstrator (2011)
15. EVITA Project: E-safety Vehicle Intrusion proTected Applications, European Commission research grant FP7-ICT-224275 (2008), http://www.evita-project.org
16. FIPS-186-3: Digital Signature Standard (DSS). NIST (1994, 2006)
17. FIPS-197: Advanced Encryption Standard (AES). NIST (2001)

18. Frischkorn, H.G.: Automotive Software – The Silent Revolution. In: Workshop on Future Generation Software Architectures in the Automotive Domain, San Diego, CA, USA, January 10- 12 (2004)
19. Furgel, I., Lemke, K.: A Review of the Digital Tachograph System. In: Embedded Security in Cars: Securing Current and Future Automotive IT Applications. Springer (2006)
20. Güneysu, T., Paar, C.: Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 62–78. Springer, Heidelberg (2008)
21. Herstellerinitiative Software (HIS): SHE Secure Hardware Extension Version 1.1 (2009), `http://portal.automotive-his.de`
22. International Telecommunication Union – ITU-T Study Group 7: Abstract Syntax Notation number One – ASN.1 (1995), `http://www.itu.int/ITU-T/asn1/`
23. ISO/IEC 10118-3:2004: Information technology – Security techniques – Hash-functions – Part 3: Dedicated hash-functions. ISO/IEC (2004)
24. ISO/IEC 11898:2003-2007: Information technology – Road vehicles Controller area network. ISO/IEC (2007)
25. ISO/IEC 18033-2:2006: Information technology - Security techniques - Encryption algorithms - Part 2: Asymmetric ciphers. ISO/IEC (2006)
26. Koscher, K., et al.: Experimental Security Analysis of a Modern Automobile. In: IEEE Symposium on Security and Privacy (2010)
27. Lemke, K.: Physical Protection against Tampering Attacks. In: Embedded Security in Cars: Securing Current and Future Automotive IT Applications. Springer (2006)
28. Luo, J., Hubaux, J.: A Survey of Inter-Vehicle Communication. EPFL, Lausanne, Switzerland, Tech. Rep (2004)
29. PRECIOSA Project: Privacy Enabled Capability in Co-operative Systems and Safety Applications (2008), `http://www.preciosa-project.org`
30. PRESERVE Project: Preparing Secure Vehicle-to-X Communication Systems (2011), `http://www.preserve-project.eu`
31. Schindler, W.: AIS 20 – Functionality classes and evaluation methodology for deterministic random number generators. German Federal Office for Information Security (BSI) (1999)
32. SeVeCom Project: Secure Vehicular Communication (2006), `http://www.sevecom.org`
33. Song, J., Poovendran, R., Lee, J., Iwata, T.: The AES-CMAC Algorithm. RFC4493, IETF (June 2006)
34. Toll Collect GmbH (2011), `http://www.toll-collect.com`
35. Trusted Computing Group (TCG): TPM Specification 1.2 Revision 116 (2011), `http://www.trustedcomputinggroup.org`