RSA Authentication for Secure Flashing of Automotive ECUs

Eduardo Ciniglio[*] Emilio P. Mancini[†] Umberto Villano[‡]

November 17, 2010

Abstract

In many embedded applications in the automotive field, there is the need to protect the data and software in Electronic Control Units (ECUs). To avoid illegal write and read operations on flash memories we propose a method to identify the operator in a secure way, by means of asymmetric key cryptographic algorithms based on the RSA standard. We measure the performance and memory usage of the authentication system, discussing if they are compatible with the memory footprint and processing power of embedded devices.

1 Introduction

Electronic Control Units (ECUs) have become pervasive in automotive systems. In most modern vehicles, they monitor a number of subsystems, from motor to entertainment modules. Moreover, they control virtually every function in luxury class vehicles. Typically the ECUs are embedded systems made up of processor, a (small) amount of volatile memory, flash memory and ROM. Most of the software and the data are stored in flash memories, to make it possible the variation of control parameters or software upgrades, as well as the addition of new functionalities.

In these systems, the security of the read/write access to flash memories is undoubtedly an issue of paramount importance. Allowing the access to unauthorized operators opens an easy way to frauds (e.g., odometer frauds: odometer data is stored in flash memory), to the tuning of motor parameters to evade legal restrictions (e.g., speed limitations), or to the steal of intellectual property (the copy of commercial software stored in the unit).

[*]STmicroelectronics, Via Remo De Feo, 1, 80022 Arzano (NA), Italy, *eduardo.ciniglio@st.com*

[†]Dip. di Ingegneria, Università del Sannio, via Traiano 1, 82100 Benevento, Italy, *epmancini@unisannio.it*

[‡]Dip. di Ingegneria, Università del Sannio, Piazza Roma 21, 82100 Benevento, Italy, *villano@unisannio.it*

Furthermore, non-skilled operators, such as the vehicle customers, have to be prevented from updating the system with not fully-compatible software.

In light of the above, trusted flashing (the secure writing of new *flashware* in the flash memory of the ECU) is a priority for ECU designers [10]; it involves the addition of the concept of trust to the complete delivery process, from development to the management of the flash memory by the final operator.

Taking also into account the errors that may arise during the flashing, the following security goals have been pointed out for the process [6]:

1. error detection: the ECU must be able to check if the flashware is corrupted due to transmission problems;

2. authenticity: it should be guaranteed that the flashware comes from a legitimate source;

3. copy protection: the flashware should be copied only on a determinated ECU;

4. confidentiality: no unauthorized party should be able to read the flashware;

5. authorization of the external programming tool (*diagnostic tester*) towards the ECU: the diagnostic tester should be suitably recognized by the ECU, in order to decide if the flashing has to be permitted or not.

This paper is essentially concerned with the last point, i.e., with the methods that can be used to authenticate the diagnostic tester and/or the flashing operator in order to prevent unauthorized modifications (or even unauthorized reading) of the ECU software and data. It should be noted that only recently this issue has been raised, proposing sufficiently robust solutions. As a matter of fact, a common authentication method employed in present-day ECU is based on the use of a digital comparator. This tries to match a password stored in flash memory with the one that can be provided by the diagnostic tester through a serial interface. If the passwords match, the flashing operations are enabled. This protection method is simple and un-expensive, but it can be easily circumvented by brute force attacks. In fact, it is relatively simple to supply to the ECU the complete sequence of keys until the valid one is found, even if this may require a long time. Once the protection is broken, the key can be successively distributed using illegal channels.

In order to prevent brute-force attacks, the above mentioned document [6] defines a security procedure, to be implemented by a *security module* in the ECU software. The diagnostic tester authenticates by requesting a *seed* from the ECU, and computing a known function of the seed. The ECU computes the same function of the seed as well. If the two computed keys match, the authorization is successful and the ECU is "unlocked". This procedure prevents the use of brute-force key guessing, provided that the seed is sufficiently random and, of course, that a "good" function is used for computing the key from the seed. However, the choice of the function seems to be left to the OEM [6].

It is common belief that the processing power and memory space available in a ECU do not lend themselves to the use of the time- and space-expensive public key cryptographic algorithms. As we think that this is not necessarily true with state-of-the-art devices, we will try to use for authentication a much more complex (and robust) method, based on the use of RSA signatures, as defined by the PKCS#1-standard [5]. We have chosen RSA as it is more widely used than ECC [4], which could be preferred because of smaller key sizes and possibly lower computational cost. The objective is not only to present a robust process to secure the access to flash memories with no additional hardware, but, more in general, to test whether the complexity of public key cryptographic algorithms is compatible with the features of present-day ECUs. The implementation that has been developed makes it possible to enable and disable selectively read/write/erase operations, with the further possibility to restrict the authorization to one or more (factory-defined) subsystems of the ECU.

This paper is structured as follows. In Section 2 the related work is described. In Section 3 the reference scenario is presented and in Section 4 the proposed authentication system is discussed. Then Section 5 presents the results of our tests as far as computing and memory requirements of the authentication software are concerned. Finally, in Section 6, the conclusions are drawn and future work is pointed out.

2 Related work

An effective protection of the flash devices can be implemented in hardware, at the flash memory chip level [2]. In this solution, the flash memory integrates a RSA module, a signature engine and a random number generator in order to prevent unauthorized modifications of the memory content. However, this hardware implementation is relatively complex. This is the reason for the most common choice in automotive ECUs of different approaches, based on cryptographic software to be executed in the ECU processor, with no additional hardware.

Security topics in the automotive field are widely dealt with in the literature. Weimerskirch, Wolf, and Wollinger describe the state of the art in [11]. They describe the possible attacks for automotive software and hardware, and the constraints for security systems. Then they present the mechanisms to protect the hardware, the software and communications, along with some critical applications.

Adelsbach *et al.* [1] propose a method for secure delivery and installation of software for embedded systems. The suggested technique involves the use of both Public Key Broadcast Encryption and Trusted computing. In particular, Trusted Computing requires additional hardware, and so the ECU must be designed to support it. Our proposal, instead, wishes to enable secure read and flashing with no additional hardware.

The OEM Initiative Software (HIS) proposes several solutions for flash security. The specifications of the software module (the *flashloader*) used to reprogram ECUs based on flashable microcontrollers are reported in [7] and the underlying software security module in [6].

Stephan *et al.* [10], instead, focus on the secure flashing process. They identify and describe the four steps of flashing process (the development, the provisioning, the distribution and the flashing), and show how a Security Management Center can support each step. The Security Management Center is the entity that manages the cryptographic services. They introduce also the security classes reported by the HIS in [6].

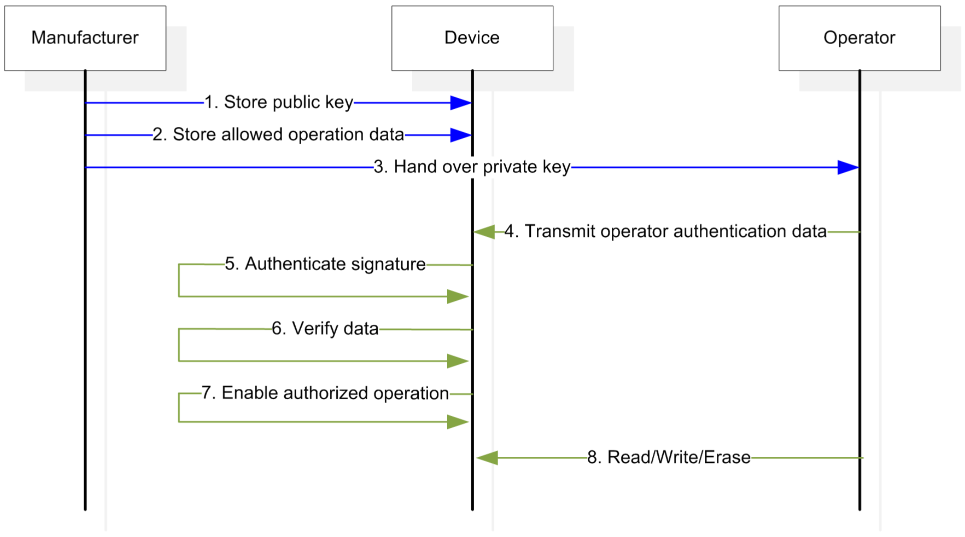
3 The authentication scenario

Public key cryptographic algorithms, also known as asymmetric key algorithms, use a pair of different, but related, keys to encrypt and decrypt data. In the basic encryption scheme, the first one, named private key, must kept secret, and should be used to decrypt the data previously encrypted with the public key, which can be distributed freely. The encrypted data cannot be decrypted with the same key used for encryption, and the knowledge of the public key is not sufficient to encrypt data as with the private key.

Public key cryptography is mainly used for encryption (to ensure confidentiality) and for digital signatures (to ensure authenticity). Here we are concerned with the latter point, as our objective is to authenticate the diagnostic tester of the ECU and its operator. In a signature scheme, the private key is used to sign a message; anyone can check the signature using the public key.

Our authentication system has the objective to enable some actions only to the operators that hold the respective rights. Hence, as a first step, it must properly recognize such operators by authenticating them. After a successful authentication, it can (partially or totally) unlock the device, making it possible to perform read or flashing operation on a range of flash addresses. For simplicity's sake, we have identified only two actors in the reference scenario (Fig. 1): the ECU manufacturer, who designs and builds both the device and the software, and the operator, who wishes to perform some actions on the ECU through a diagnostic tester connected to the device.

In our scenario, for each operation, defined by the pair (*type of operation*, *address range*), where *type of operation* can be *read*, *write* (flashing), or both,



Manufacturer
Device
Operator
1. Store public key
2. Store allowed operation data
3. Hand over private key
4. Transmit operator authentication data
5. Authenticate signature
6. Verify data
7. Enable authorized operation
8. Read/Write/Erase

Figure 1: The reference scenario

generates a public and a private key. The public key is stored together with the data specifying the allowed operation in the device (1, 2). The private key is handed over to the legitimate operator (3). In order to authenticate himself, the operator generates a signature using his private key (4). This is transmitted (together with information on the operation to be authorized) by the diagnostic tester to the ECU, which independently authenticates the signature using the public key corresponding to the requested operation (5, 6). If authentication is successful, the device is unlocked for the requested operation (7, 8), until it is finished or the device is reset.

It should be noted that:

- it is very unlikely that a correct signature is generated without the private key;
- inside the device, only public keys are stored. Even if they are read, by means of an application code exploit, for example, security is not compromised, as only the *private* key can generate valid signatures;
- there is no security leak if an operator makes its private key publicly available, as the key is in any case tied to the particular operation allowed to that operator, and, above all, the key is useless on any other device (where a different public-private key pair is used).

In this scenario, the only performance-critical point is the authentication (5), which, in our approach, must be executed using solely the computational resources of the ECU.

4 The authentication system

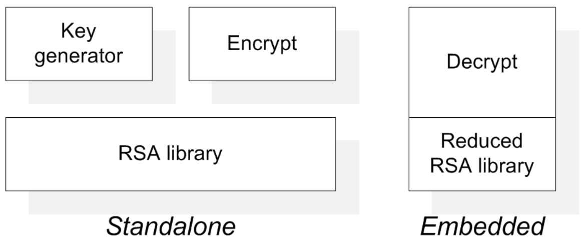


Figure 2: The system architecture.

The implemented authentication software is made up of two crypto-modules, as shown in Fig. 2. The first one is standalone, in that it does not run on an ECU, but on a host computer, being used by the manufacturer to generate the unique pair of keys that are associated with every operation to be enabled/disabled. As is clear, there are no particular performance requirements for this module. The second one is instead to be executed in ECUs, to perform the authentication of the flashing operator and to trigger the "unlocking" of the device. The performance of this module is critical, due to the lack of memory space and low processing power of the target devices. The implemented authentication module has been designed to have as small a footprint as possible, and to be computationally efficient using processor-specific instructions.

One of the most widely used asymmetric cryptographic algorithms is RSA. A deep analysis of RSA can be found elsewhere [3, 5, 8]. In the next paragraphs we will just outline the algorithm. To compute the two keys used by RSA, a pair of random prime numbers, p and q has to be chosen. These can be used to compute the product *n=pq*. Another random number *e*, should be chosen prime with the product *(p-1)(q-1)*. The pair $\{n,e\}$ is the public key. The private key, *d*, can be computed by the equation $d = e^{-1} \pmod{(p-1)(q-1)}$. Our software generates the keys both as text and as constants to be included in C files. The quality of the keys depends on how random the numbers p and q are. The implementation on which our code is based (Havege, [9]) exploits the uncertainty introduced in the internal states of the processor by external events to produce high quality random numbers.

RSA, unlike other digital signature schemes, has many similarities to an encryption scheme. So it may be worth to recall briefly that c_i, the encrypted version of the message m_i, is computed by the equation: $c_i = m_i^e \pmod{n}$. It can be decrypted by computing $m_i = c_i^d \pmod{n}$. The use of large keys implies that such computations cannot be performed with the usual hardware, and so the implementation has to support variable length arithmetic. This slows down considerably the encrypting/decrypting operations.

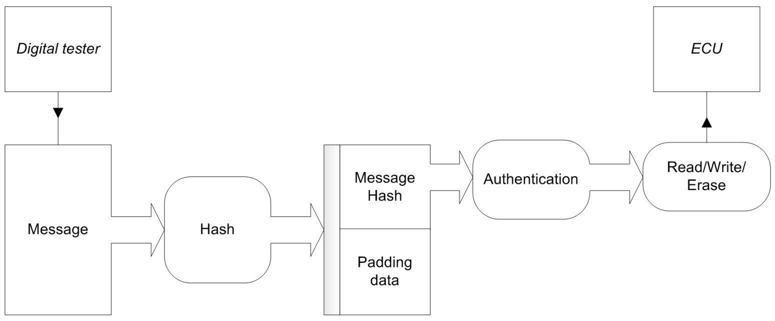


Figure 3: The phases of authentication

One of most used signature schemas is the one with appendix. This consists of a signature generation operation and a signature verification operation. The signature generation operation produces a signature from a message with a signer's private key, and the signature verification operation verifies the signature on the message with the signer's corresponding public key. To verify a signature constructed with this type of scheme it is necessary to have the message itself. It is important to point out that in the RSA scheme *the signature generation and verification operations are essentially the encryption and decryption primitives* [5]. Hence the performance analysis of the authentication schema is also valid for RSA encryption/decryption operations.

The signature generation operation applies a message encoding operation to a message to produce an encoded message, which is successively converted to an integer message representative. A signature primitive is applied to the message representative to produce the signature. On the other hand, the signature verification operation applies a signature verification primitive to the signature to recover a message representative, which is then converted to an encoded message. A verification operation is applied to the message and the encoded message to determine whether they are consistent. The encoding method can be deterministic (e.g., EMSA-PKCS1-v1_5) or randomized (e.g., EMSA-PSS) [5], but these details are out of the scope of this paper.

For our purposes, it worth mentioning that the message is a structure containing the ECU code, the operator credentials and the details of the flashing operation to be authorized (type of operation, address range). The ECU code is redundant, as the private key handed to the operator is uniquely linked to the ECU. Similarly, the details of the flashing operation are not really necessary, as they are permanently stored in the ECU ROM along with the private key to be used. On the other hand, the operator credentials could be used to implement a higher security level, restricting operations to some class of operators, even if in possession of the valid private key.

In any case, the phases of authentication are those shown in Fig. 3. It is worth pointing out that authentication requires the use of an encryption operation at the operator side. The authentication data transmitted to the ECU are to be decrypted for authentication purposes at the ECU side. Due to the padding of original message, the length in bytes of the data encrypted data will be equal to the key length chosen [8].

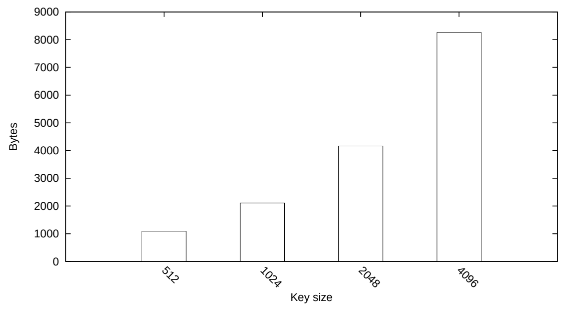


Figure 4: Dynamic memory requirements

5 Memory usage and performance analysis of the embedded module

Our authentication system has been extensively tested, both on embedded devices and on standard PCs, to evaluate the performance using different key lengths. For the reasons mentioned in the previous section, authenticating the operator at the ECU side requires the decryption of a block of data whose size is equal to the key size. The results that we will present here have been obtained on an ARM Cortex M3, a Qualcomm 7200 and an Intel Xeon processor. The first processor is available in a low cost ST development system, the STM32-PerformanceStick. This provides a Cortex-M3 CPU running at 72 MHz with 128 KB of Flash and 20 KB SRAM [12]. The second test device was a Qualcomm 7200 running at 400MHz, under the Windows Mobile 6.0 operating system. The last test device was a standard PC with a XEON CPU running at 3.2 GHz under Windows XP. It is clear that the only platform of interest in the automotive field is the ARM Cortex, which is widely used for embedded applications. The results on the other two processors, which are mainly used in hand-held and desktop devices (for the Qualcomm 7220 and the Xeon, respectively), are provided just as a reference.

The dynamic memory requirements of the embedded authentication module (i.e., the one that has to be executed in the ECU) are presented in Fig. 4. In addition, the authentication framework also needs about 8 KB to store

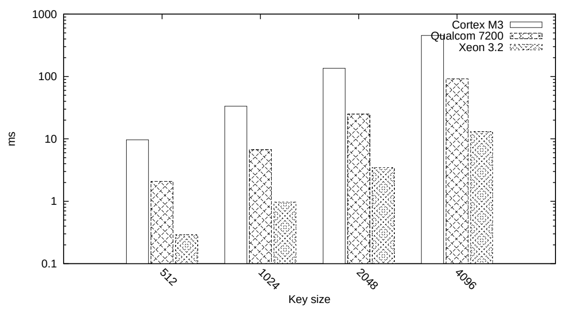


Figure 5: Execution times

the compiled code (in the case of the Tasking C compiler for Arm Cortex M3). Of course, different compilers may require different amounts of memory. This difference may increase for different architectures (e.g., RISC vs. CISC, 32 bit vs. 64 bit, ...). Clearly, the implementation of additional functions requires more memory. Another important point is the evaluation of the the variations of decryption time as a function of key length. A longer key is more secure, but leads to higher response times and to larger memory requirements.

Key	**Cortex M3**		**Qualcomm 7200**		**Xeon 3.2**	
bits	*ms*	*cycles* (10^6)	*ms*	*cycles* (10^6)	*ms*	*cycles* (10^6)
512	9.66	0.70	2,08	0.62	0,29	0.83
1024	33.46	2.41	6,68	2.00	0,97	2.67
2048	135.72	9.77	24,98	7.49	3,48	9.99
4096	455.70	32.81	92,00	27.60	13,07	36.80

Table 1: Performance data in ms and clock cycles

The response times presented here for a variety of computing platforms are the times required for decryption, and do not include any message transmission or delivery time. They are shown in Tab. 1 and in Fig. 5. As was to expected, the response time trend, as a function of key size, is similar in different architectures. On the other hand, the response times differ heavily for each testbed. To make a better comparison on different architectures, we have also expressed the response time figures in terms of clock cycles. The results are shown in Fig. 6. In fact, the use of the number of cycles as performance index shows that there is no great difference between the three processors, and that the long response times of the Arm Cortex M3 are due to its slow operating frequency.

Whether the memory requirements of the crypto-libraries used for au-

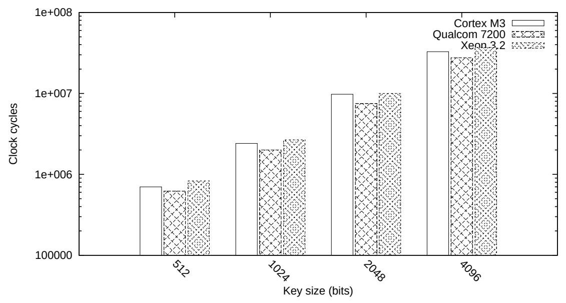


Figure 6: Performance analysis in clock cycles.

thentication are compatible or not with embedded automotive applications, depends on the footprint of the memory available in the ECU. 8 KB for data and about 8 KB for code (the memory requirements for a 4096-bit key) are not too much, after all. On the other hand, 455 ms (the time required to decrypt a block in the case of a 4096-bit key) are decidedly a long time. In the case of authentication, this time penalty has to be paid only one time, and it is likely to be acceptable. In the case of encrypted communications between ECU and diagnostic tester, the delay is not tolerable. At the state of the art, low-frequency CPUs such as the Cortex M3 cannot yet afford long key lengths. However, the use of the CPU-eager RSA cryptography with shorter keys is certainly feasible at the state of the art.

6 Conclusions

In the automotive field securing the access to data is a challenge. Intellectual properties steals, or malicious tuning of vehicle parameters should be prevented. New market paradigms, such as value-added software vending, make this a very important point.

In this paper, we have presented an authentication system that can be used for embedded devices in the automotive field, to enable only authorized flashing operations. Our system is based on the authentication procedure of the RSA standard, and can run in embedded devices without any additional hardware.

It is widespread opinion that RSA authentication is too complex for embedded devices, which are typically provided with simple and rather slow CPUs and small memory footprints. We have measured the performance of our implementation on various architectures, finding that memory requirements are not too high, but that long keys (e.g, 2048-4096 bits) require a processing power inadequate to low frequency ECU processors. As the authentication is based on RSA encryption/decryption operations, this results is also true if the RSA standard is chosen to provide encrypted communication between diagnostic tester and ECU. However, we believe that the use of the standard and of asymmetric key cryptography is currently possible, if shorter keys are used.

Currently we are porting our code to ST Power Train & SafetyProducts, which are based on the Power PC Architecture. We expect that this family of processor will enable the use of longer keys, which is prohibitive with most CPUs available currently in ECUs.

References

[1] André Adelsbach, Ulrich Huber, and Ahmad-Reza Sadeghi. Secure software delivery and installation in embedded systems. In *Embedded Security in Cars*, pages 27–49. Springer Berlin Heidelberg, 2006. `http://www.springerlink.com/content/m848u3w345776k16/`.

[2] Tiago Alves and John Rudelic. ARM Security Solutions and Intel Authenticated Flash, 2007. available on-line.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[4] Darrel R. Hankerson, Alfred J. Menezes, and Scott A. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, 2004.

[5] RSA Laboratories. PKCS 1: RSA cryptography standard, 2006. `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf`.

[6] Thomas Miehling, Burkhard Kuhls, Heiko Kober, Hartmut Chodura, and Marcus Heitmann. HIS Security Module Specification, version 1.1. Technical report, Daimler Chrysler AG, 2007. `http://www.automotive-his.de/download/HISSecurityModuleSpecificationV1.1.pdf`.

[7] Thomas Miehling, Pavel Vondracek, Martin Huber, Hartmut Chodura, and Gerhard Bauersachs. HIS Flashloader Specification, version 1.1. Technical report, Daimler Chrysler AG, 2006. `http://www.automotive-his.de/download/HISFlashloaderSpecificationv1.1.pdf`.

[8] Bruce Schneier. *Applied Cryptography, Second Edition: Protocols, Algorthms, and Source Code in C.* Wiley Computer Publishing, 2 edition, 1996.

[9] André Seznec and Nicolas Sendrier. HAVEGE: a user-level software heuristic for generating empirically strong random numbers. In *ACM Transaction on Modeling and Computer Simulations (TOMACS)*, volume 13, pages 334–346, october 2003.

[10] Winfried Stephan, Solveig Richter, and Markus Müller. Aspects of secure vehicle software flashing. In *Embedded Security in Cars*, pages 17–26. Springer Berlin Heidelberg, 2006. `http://www.springerlink.com/content/pk72544uvnp15t63/?p=493bfbc3b7b2481c8ac5c755e6f543ca&pi=1`.

[11] A. Weimerskirch, M. Wolf, and T.Wollinger. State of the art: Embedding security in vehicles. *EURASIP Journal on Embedded Systems, Special Issue: Embedded Systems for Intelligent Vehicles*, 2007. `http://www.hindawi.com/GetArticle.aspx?doi=10.1155/2007/74706`.

[12] Joseph Yiu. *The Definitive Guide to the ARM Cortex-M3.* Newnes, August 2007.