Firmware Security Module

A Framework for Trusted Computing in Automotive Multiprocessors

Claudius Pott¹ · Philipp Jungklass² · David Jacek Csejka² · Thomas Eisenbarth¹ · Marco Siebert²

Received: 2 October 2020 / Accepted: 13 April 2021 / Published online: 30 April 2021
© The Author(s) 2021

Abstract

New services such as autonomous driving, the connectivity of the traffic infrastructure and the tight coupling of user operated smart devices with the vehicle have significantly increased the demand for cryptographic protection in the automobile. To provide a secure environment for the calculation and verification of cryptographic material, automotive microcontrollers now frequently integrate Hardware Security Modules (HSM), special well-protected co-processors, which are protected against manipulation and external access. HSMs use special hardware accelerators to provide the required cryptographic services. While these accelerators provide good performance, they limit flexibility and updatability. In addition, as more services require cryptographic protection, the amount of key material that needs to be managed by the HSM also increases, turning the limited protected memory of the HSM into a bottleneck. This paper presents a framework that uses the safety mechanisms of a microcontroller to achieve both HSM-equivalent security through software solutions, while providing an equivalent level of functionality. Furthermore, the proposed framework provides crypto-agility, as the security stack can be updated if desired. In order to verify the functionality, the presented framework is implemented and evaluated on an Infineon AURIX TC399 and compared with the integrated HSM.

Keywords Embedded software · Cryptography · Random number generation · Real-time systems · Memory management

1 Motivation

In recent years, modern automobiles have developed from vehicles which are independent and completely isolated from external communication to connected systems which are integrated into the traffic infrastructure. The permanent

connection of these systems to the existing communication infrastructure allows new functions such as autonomous driving, traffic-dependent navigation and the integration of smart devices. However, every part, software or hardware, added to a car must conform to the high safety standards of the automotive industry, as human lives are potentially at risk in the event of a malfunction. The permanent connectivity of modern vehicles results in new security requirements for their integrated systems. Whereas in the past, functional safety was more important, today additional constraints such as secure communication, verification and authentication of software updates as well as compliance with legal privacy requirements must be met¹. This results in an increased demand for cryptographic operations, which are indispensable for the provision of these services. For this purpose, various manufacturers of control units for security-relevant applications integrate special co-processors into their microcontrollers, which are called Hardware Security Modules (HSMs). These

✉ Claudius Pott
c.pott@uni-luebeck.de

Philipp Jungklass
philipp.jungklass@iav.de

David Jacek Csejka
david.jacek.csejka@iav.de

Thomas Eisenbarth
thomas.eisenbarth@uni-luebeck.de

Marco Siebert
marco.siebert@iav.de

¹ University of Lübeck, Institute for IT Security, Lübeck, Germany

² IAV GmbH, Embedded Security, Gifhorn, Germany

¹ Standardization of the requirements by the norm ISO/SAE 21434

processors are characterized by the fact that they are particularly isolated from the overall system and have a large number of hardware accelerators that speed up the calculation of cryptographic operations. It is becoming increasingly apparent that the demand for security-relevant functions is growing faster than the available computing power that can be provided by an HSM. This circumstance is further exacerbated by the fact that the manufacturers of such microcontrollers often use lower-performance processor cores for the HSM than for the general purpose processor cores in the system [1, 2]. To compensate this deficit in computing power, hardware accelerators are used to calculate cryptographic operations. However, these are designed specifically for one functionality and are therefore inflexible in their possible applications. A similar behavior, analogous to the computing power, can also be observed for the memory sizes of the HSM. Currently, the limited memory provided by state-of-the-art HSMs is a major problem, because more and more functionality is relocated to the HSM due to the increasing number of security relevant services. The same applies to the key material, which is increasing in parallel with the number of functionalities. This article presents a framework that isolates one high-performance processor core of such a multicore system in a way that HSM-equivalent security and functionality is achieved. The framework, which will be referred to as FSM, is designed to be implemented on different platforms. Therefore, part of the discussion is focussed on security requirements that need to be fulfilled when implementing the framework for a specific architecture. The FSM framework allows to satisfy all the requirements placed on an HSM and utilizes the significantly higher computing power and the larger memory of the high-performance cores. In order to evaluate the approach, a concrete implementation was conceptualized for the Infineon AURIX TC399XP. Here the extensive safety features of the AURIX microcontroller family are utilized to isolate a part of the available memory to effectively prevent unauthorized access to confidential data. Cryptographic operations are implemented in software and compared with the hardware-accelerated implementations of the HSM of this chip. Furthermore, concepts are presented that enable additional functions such as secure boot, secure deletion and the logging of security-relevant events.

2 Related Work

The challenge of securing applications from attackers that control the OS is a well researched topic and different approaches address this problem. Three popular mechanisms providing secure computations to a different extent are Trusted Execution Environments (TEEs), HSMs, and Trusted Platform Modules (TPMs).

Trusted Execution Environments provide secure *enclaves* in which software can be executed even in untrusted systems on the primary CPU. This means that the full processing power of the CPU can be used while a layer of isolation prevents even the operation system from accessing the data of the enclave. Most of today's commodity CPUs implement forms of TEEs [3–5], which need hardware support. In deeply embedded CPUs, on the other hand, TEEs are still rather uncommon.

Hardware Security Modules approach this challenge from a different angle. They are implemented as isolated hardware, connected to the CPU for secure computations. Usually it is possible to store keys in these modules and to use them for cryptographic computations [6]. It is common that the secret keys never leave the HSM and are thus not at risk from being leaked. HSMs in general are created for a specific purpose and thus only provide a fixed set of services. However, some HSMs are programmable and can be used more flexible since the exact services that are provided can be defined by their firmware [1, 2].

Trusted Platform Modules are security modules for which the interface and services are standardized [7]. In most desktop computers, TPMs are implemented in separate hardware, meaning an HSM complying with the standard is used. However, there are also approaches that implement TPMs in firmware [8], the required level of isolation is accomplished by using TEEs as a basis. TEEs can be used to run arbitrary attested code in untrusted environments, while the scenario usually assumes that the code is provided during runtime. This can for example be used for computations on confidential data in untrusted cloud environments. HSMs and TPMs on the other hand provide a fixed set of services. These are usually cryptographic services like key storage, memory encryption, document signing and the verification of signatures. In contrast to HSMs and most TPMs, no isolated hardware components are used in this approach. It is however useful to consider work done in those fields, since many challenges are the same. Wolf and Gendrullis [6] have designed an HSM for the deployment in a vehicular setting. It gives an overview of the services that need to be provided in such a setting. A firmware-based TPMs for mobile devices has been discussed in [8]. In contrast to the presented approach, they require hardware modification.

In safety-critical real-time systems, the use of Memory Protection Units (MPUs) has been a solution for many years to avoid interference with runtime and memory consumption. Recent literature offers various approaches for isolating applications of different criticality [9–12]. In contrast to the concept presented in this article, however, all authors focus exclusively on achieving the highest possible safety-level. Security-relevant factors are not discussed when evaluating the different approaches.

3 The FSM Framework

3.1 Requirements

The use case of FSMs is a more flexible alternative to HSMs, which means that FSMs must fulfil the same or similar security requirements. The most important security features for HSMs are *strong isolation* and *secure storage*. For HSMs, physical isolation in form of a separate chip or fully isolated core ensures that software compromises on the main processor will not compromise the HSM or any sensitive key material stored in its protected memory. For FSMs, physical isolation is not an option. Thus, logical isolation options for memory and resources during run time must be enforced such that even OS-level adversaries may not compromise or manipulate the functionality and state of the FSM, including its storage. Secure storage is required in order to protect secret keys and confidential data used for FSM-internal computations. Secure storage is also needed to prevent the OS from deleting parts of the FSM and thus disabling services like secure boot. HSMs for automotive applications are used to ensure that safety systems cannot be illegally modified, since this would cause an unknown state for which safety cannot be guaranteed. Higher-level services of HSMs that are essential in the automotive sector are *secure boot*, *secure updates* and *a permanent locking mechanism* [6]. Secure boot is required for tuning protection and the prevention of sabotage. Secure updates are vital to prevent compromise of the system through the update process. The whole system is permanently locked whenever some form of tampering is detected. As for HSMs, their implementation requires careful system design as well as cryptographic services, i.e., *symmetric and asymmetric cryptosystems*, as well as a *RNG*. These services are also made available to processes outside the FSM (e.g., for secure communication) via an API. Besides strong logical isolation and above security services, performance and security of the crypto primitives are key factors for the quality of the FSM. While physical and side-channel attacks may be of relevance in the described use case, they are not explicitly considered for the framework. Protection against these is dependent on the concrete implementation of services and the used platform, and it can therefore be included in specific designs. Note that the strong logical isolation of the FSM's resources already prevents contention-based side-channel attacks (e.g., cache-attacks). Further, cryptographic agility, i.e., the ability to exchange cryptographic services and implementations during the lifetime of an FSM, is a major advantage over the more statically designed HSMs.

3.2 Proposed Structure of an FSM

The main idea of the framework is to isolate one core of a multiprocessor from the rest of the system. This core will therefore act as an independent entity and exclusively execute FSM computations. This also means that the OS running on the other cores must not be able to change the configuration of the FSM or prevent it from starting in the first place. The basic structure of the FSM consists of five components, which are shown in Fig. 1. The provided services and necessary configurations are grouped into three categories, which are managed by a separate manager each. The first category contains the more complex services like the secure boot and secure updates. All cryptographic services as well as the key management make up the second category, and the remaining services are combined in the group of auxiliary services. All of these services are managed by and run in the FSM's own OS. It defines a time base and provides the activity manager, which assigns the requests from the bridge module to the corresponding service managers. The final component defined in the framework is a bridge module, which presents the only way for other cores to access services of the FSM. To this end, the bridge module periodically checks designated shared memory regions for requests made by the host system. This means that other cores cannot access the FSM directly and it ensures that only requests that are explicitly allowed by the bridge module are handled. In the remainder of this section, some general requirements for the services of the different categories will be discussed.

The secure boot process needs to run and complete once before even the operating system of the host is allowed to boot, in order to prevent an altered operating system from causing any harm. The same procedure will be repeated periodically during runtime. Should some violation be detected, the system must not be allowed to start and it needs to get permanently locked. Secure updates check any given software updates for their validity before deploying them. This process must also update the protected secure boot parameters, making it the only valid option to change any software or configuration. After a successful software update, a restart is executed, during which the secure boot validates the updated software. A requirement for HSMs in automotive applications is the option of permanently locking the Electronic Control Unit (ECU). This functionality is used, for example, if a manipulation of the ECU software or the stored parameter set is detected by the secure boot. In this case, it can no longer be guaranteed that the ECU complies with the high safety standards and therefore needs to be rendered unusable. The previously discussed functionalities draw on basic cryptographic services. Since the FSM is implemented in software only, there is no option

Fig. 1 Structure of the FSM framework with data flow

to use hardware accelerators for computation-intensive tasks. The system's core is, however, more powerful than that of comparable HSMs. Further, the software approach provides flexibility and timely updates without the need for new hardware iterations, which would be necessary to replace hardware accelerators. This is also the reason that no fixed cryptographic algorithms are chosen for the FSM framework. Another task in this component is the key management, since it is a requirement that the private device keys are only accessible by their corresponding cryptographic modules and must never leave the FSM. Finally, the FSM requires a RNG. HSMs usually provide true RNGs, which use special hardware in order to create true randomness. Unfortunately, true randomness is not available in software-only implementations of RNGs. Instead, a deterministic RNG is used, whereas the implementation method needs to be chosen according to the platform. Auxiliary services make up the third component, which is managed by the auxiliary manager, that assigns requests to the corresponding services and monitors their response time.

An internal log for security relevant events is one of these services. A time stamp is stored for each entry in order to register the frequency of these events and to be able to analyse them. For many applications, it is also essential that an ECU can be uniquely identified, for example for communication with the manufacturer's backend, which is why the framework provides an identifier. Another service of the FSM is secure deletion. During the computations of the FSM, confidential data will be copied into its memory units. In order to not accidentally leak this data, it must be securely erased as soon as it is no longer needed. A correct device configuration is also crucial as demonstrated by Majéric et al. in [13]. They showed that an open debug interface can allow direct access to all memory units of a system independent of otherwise effective MPU configurations. Therefore, such interfaces must be deactivated or password protected in production usage. Finally, isolating the FSM core from the rest of the system requires protecting its resources from unauthorized access. It is vital that the memory units, caches and computational registers, used

Fig. 2 Architecture of the Infineon AURIX TC399XP

by and assigned to the FSM, cannot be accessed by any other core. How this can be achieved depends strongly on the used platform and is therefore implementation specific.

4 FSM Example Implementation

4.1 AURIX Architecture

The multicore microcontroller Infineon AURIX TC399XP is designed for safety-critical applications [1, 14, 15]. Figure 2 shows its schematic structure. The used microcontroller is a derivative with six proprietary TriCore processor cores, which operate with 300 MHz clock frequency and are based on the modified Harvard architecture. Accordingly, each processor has separate interfaces for code and data. Each of these interfaces consists of a scratchpad, which must be managed by the developer, and a two-way associative cache, which is managed by the core itself. In addition to these local memory units, each processor core has a Static Random-Access Memory (SRAM) and a separate flash memory. All memory units of the Infineon AURIX can be used for both data and code, but in the case of the local memory units the maximum access speed is only achieved when used as specified. The size of all memory units in the system is listed in Table 1. A crossbar is used to connect the remaining memory units and to enable the communication between the processor cores. The connection of various peripheral modules is realized via a bus system which is accessed competitively by all cores. This bus system also provides the connection to the HSM which thus gains access to all memory units in the system.

Memory Protection Due to the focus of the AURIX microcontroller family on safety-critical applications, the TC399XP has a large number of MPUs that protect against

unauthorized access. The MPUs were primarily integrated into the chip to isolate functions of different criticality from each other, but they can also be used to implement access authorization. In general, each memory unit in the AURIX has a separate MPU that can exclude processor cores from access. Access authorization can be defined for the complete memory unit or for selected memory areas. The type of access can be restricted, e.g., only allowing read access. The MPU is configured via specially protected registers, which can also be protected against manipulation. In addition to the memory unit's MPUs, each processor core has its own MPU with which different tasks running on that core can be isolated from each other. For this purpose, different configuration sets can be stored for these MPUs, allowing to change the configuration, e.g., at a context switch. Another way to prevent memory manipulation is to use the irreversible One-Time-Programmable (OTP) feature of the AURIX. Using this feature, memory areas can be permanently configured in such a way that they are locked for further modification. A special feature of the AURIX's OTP capability is being able to differentiate between write and read operations.

Table 1 Memory sizes of the Infineon AURIX TC399XP

Processor Core	Flash-Memory / KB	SRAM-Memory / KB
Core 0 (TriCore)	Flash Bank 0: 3192	Program RAM 0: 64 Program Cache 0: 32 Data RAM 0: 240 Data Cache 0: 16 RAM Bank 0: 64
HSM (Cortex-M3)	Flash Bank 0: 640 (Provided by the host system)	Unified RAM: 96 Unified Cache: 4

Privilege Modes Each processor core of the Infineon AURIX has a rudimentary rights management system consisting of three authorization levels. The three modes differ in terms of access to critical registers. In User Mode 0, only tasks, which do not require interaction with peripherals or configuration registers, can be performed. In User Mode 1, access to peripherals is possible, but system-critical properties such as the MPU cannot be manipulated. Only the supervisor mode allows unrestricted access to all registers and peripherals. However, it should be noted that critical configuration registers can also be locked in such a way that even in supervisor mode access is only allowed after a restart.

Hardware Security Module For the calculation of security-relevant functions, the TC399XP has an integrated HSM, based on a Cortex-M3 with 100 MHz, which corresponds to the full standard according to the EVITA classification [6]. The HSM is connected to the host system via the peripheral bus and has separate, specially protected memory areas which are provided by the host system. Communication between the HSM and the host system is realized via a special bridge module, which enables the transmission of commands. Furthermore, the HSM can trigger interrupts which are routed to the host system. A special feature of the HSM is that it has full read access to all memory units in the host system. It is not possible to limit access by an MPU. To accelerate cryptographic operations, the HSM has special hardware units for the calculation of AES 128, PKC ECC 256 and SHA2 256 as well as an AIS 31 compliant True Random Number Generator (TRNG). Finally, it is important to note that the HSM provides no integrated protection against possible side-channel attacks. The realization of this protection is the task of the used firmware.

Secure Boot The system start of the Infineon AURIX is performed on processor core 0, which starts the HSM if configured accordingly. The HSM is used to implement the secure boot process, which validates the memory contents of the whole system. Only after confirmation of the HSM, the start process of the host system is continued and, depending on the configuration, the debug interface is initialized. The processor core 0 also activates the other processor cores.

4.2 Implementation of an FSM

The core that is chosen as the FSM is core 0. The advantage of this choice is that it also handles the initial start-up of the whole system and is thus the first core to become active. This means that no other core can change the FSM's configuration before it is even started or prevent it from booting at all. This section will discuss the implementation of all components that are defined in the framework.

Secure Boot Secure boot is implemented analogously to the concept used by Infineon in the HSM. For this purpose, the function for verifying the memory content is called directly after system startup. Since the FSM is a software solution, different variants for verifying the memory are possible. In addition to simpler implementations such as verification using hashes, variants based on signatures are also supported. It should be noted, however, that the start time is significantly extended for complex cryptographic operations, which is only possible to a limited degree in automotive applications. To effectively protect the secure boot functionality from manipulation, the corresponding memory area is marked as OTP. The same also applies to the memory area that defines the start address of the processor core 0, so that skipping the secure boot is prevented. In addition to validating the memory contents, the secure boot also checks whether the debug interface and the MPU are configured correctly. Only if all boundary conditions are fulfilled, the secure boot starts the application code.

Secure Update Secure software updates are implemented in the same way as the secure boot procedure. The required boot loader is also located in a memory marked as OTP. Again, different variants of a secure software update are

Fig. 3 Concept for the permanent locking of the FSM

possible, because all cryptographic operations can be provided in software. The variant with the highest security level is the authentication using asymmetric cryptography. For this purpose, the manufacturer of the ECU signs the software update with a private key, whereby the public key is stored in the protected memory of the FSM. To ensure that the public key is not altered, the OTP feature can also be used here. After a successful software update, a restart is executed, during which the secure boot validates the updated software.

Permanent Lock The implementation of this feature can be realized by using the OTP functionality of the flash. A schematic illustration can be found in Fig. 3. The secure boot code is stored in sector 0 of core 0, which verifies all memory units directly after the system start. Since only core 0 is started at this time, the other processor cores are not able to manipulate this process. If the check is successful, the software jumps to the normal startup code, which is located in sector 2 and also initializes the other processor cores. If, however, an error is detected during the check, a jump is made to the following sector 1, which contains an endless loop. It is important to note that sector 0 and sector 1 are implemented as OTP memory. As flashing these memory areas is irreversible, they can be assumed secure. Since the verification of the memory is not only executed during the system start, but is also repeated cyclically at runtime within the scope of tuning protection, it is necessary that a detected manipulation is not forgotten after a system start. For this reason, a flag is set in a free and

FSM-exclusive memory area and then marked as read-only using the OTP feature. This flag is also analyzed by the secure boot and the endless loop is started accordingly. As the authors have shown in [13], specific security checks can be skipped by means of targeted failures, which can also bypass the implementation of the permanent lock. The safety functionalities of processor core 0, the dedicated lockstep core and the memory correction, help to detect and prevent this kind of manipulation. If a corresponding failure is detected during the execution of the permanent lock, a restart is initiated, which results in the execution of the endless loop.

Like explained in the framework, the system's core is more powerful than that of comparable HSMs. Further, that of the AURIX has a direct connection to the crossbar allowing quicker memory accesses for the FSM than it is the case for a peripheral HSM.

Cryptographic Methods The cryptographic algorithms chosen for this FSM implementation are AES 128, SHA 256 and ECC 256, which represents 128-bit (pre-quantum) symmetric security. These algorithms are also implemented in the AURIX HSM, easing a comparison of the achievable performance in Sect. 5. For asymmetric encryption elliptic curve cryptography is used and Curve25519 is implemented. That particular curve is considered secure and allows for efficient computations [16], making it well suited for the deployment in smaller processors [17].

Fig. 4 Concept for the bridge module of the FSM

Private Keys To protect the device's private keys, all FSM services run in User Mode 0, with the only exception being the Advanced Encryption Standard (AES) module, which runs in Supervisor mode. This does not restrict any computations done by those services, since no accesses to peripherals or system configuration registers are required. By default, any access to the device keys is prohibited by the core's MPU, making them inaccessible to all User Mode 0 services. When the AES module requires access to one of those keys, it changes the MPU's configuration, then reads the key and afterwards resets the MPU's configuration to its default setting.

Random Number Generator Deterministic RNGs require a seed, which is used to generate pseudo-random numbers. For the FSM, a truly random seed is stored in its protected flash memory during the time of programming. To avoid the output of the same sequence of numbers after each reboot, the number of times the system was booted is stored in non-volatile memory and incremented during the boot process. This number and the random seed are then used to generate a temporary seed used by the deterministic RNG until the next reboot. Additionally it is possible to replace the truly random seed with encrypted updates to introduce new randomness.

Memory Protection The local memory units of core 0 are exclusively assigned to the FSM by using their MPUs, making them inaccessible to all other cores. This configuration is done and locked during the start-up of the system, before any untrusted core is started. Protecting the used configuration registers ensures, that no core can change them during runtime [9]. Using the same mechanisms as the HSM, ensures strong isolation of the FSM's resources. Since the cache of core 0 is not shared, it requires no explicit protection to prevent cache-attacks.

Debug Setup In the second AURIX generation, the debug interface is deactivated using special configuration registers. These registers are read out by the FSM at system startup and are checked for a correct configuration. If this check fails, all of the system's memory units are deactivated using the OTP capability and locked for potential readout. If the system is already under the control of the debugger at this time, a readout cannot be prevented [1].

Event Log In the implementation of the FSM framework, all events are recorded in a ring buffer with 256 entries. The time for the timestamps is provided by the internal system timer 0, which is implemented as a free-running 64-bit counter, where the current counter value cannot be manipulated by a program. The only way to manipulate the system timer is to disable it. Therefore, the corresponding registers are protected by core 0 against external access by the untrusted processor cores

directly after system startup. For this reason, an external Real-Time Clock (RTC) is not used, since it offers significantly more potential attack vectors. To overcome the resets of timer 0 at each restart, the FSM logs every restart to the event log.

Secure Deletion The FSM implementation provides a function to be called every time a cryptographic computation completes. It first zeros the memory regions holding variables during the computations and afterwards clears the caches of core 0.

Unique ID In the Infineon AURIX, a Target ID is stored in the chip during production and then protected against manipulation by using the OTP feature [1]. This ID is used by the FSM, which provides it to other applications via an interface.

Bridge Module A special characteristic of real-time capable systems is the requirement for absolute determinism. Only when this is given, a real-time system can always meet the required deadlines under all conditions. For this reason, memory is allocated statically and functions are executed in a plannable time period. This behavior is also applied to the FSM, which polls the requests of the other processor cores in a fixed time schedule and processes them accordingly. All access periods and the number of queues and requests are stored in the request configuration. The memory units used for all communication are the global SRAM memory units of the respective processor cores. These provide request queues at fixed addresses in their memory, which the bridge module periodically checks for new requests (see Fig. 4). As multiple applications of different criticality can be executed on one processor core, the bridge module supports a flexible number of request queues per core. The queues are polled by the request manager, who is given the corresponding cycles by the request configuration. For each query, the validity of all contained requests is compared with the request configuration. All valid requests are transferred to the request queue of the bridge module, which holds them for processing inside the FSM. The activity manager of the operating system reads the request queue cyclically and processes it accordingly. The results are only written back at a time defined by the request configuration. This procedure ensures that the FSM behaves in an absolutely deterministic way to the outside system, making inferences about the current workload of the FSM impossible.

5 Experimental Results

To verify the concept for the framework, a test environment is implemented, which is designed according to current automotive and real-time ECUs. Since the most used standard

Table 2 Development and test environment

Designation	Utilization
Erika Enterprise ERIKA v3 GH56	Operating System
Infineon Software Framework	Development framework
Infineon Low Level Driver 1.0.1.10.0	Microcontroller driver
Tasking Compiler TC62r1 (O3 –trade-off=0)	Compiler, Linker
Lauterbach TRACE32 2018-02	Debugger
Infineon TriBoard TC399 TH V2.0	Evaluation Board

is the Classic Platform of the AUTomotive Open System ARchitecture (AUTOSAR), it is also used for the test environment. A project based on the Infineon Software Framework and using ErikaOS v3 as OS is created. The Infineon Low Level Driver library is used to control the integrated peripheral modules of the AURIX TC399XP. The used components and development tools are listed in Table 2. The feasibility of cryptographic algorithms on a core of the AURIX will be evaluated here. To this end, some of these algorithms were ported to that architecture, making it possible to compare their performance to that of comparable HSMs. The properties that are evaluated are the execution time for basic operations and the size of the implementation. All measurements taken for the FSM are using the hardware performance counters of the executing core in order to measure the number of clock cycles needed for completion. The executed code and used variables are stored in the local program and data scratchpad RAMs. The measurements that are taken for a comparison are made inside an HSM of the AURIX TC399XP. For this purpose, the internal timer of the HSM is used to perform time measurements based on the HSM's clock. Since the AURIX core and the HSM are clocked differently, all measurement values from the HSM's counter are multiplied by three for the following evaluations. The code size of the HSM is not used for comparison, since it performs these tasks using integrated hardware accelerators. The 256-bit variant of Secure Hash Algorithm (SHA) as well as the 128-bit and 256-bit variants of AES were implemented for the evaluation. The results of all measurements are shown in Table 3. Note that for the HSM the input data need to be copied into specific registers, this copying process is included in the measurements to obtain comparable results. For the evaluation of SHA, 256-bit values were hashed and the duration of this operation was recorded 512 times. It is shown that the software implementation of the SHA algorithm is slower than the hardware accelerators of the HSM by a factor of about 4.5. The measurement results for the FSM have only two possible outcomes, which could be explained by different timing behavior of memory calls or instructions depending on their input. The measurements for the HSM in comparison are

Table 3 Execution times of cryptographic operations (in TriCore clock cycles). Note that the HSM offers no hardware-support for AES 256

Algorithm		FSM	HSM
SHA 256	First execution	4463	1239
	Later executions	4463	993
AES 128 (256)	Initialization	1432 (1859)	-
	Encryption	574 (768)	361 (n/a)
	Decryption	584 (786)	395 (n/a)

not that predictable, which is likely caused by the hardware accelerator's connection to the HSM's processor via a bus that does not guarantee exact response times. Further, it is noteworthy that the first SHA computation of the HSM takes nearly 300 clock cycles longer to complete, which could be caused by an initialization of the SHA module. The measured SHA implementation for the FSM requires a size of 972 bytes, which is well within the available memory of the FSM. Note that it is to be expected that optimizations on the assembly level would further improve the performance of SHA. For AES, the implementation of OpenSSL² was ported and optimized for the AURIX architecture. It uses the faster T-table approach, which requires more memory space. Should memory space become an issue in the FSM due to more complex services, it is possible to switch to a slower S-box implementation. For the software implementations, three different operations were measured: the initialization (which needs to be done once per key), the encryption and the decryption. For the HSM, no initialization is measured, since it seems to happen in the background. Each measurement has been done for a single block in ECB and was repeated 512 times. The FSM implementation only is about 1.6 times slower than the hardware accelerators. The execution times for the FSM are constant, while the HSM, again, produces some unpredictable variation, likely caused by the usage of a bus for the hardware accelerator. It is also noteworthy that the software implementation needs more time for the initialization than for the encryption and decryption, because some key-dependent values are precomputed during that phase, which speeds up the encryption and decryption. The AES implementation for the FSM has a size of 11056 bytes of which 8488 are used to store constants like the required T-tables. Even though this implementation is much bigger than the SHA algorithm, both will easily fit into the local SRAM memory units, not even filling them to a quarter. To show the flexibility of the FSM approach, the 256-bit variant of AES has also been implemented and the measured execution times are shown in brackets in Table 3.

² <https://github.com/openssl/openssl/tree/master/crypto/aes>

These experiments show that the software implementations do not provide more performance than hardware accelerators that are specifically designed for their tasks. However, the AES implementation shows that, depending on the algorithm, it is possible to get close. Note that even though the AES runs in constant time, these implementations are not designed to provide protection against side-channel attacks. It is possible to prevent such attacks in software implementations; however, this can also cause a (significant) loss in performance.

6 Discussion and Future Work

This paper presents the FSM framework, which isolates one core of a system and provides all capabilities and services typically offered by an HSM. The primary characteristic of this approach is that all configurations and implementations are done purely in software. The framework lists services that need to be offered, describes an architecture for implementations to follow and discusses requirements that need to be met on the used platform. Note that a generic analysis of the achievable security primitives for the framework is not feasible because how and to what extent one core can be isolated is highly dependent on the architecture and provided features of the target platform. When implementing the framework on a new platform, it is important to ensure that the software approach introduces no vulnerabilities (e.g., via side-channels). One benefit of using a high-performance core is the direct and fast connection to all memory units of the system. This is advantageous for the offered services, since many of these require access to such memory units. Further, the better performance of the processor benefits all computations that are typically done by an HSM, but for which it has no hardware accelerators (e.g., pre-computations for cryptographic algorithms). Most importantly, the software approach gives the FSM an amount of flexibility that is impossible to achieve when using an HSM. This can be a huge benefit for the long-term security of the system, since outdated cryptographic algorithms can be replaced or upgraded with a simple software update, which might become necessary when algorithms for post-quantum security are standardized. Updating HSMs is much more difficult, since the hardware accelerators define which algorithms can be computed efficiently. The amount of memory available to the FSM adds immensely to its flexibility. An HSM usually has a limited amount of memory that cannot be exceeded. For the FSM, however, additional memory units can be added, without giving up the strong isolation of FSM resources. Moreover, complete cores could be added to the FSM to greatly enhance its computational power. This could be used to provide certain applications exclusive access to an own FSM, eliminating contentions caused by other processes, which could be especially useful for real-time applications. A

concept for an implementation in the AURIX 2G architecture has been presented. Strategies for protecting the FSM's private memory units from access by other cores, for preventing malicious software from interfering with the FSM's boot process and for tampering protection show that strong isolation can be achieved using the FSM framework. The experiments show that the increased flexibility comes at a price, since the lack of hardware accelerators also has a downside. Software implementations of cryptographic services result in an overhead in computation time. However, the AES experiments showed that the overhead can be small, making software implementations a feasible option. Note that all operations usually executed on the HSM's processor (like additional cryptographic operations) profit from the higher performance of the FSM. On the AURIX, a disadvantage is the exclusive configuration of safety features (e.g., MPU) by the FSM, which makes it safety-relevant. This means that the firmware of the FSM must conform to the highest safety standard (e.g., ASIL) of all other applications on the ECU. Additionally, core 0 is mainly used as a safety core in many applications, since it is activated first at startup. Future research could focus on combining an FSM core and an HSM to unify the advantages of both approaches where the faster hardware accelerators and TRNGs can be used, while complex services run in the FSM profiting from the faster processor. Furthermore, it is planned to implement the FSM framework on additional hardware platforms. Thereby, validating whether the requirements and approaches of the framework are generic and comprehensive enough to be applied to different platforms.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability On request

Code Availability Only the FSM codebase can be made available, not the Infineon AURIX driver library

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Infineon Technologies AG (2020) AURIX TC3xx User's Manual. Infineon Technologies, AG. V1.2.0. https://www.infineon.com/dgdl/InfineonAURIX_TC3xx_Part2UserManualv01_00EN.pdf?

2. STMicroelectronics (2018) SPC58xEx/SPC58xGx 32-bit Power Architecture microcontroller for automotive ASILD applications - Reference Manual. RM0391 Rev. 5. https://www.st.com/resource/en/reference_manual/dm00148989spc58egline32bitpowerarchitectureautomotivemcutriplez4cores180mhz6mbytesflashhsmasildstmicroelectronics.pdf
3. ARM A (2009) Security technology building a secure system using trustzone technology (white paper). ARM Limited
4. Costan V, Devadas S (2016) Intel sgx explained. IACR Cryptology ePrint Archive 2016:86
5. Kaplan D, Powell J, Woller T (2016) Amd memory encryption. White paper
6. Wolf M, Gendrullis T (2012) Design, implementation, and evaluation of a vehicular hardware security module. In H. Kim, editor, *Information Security and Cryptology - ICISC 2011*. Berlin, Heidelberg, Springer Berlin Heidelberg, p 302–318
7. Group TC (2020) Tpm 1.2 main specifictaion. <https://trustedcomputinggroup.org/resource/tpm-main-specification/>. Accessed 04 June 2020
8. Raj H, Saroiu S, Wolman A, Aigner R, Cox J, England P, Fenner C, Kinshumann K, Loeser J, Mattoon D, Nystrom M, Robinson D, Spiger R, Thom S, Wooten D (2016) fTPM: A software-only implementation of a TPM chip. In 25th USENIX Security Symposium (USENIX Security 16), Austin, TX. USENIX Association, p 841–856
9. Barth T, Fromm P (2016) Functional safety on multicore microcontrollers for industrial applications. In Embedded World 2016 Exhibition & Conference
10. Hattendorf A, Raabe A, Knoll A (2012) Shared memory protection for spatial separation in multicore architectures. In 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), p 299–302
11. Jungklass P, Berekovic M (2019) Memopt: Automated memory distribution for multicore microcontrollers with hard real-time requirements. In 2019 IEEE NORCAS, p 1–7
12. Saidi S, Ernst R, Uhrig S, Theiling H, de Dinechin BD (2015) The shift to multicores in real-time and safety-critical systems. In Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis. IEEE Press, p 220–229
13. Majéric F, Gonzalvo B, Bossuet L (2018) Jtag fault injection attack. IEEE Embed Syst Lett 10(3):65–68
14. AG IT. AURIX 32-bit microcontrollers for automotive and industrial applications - Highly integrated and performance optimized. Infineon Technologies AG. V1.0
15. AG IT. HSM - Hardware Security Module, AURIX TC2xx Microcontroller Training. Infineon Technologies AG. V1.1
16. Bernstein DJ (2006) Curve25519: New diffie-hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography - PKC 2006*. Springer Berlin Heidelberg, p 207–228
17. Düll M, Haase B, Hinterwälder G, Hutter M, Paar C, Sánchez AH, Schwabe P (2015) High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. Designs, Codes and Cryptography 77(2–3):493–514