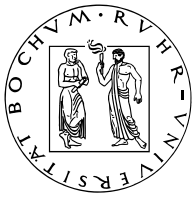
Cryptographic Protocols for Component Identification and Applications

Katrin Höper

5. September 2002

Ruhr-Universität Bochum



Communication Security Group

Prof. Dr.-Ing. Christof Paar

Erklärung

Hiermit versichere ich, dass ich meine Diplomarbeit selbst verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Ort, Datum _____

Abstract

The main contribution of this work is to provide solutions for component identification in composite technical systems such as automobiles or airplanes. We focused on the following three goals: (1) piracy protection to identify original parts and detect bogus parts; (2) system protection to monitor a system in order to ensure system integrity; and (3) theft protection to prevent the usage of stolen components. There are applications in a variety of fields, such as automobile, aviation and telecommunication industry. For example, our work is able to ensure that cell phones only work with original batteries, or that air planes do not start when bogus parts are built in. Our solutions help to gain profit for manufacturers and ensure quality of products and services for consumers.

We attempted to treat the problem of component identification comprehensively in this thesis. We hope that the thesis can serve as a reference to implement cryptographic protocols to achieve one or more of the above mentioned goals for all kinds of applications. After a motivation for component ID systems we develop a variety of solutions to meet the requirements of different systems. We consider protocols using different encryption schemes, such as symmetric and asymmetric encryption, as well as suited protocols for different kinds of network architecture, e.g., server-client and peer-to-peer networks. Furthermore, we introduce parameters to customize the solutions for individual needs.

Contents

List of Figures

List of Tables

1 Introduction

Today's electronic systems usually consist of numerous replaceable components. These components become more and more networked and communicate among each other. The networking of all components and systems induces the need of special protection and the use of security features. It turns out that the identification of single components is the main goal to ensure. The component's identity forms the basis for achieving all further security features. The main concept of this work is to assign all components a unique identifier which can be electronically verified. By providing identification mechanism we can derive piracy, system and theft protection as well. Piracy protection enables to distinguish between original and bogus parts. System protection ensures the system integrity, and theft protection prevents the use of stolen components in another system.

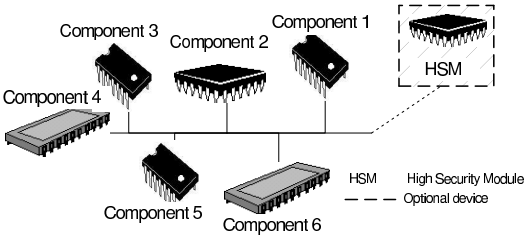


Figure 1.1: Composite off-line system with networked components

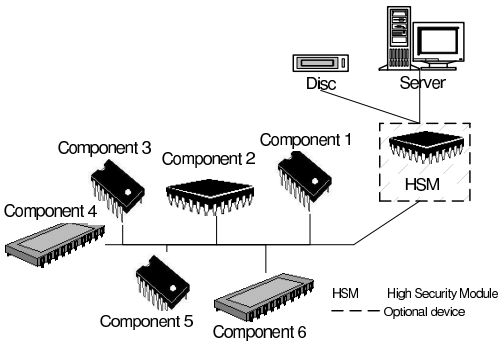


Figure 1.2: Composite on-line system with networked components

An example of a composite off-line and on-line system with networked components is illustrated in Figure 1.1 and 1.2, respectively. We now consider an automobile as suited system. Within a car the brakes, injection pump, sensors and the on-board computer are talking to each other. In the near future most of the systems will be hooked up to the Internet and it will become possible to communicate via the Internet with these systems. The automobile industry is a precursor for this innovation.

As presented in [1], the next generation of cars will come with microprocessors to provide GPS, Internet services such as web portals and email servers, and telematic services, as the recall of traffic data or availability of parking lots. The next step is then the wireless communication, e.g. by the use of Bluetooth components. This has been already considered for cars, e.g., as presented in [2] and [43]. The cables of the data bus would be therefore replaced by radio links. Besides the communication and navigation electronics of automobiles, there are many application areas, such as the electronic systems of airplanes, other kinds of vehicles, cell phones, computers, laptops, palmtops, special machines e.g., construction machinery, and many more, which consist of numerous networked components.

1.1 Motivation

Counterfeits achieved almost perfection over the last years and are hard to distinguish from their originals even by professionals. According to [51] the transaction volume of bogus parts totals approximately 60 billion US$ or 2-5% of the world trade volume. This does not only cause financial damages but also the health of consumers is in great danger, e.g. as demonstrated by the below-mentioned airplane accidents. There are many reasons for implementing systems providing piracy and/or system and theft protection. From the view of the manufacturer and component supplier this is mostly of financial kind. For the consumer it ensures good quality, provides special services and protects their health.

Many air plane crash accidents are suspicious to be caused by bogus parts, for instance the American Airlines Airbus crash in New York City in November 2001, which killed 265 people, the Minerva Airlines Dornier 328 which overshot the runway killing four people in February 1999, the ValuJet Flight 592 which crashed in the Florida Everglades and killed 110 passengers in May 1996, the ATI ATR 42 crash near Milan killing 37 in October 1987. According to [12] bogus parts, including fakes, used parts sold as new or refurbished, and new parts sold for unapproved purposes, have found their way into the inventory of every major American commercial airline, not even Air Force One is exempt. The present procedure to check parts for their originality is by certificates and tags which come with each component. Besides many procedural weaknesses there is a worldwide operating mafia selling counterfeits with forged papers and also the tags can easily be bought on the black market. Referring to [12], the selling of substandard replacement parts is assumed to be

even more profitable than trafficking in illegal drugs. The interested reader can find more about airplane accidents caused by bogus parts, the lack of security and the organized crime taking advance of procedural weaknesses in [12, 44, 8, 9, 39, 18].

Another main area of counterfeits is the automobile industry and their component suppliers as described in [15] and [7]. Counterfeits exist for almost every part, e.g., brake pads, stop lights, rims, crank shafts, oil filter etc.. There are even some garages which offer cars, e.g. the Ferrari GTS, entirely built of cloned parts. Not even the race cars of Formula one are safe of bogus parts, as proven by the damage of Mika Häkkinen's gearbox during the race in San Marino in 1998, which was caused by the forged ball bearing as stated in [15].

Besides the aviation and automobile industries many other branches of industry are affected as well. The reader is referred to the following newspaper reports to read more about forged phone cards [52], cell phone accessories [53], SIM cards [25], Xbox games [26], ink cartridges [24, 13, 14], etc..

As mentioned above the complex networking of components in technical systems already exists and progresses quickly. Many applications are conceivable. As in all systems the availability, authenticity, integrity and confidentiality of the data should be ensured. The three major properties presented in this thesis are piracy protection, system protection, and theft protection. Piracy protection provides the ability to identify original parts and the possibility to detect counterfeits and cloned components. System protection provides system integrity by monitoring systems for unauthorized changes. This feature can be combined with piracy protection to ensure that a system consisting of original parts only cannot be altered in an illegal way. This combination provides lasting piracy protection since the system is guaranteed to consist of original parts permanently by itself. Also a protocol for system protection by itself would be reasonable. Once system protection is provided, implementing theft protection is easy. We are looking for a solution which provides piracy protection without the need of additional tools or ancillary equipment, and which can be easily used, i.e., without any additional costs or special knowledge, by individual users. The components to be verified should be checked within the system in which it is supposed to be used. Consequently we are considering a system in which all components to assembled hold authentic data which can be verified via a data bus to which all claimants and at least one verifier are connected.

1.2 Outline

The following chapter describes previous work in the field of piracy, system and theft protection. Registered patents and existing implementations are considered and their advantages and disadvantages are discussed. We outline what we can adopt from already existing scenarios and how we can avoid the typical shortcomings of existing implementations.

Chapter 3 introduces the foundations of identification protocols. The basic terms are defined, objectives and attacks are described and finally some basic protocols are presented.

In Chapter 4 we consider categories of the later presented solutions. Three different scenarios are introduced which provide piracy protection, system and theft protection or a combination of all three security properties at once, respectively. The characteristics of suited systems as well as many suited applications are given for each solution.

Chapter 5 defines all necessary terms and introduces required parameters and assumptions. General system categories and the general objectives of suited systems are presented which are followed by the design criteria of all developed protocol.

In Chapter 6, 7 and 8 the protocol solutions for the three problems of Chapter 4 are presented, namely the protocol of the *system check*, the *proof of origin* and the *proof of origin and system check*. The implementation of the *system check* is presented for the use of symmetric encryption only, whereas the following protocols are described for both, the symmetric and the asymmetric method. In each of the solutions the participants which might take part in the protocol are introduced. The second section deals with the necessary preparations of the system and of the participants before the protocol is executed the first time. That section is followed by the actual protocol flows. The entire protocol for one solution is divided into three parts, which represented the three different periods of life of a component within one system, namely the *assembly*, the *running system* and the *disassembly*. All procedures which need to be executed during the particular periods are introduced in single sections. The description of the procedures compromises all additional[1] parameters to be set and assumptions to be made, the protocol flow in generally and at least one example of a realization. After the basic protocols some features which could be additionally provided are presented. In the case of the

[1] additional to the general ones introduced in Chapter 5

proof of origin and the *proof of origin and system check* the chapter finishes with a comparison of the symmetric and asymmetric solution.
Chapter 9 deals with possible enhancements and improvements of the presented protocols. The use of zero knowledge identification protocols as replacement for challenge-response protocol is briefly considered. The possibility of the use of threshold cryptology and one-time signatures for the protocols presented later are considered to improve the security or the speed of execution, respectively. One section considers the similarities between the presented protocols and the security model developed in [41]. The last section deals with key hierarchies which might be useful in an implementation. This could be, for instance helpful in complex systems consisting of many different groups which all hold different levels of authorization. An example is an automobile, with the owner, the dealers, the manufacturer and the mechanics of authorized or independent garages. All these groups require different authorizations and have different interests.
In Chapter 10 the security issues of the presented protocols are considered. The feasibility of attacks, the ones introduced in Section 3.3 and additional scenarios, on the presented protocols are considered. The consequences of compromised data is discussed in the following. We consider then key aspects as well as issues of tamper resistance. The next section considers the consequences of successful attacks on the protocols. The last section deals with constraints due to the technical environment, e.g. by limited energy sources or computation power or the used data bus. As an example the environment of a vehicle is considered in general and the CAN bus in particular.
Chapter 11 summarizes all presented solutions and gives some hints for future work on this subject.

2 Previous Work

Before developing new solutions for providing piracy protection, theft protection and system protection we take a closer look at some approaches described in specifications and at currently used systems. We discuss the advantages and disadvantages of each work and explain why currently used solutions fail are pointed out. Some of the solutions presented in the following three sections are satisfactory for providing one or maybe even two of our three security goals. A solution combining all three goals is still missing. It has to be kept in mind that we are looking for a general solution suited for systems with several removable components, whereby all components are able to communicate with each other. Furthermore, we are looking for a solution where all components can be electronically verified.

2.1 Piracy Protection

Piracy protection has a long history. For a long time people have used seals to authenticate important documents. The seals were uniquely associated to their owner, e.g. companies, notaries and even nations, and hard to copy. Since then the use with documents has become less important, but seals are still used for tamper-evidence. Nowadays products are protected by stickers or labels attached to the product, or marks are embedded in the products themselves. Three levels of security attributes can be distinguished: The first-level-category comprises all visible security features which can be easily seen by the user. This could be for instance the use of security printing as used for protecting bank notes. Another variant is the use of holographic stickers or of other optical variable devices (OVDs) which might be attached to the product. For security printing techniques, which also include watermarks, we refer to [5]. Another possible realization is the etching of serial numbers into a substrate. The possibility of a computer based evaluation of those IDs is introduced in [36]. Holographic stickers are widely used because they are hard to copy or to imitate. The optical diffractive structure will be destroyed when the

sticker is removed from its original place and thus the fraud will be detected. In [45] an apparatus is introduced to identify whether products pasted with holographic stickers are genuine or not.

Security features of the first level are often combined with solutions of the second and third level. Security features of the second-level-solutions can only be visualized by the use of special means, e.g. by labels which are only visible under ultraviolet or infrared light. The German national print office of Germany ("Bundesdruckerei") holds a patent for granulate material which can be added to all kinds of synthetics to uniquely mark it. This mark is reflecting under ultraviolet light and thus products can be easily checked for their genuineness.

The third-level-category features a high-security level. These features make use of physical, chemical and biological properties of materials and usually cannot be detected by users. A popular approach is the use of DNA as unique identifier. A DNA security stamp system was invented by the national print office in cooperation with the *november AG*. The synthesized DNA is mixed into special ink which is usable with regular stamps. The originality of the imprint is then verified by the use of a felt-tip pen which contains the matching DNA string, whereby the result can be visualized by a scanner. Another solution based on DNA uses so called DNA-labels. The *november AG* registered a patent for this application. They invented labels on the basis of nucleic acid, so called *barcode DNA*, whereby the DNA consists of two complementary DNA strings. One is used for encrypting the information and the other to read out the information. The labels can be pasted on everything and the originality verified everywhere by the use of a laser. For more realizations, patents and information especially about high end seals and sticker we refer to the homepage of the national print office [11] and the homepage of *Verpackungsrundschau* [50], a German magazine which deals with secure packages.

Most modern realizations used for providing piracy protection are not easy to imitate or clone. Even removing the stickers or labels and sticking it on counterfeits is not working anymore with the use of high-end OVDs. Unfortunately most applications are still using old technology and are thus prone to frauds. There still remains the possibility to buy labels and tags on the black market. According to [12] the tags which authenticate fixed parts of airplanes can be bought for $100 on Miami's black market. We can almost be sure that this is not the only case.

All presented solutions have a drawback in common as the verification process cannot be automatically executed within large systems as human interaction is required.

None of the implementations would be suited for protecting systems with a great number of components, for instance among many parts of an airplane, since all parts have to be checked of high expenses. Additionally most of the security features cannot be verified by a consumer. All items have to be verified separately with the use of an additional apparatus. Two realizations using an apparatus are registered as patent [37, 35]. In the first realization all components hold a secret and a public ID. The verifier proves the identity of the claimant by a challenge-response protocol. The verifier does not need to know the claimant's secret a priori, in fact it computes the secret by a secret manufacturer key and the public ID. However, the verifier must be a server. This leads to a single point of failure and requires an on-line system with permanent access to the server. The invention does not deal with cloning. The server in this realization cannot distinguish between an original part and its clone. The second invention deals with validity checks of components. It also uses challenge-response protocols, and all components hold a secret key. All components need to share a secret a priori though. The issue of clone detection is also not looked at.

Finally we introduce two case studies which are presently used for providing piracy protection. The first one is a patented system of *Epson* to avoid the use of forged ink cartridges in printers. Secondly the Xbox from *Microsoft* is presented, which should only accept original games.

Epson invented an ink cartridge with an attached chip which is supposed to prevent the re-fill of original cartridges and the use of cheap ink cartridges of other brands. The chip counts the printed pages and tells the user after a certain number of prints to change the cartridge independent of the amount of the remaining ink. The number of allowed prints is fixed. Thus the user cannot simply re-fill the cartridge with cheap ink. In addition, that only cartridges holding the patented "counter chip" can be used with the Epson printers. But the designer of Epson have not expected the inventiveness of some users. As described in [13] a company invented a device which sets the counter of the chip on "full" again. The device comes with refillable ink cartridges. The user has now to remove the chip of the original cartridge and stick it on the refillable one. Every time the ink is empty the cartridge can be refilled and the counter is reset. But the worst is yet to come, there is another way to escape the restrictions which does not need any extra devices or special tools. This attack simple uses a security hole in the protocol [14]. The counter of the chip is only veri-

fied when the printer is turned on, or in case of an "official"[1] cartridge exchange. In all other cases the last counter reading, which was buffered in the printer's memory, is used to overwrite the current one in the chip. This enables the following trick. A full cartridge is used for a print job and thus the counter reading of this full one is buffered. Then the user exchanges the full cartridge with a refilled one (note that the counter reading of this chip is "empty") without turning off the printer. When a new print job is executed, the last counter reading of the full cartridge is written over the old ("empty") one of the refilled cartridge. The cheater needs only to keep one ink cartridge with a full counter reading which comes with the new printer anyway.

Microsoft's Xbox is a game console on which only authorized games are supposed to be executable. According to [26] it took an MIT student three weeks to hack the Xbox. He compromised the security key which is used for the authentication of the games. With the knowledge of the key it is possible to run any software on the box. Due to its high transfer rate the used hypertransport protocol for transmitting the key was assumed to be an adequate protection from eavesdropping. Hence the secret key is transmitted in plaintext. The assumption proved to be wrong by Andrew Huang, who used a Xilinx Virtex-E FPGA to intercept the 200MHz clocked DDR signals to obtain the key.

We have learned from the presented inventions and implementations that the security features should not be removable from the component (regarding stickers and the Epson chip). Secondly the protocol should be designed carefully (regarding the Epson chip and the unencrypted communication in the Xbox protocol). A well protected component should be hard to clone including its attached or embedded security feature. Finally a solution should not be based on poor assumptions (environment can change) and not require too much equipment (regarding server, on-line solutions and solutions which needs additional apparatus and devices).

2.2 System Protection

System protection ensures system integrity or entity integrity of particular components. One possibility to provide system or entity integrity respectively, is to use

[1] by pressing the particular button on the printer

tamper resistant components only. Tamper resistance ensures that a system or entity cannot be modified by unauthorized persons or entities, but according to [40] it is almost impossible to implement it without vulnerabilities. Thus it is more usual to use tamper-evident components, for instance by using special seals. The attacker is able to violate the component's or system's integrity but leaves a trace of this activity. Thus special seals, tags and stickers are used, as described in the previous section, to protect the system or a single component. Besides the OVDs, also introduced in the previous section, another special label is used to provide tamper-evidence, namely the so called void-labels. When removing those kinds of labels an individual emblem or writing will remain on the surface. This emblem or logo is visible on the surface as well as on the label itself.

The mentioned solutions all use physical mechanism to provide system protection. We are looking for a solution using digital data used as authentic characteristic and communication between the components and the verifier for the verification process. Three registered patents will be presented in the following which all implement system protection in systems where all components are connected to each other and are able to communicate.

The first embodiment [47] is suited for vehicles where all components are connected via a bus. Each component holds a unique ID, and a central controller holds a list of all IDs within the system. During a system check, which is either executed when the vehicle is started or periodically in the running system, the controller compares all components' IDs with the ones on its list. In the case that two or more corresponding IDs do not match, the controller sets an alarm and/or disables the components which failed. The IDs are derived from the serial number of the car and stored in the EPROM of a component. The key is zero in new components and is set during its assembly into the vehicle by qualified personnel. The key memory cannot be manipulated and readout guaranteed by an anti-tamper feature. There is a single point of trust and thus a single point of failure due to the use of a central controller. Besides that components can be exchanged or re-setted by authorized personnel only. From latter follows that every time a vehicle's component needs to be fixed the owner has to go to an authorized garage. If the owner of the vehicle wants to resell a component of her car she has to visit an authorized garage for re-setting the component in order to enable the legal use of this component in another system. Thus this solution is not user friendly. No encryption is mentioned for the communication between the components and the controller. Thus an attacker could

easily eavesdrop the authentication step in order to obtain a valid ID to set up his counterfeit and then built it into his car without being detected.

The second embodiment [32] is supposed to prevent illegal assembly and disassembly of components of a vehicle. All components are specifically initialized for each vehicle during the assembly of the parts. The system only runs properly if all components in the system and the vehicle share the same secret. If one component's key is detected not to match the other ones during a check measures are taken, e.g. by disabling the failing parts, and/or setting an alarm. Several realizations are described in detail, e.g. in case of an alarm the manufacturer is informed and the failed components can only be re-enabled by authorized personnel. The realizations are fully developed and many different scenarios, e.g., for sanctions in the case of failure, or implementations without a server are considered. However, there are some drawbacks. The manufacturer holds a secret list of all vehicles. Access to this list is needed every time a new component is built into the system. It follows that on-line access is required for each assembly and can only be executed by specialized personnel of an authorized garage. Additionally the privacy of the vehicle's owner is endangered since the manufacturer is keeping track of all vehicles. The actual implementation of the protocols, the use of encryption and a possibility of resell of components are not mentioned at all. The embodiment is not user friendly since actions can only be executed by authorized personnel and since all data is stored by the manufacturer.

The third embodiment [33] deals with a group of components of a vehicle forming a system. Each part is assigned a unique code. Two scenarios of systems are considered, a system with a server and one without. In the first scenario the server holds a list of all codes of the components within the system while in the second scenario all system components are holding this list. Every time the system is checked all parts are asked for their secret codes and the response is compared to the stored codes. In case of a mismatch an alarm is set. If a part is removed or illegally exchanged from the system it will be noticed during the check. This embodiment considers server-less solution and the possibility of encrypted communication. However, no actual implementation to achieve the claimed security properties is presented. Neither the initialization of the component nor its reset is described. Furthermore, a presentation of the hierarchies is missing as well, i.e. who is allowed to perform which actions.

After presenting patents dealing with system integrity we consider an example of

an already implemented system, namely the digital tachograph. Tachographs are often tampered to forge the drivers' or vehicles' records to avoid regulations about driving hours, speed limits etc. It is referred to [5, 4] to read about frauds with tachographs and about the new smartcard-based tachograph system of the EU. There is a good reason for protecting the integrity of a tachograph since truck driver falling asleep at the wheel cause several times more accidents than drunk drivers. Furthermore, accidents involving trucks are more likely to lead to fatal injuries because of the truck's mass. The old fashioned tachographs are either directly tampered with or its supply is manipulated. Typical offenses are miscalibrating and power interruption. A high-tech attack introduced in [5] uses a complete system pretending to be a "Voltage Regulator" which is spliced into the tachograph cable and controlled by the driver using a remote control fob. By pressing the fob the recorded speed drops. The system can also be switched back to proper use when checked by the police. All countermeasures try to defeat cheatings by controlling the drivers and their records and not by protecting the system itself. Therefore the European Union introduces a new system called *Tachosmart* which replaces the existing paper-based charts with smartcards. This system is very complex and provides different hierarchies. Each driver holds a smartcard containing a record of his driving hours and further information. The truck contains a unit holding a year's history, and authorized mechanics hold a card to calibrate devices. In addition to the lower tamper-resistance of electronic versus mechanical signaling, there are several other interesting problems with tachographs being digital, which are all considered in [5]. Another problem is the considered solution is based on the trustworthiness of fitters. But the most substantial objection to the move to a digital system is that most of the frauds are of procedural kind, e.g., truck drivers are switching their vehicles half way.

In summary, there are some patents which deal with system protection. Most of them describe the objectives to achieve but a detailed description of implementation is missing. The first two considered patents do not consider the use of cryptographic protocols and server-less solutions at all. The third patent describes a system providing system integrity with the use of cryptology and a server-less scenario.

2.3 Theft Protection

There are many attempts for providing theft protection for all kinds of items but they are all, more or less, without success. Especially for cars and some of the commonly stolen parts out of cars, e.g. car radios and airbags, there are many embodiments for theft protection. Most of the solutions are mechanically like anti-theft devices and immobilizer. Some cars are even protected with the use of biometric methods, e.g., the owner of a car can only start the engine after successfully authenticating himself by his fingerprint [3].

Two different kinds of theft protection can be distinguished, namely the direct theft protection and the indirect one. The first category compromises all mechanical protection devices, alarms and all kinds of protection which directly tries to prevent the theft of the object. All systems of the second category are based on the approach that components which cannot be used in another system besides the original one would not be stolen. Systems where all parts are holding a unique identifier which can be compared to IDs on a list of stolen components belong to this category. An example for the latter is described in the patent specification [48] and [46]. In both embodiments all components need to be checked manually and compared to a list of stolen parts, e.g., as it is provided by the police. This check can be easily avoided by drivers by never choosing an authorized garage, and doing all repairs on their own. It is a challenge to keep the list up-to-date. In both patents all components are assigned to a specific car, but a method how to obliterate the associations in the case of a resell of parts is missing. It is unclear how an owner could legally add or change parts. Neither the use of strong authentication nor of encryption is mentioned for the communication between verifier and components. Consequently, both embodiments are not satisfactory for providing theft protection since they are not user friendly and the check has to be performed manually and with great expenses. Another patent [34] describes components holding a unique identifier as recallable data. In this embodiment the data is automatically verified in the vehicle itself and also compared to a list hold by a server. For each component significant data, for instance the ID of the current vehicle, data about possible repairs, date of first and all following assemblies and disassemblies etc., are stored in the list.

Now we consider three embodiments, [47], [32], [33], which basically provide system integrity but also theft protection as additional feature. The embodiments have already been discussed in the previous section in their capacity of providing system

protection. All of them are based on the idea that all parts of a system hold a system secret. This secret is used to check system integrity, e.g., with the help of a list containing all system components. This check implies theft protection. Stolen components would lead to an alarm in their original car because a part is missing. In addition it would lead to an alarm in the car the stolen component is built into, because it does not hold the assigned key of the new system.

2.4 Conclusion

Most of the above presented patents deal with solutions for vehicles. Although we are looking for a general solution for systems with several components these embodiments can still be used as concepts. Other than most existing systems we want to use strong authentication and encryption. Our solution should be serverless, at least in some implementations. Some of the ideas of the presented solutions or a combination of them can serve as basis for developing a solution providing all three security features at once. In summary we are looking for a solution ensuring component identification in electronic systems with networked components by the use of cryptographic protocols. The verified identity is then used to provide piracy, system or theft protection or a combination of these tasks.

3 Identification Protocols and Security Aspects

This chapter considers identification protocols and their security issues in general. The basic terms are defined, and standard protocol schemes and some basic techniques to increase the security or provide other security features are presented. The first section defines the basic terms and introduces the participants of identification protocols. Some general information about identification protocols is given as well. The objectives of identification protocols are then considered in Section 3.2. In the following section the possible attacks on identification protocols are introduced and in the next section the basic identification protocols are developed step by step regarding the prevention of those attacks. After the presentations of the protocols in Chapter 6, 7 and 8 the list of attacks and other possible attacks are checked again in Section 10.1 for showing the security of the solutions. To provide a high level of security the use of strong authentication is recommended. Strong authentication can be achieved by challenge-response protocols, which are introduced in Section 3.4. The basic variants, possible attacks and countermeasures are presented. The knowledge of the facts introduced in this chapter is essential for designing sophisticated identification protocols.

3.1 Some General Notes about Identification Protocols

To prevent confusion about used terms, e.g. identification and authentification, they are defined here once and used in the same manner further on. Most terms and definitions are adopted from [30]. Before considering identification protocols, the general definition of cryptographic protocols is given. A cryptographic protocol is defined as follows:

> **Definition [30]** A *cryptographic protocol* is a distributed algorithm defined by a sequence of steps precisely specifying the actions required of two or more entities to achieve a specific security objective.

The term *identification* is also denoted as *entity authentication* or just *authentication* which is defined by [30] as follows:

> **Definition [30]** *Entity authentication* is the process whereby one party is assured (through acquisition of corroborative evidence) of the identity of a second party involved in a protocol, and that second has actually participated (i.e., is active at, or immediately prior to, the time the evidence is acquired).

Participants of an identification protocol are the *verifier* (can be a single entity or several) and the *claimant* (can also be a single entity or more). The outcome of an identity protocol is either *acceptance* or *rejection*. Whereby *acceptance* means that the protocol proved the claimant's identity as authentic to the verifier(s) and *rejection* that this proof failed. An identification protocol is a "real-time" process in the sense that it provides an assurance that the party authenticated is operational at the time of protocol execution. Identification protocols provide assurances only at the particular instant in time of successful protocol completion. When talking about authenticity and authentication protocol in the following it has the same meaning as identity and identification protocols, respectively, authenticity and authentication protocol.

3.2 Objectives of Identification Protocols

The objectives of identification protocols are independent from the actual implementation and build the framework for developing our protocols. The achievement of these objectives and the prevention of the attacks should serve as the minimum requirement for all protocols to be developed. According to [30] the general objectives are:

1. A is able to successfully authenticate itself to B, i.e., B will complete the protocol having accepted A's identity.

2. Transferability

   B cannot reuse an identification exchange with A so as to successfully impersonate A to a third party C.

3. Impersonation

   The probability is negligible (not of practical significance) that any party C distinct from A, carrying out the protocol and playing the role of A, can cause B to complete and accept A's identity.

The previous points remain true even if:

1. A (polynomially) large number of previous authentications between A and B have been observed.

2. The adversary C has participated in previous protocol executions with either or both A and B.

3. Multiple instances of the protocol, possible initiated by C, may be run simultaneously.

3.3 Attacks on Identification Protocols

In this section types of attacks on identification protocols, taken from [30] and extended by the known-plaintext attack, are introduced. Some hints are given how to avoid them.

1. *Impersonation*, a deception whereby one entity purports to be another.

   Can be prevented by using authentic and secret key material only.

2. *Replay attack*, an impersonation or other deception involving use of information from a single previous protocol execution, on the same or a different verifier. For stored files, the analogue of a replay attack is a restore attack, whereby a file is replaced by an earlier version.

   This attack is prevented by the use of challenge-response protocols which are presented below, since identical messages would not be accepted a second time.

3. *Interleaving attack*, an impersonation or other deception involving selective combination of information from one or more previous or simultaneously on-going protocol executions (parallel sessions), including possible origination of one or more protocol executions by an adversary itself.

   Since this attack makes use of neat combinations of protocol messages of parallel or previous sessions, it has to be kept in mind when protocol steps should be implemented parallel. The designer of the protocol should also use a neat combination and interleaving of the protocol messages to prevent this attack.

4. *Reflection attack*, an interleaving attack involving sending information from an on-going protocol execution back to the originator of such information.

   Can be easily prevented by the use of identifiers in some protocol messages, e.g., by sending the verifier's ID in the response, as shown in the next section.

5. *Forced delay*, forced delay occurs when an adversary intercepts a message (typically containing a sequence number), and relays it at some later point in time. Note that a delayed message is not a replay.

   This might be prevented by the underlying protocol used for the actual transmission of the messages, e.g. by the use of timeouts, and is not an issue for the protocol steps of the identification protocol.

6. *Chosen-text attack*, an attack on a challenge-response protocol wherein an adversary strategically chooses challenges in an attempt to extract information about the claimant's long-term key. Chosen-text attacks are sometimes referred to as using the claimant as an oracle, i.e., to obtain information not computable from knowledge of a claimant's public key alone. The attack may involve chosen-plaintext if the claimant is required to sign, encrypt, or MAC the challenge, or chosen-ciphertext if the requirement is to decrypt a challenge.

   An attacker should not be able to select specific challenges to extract information about the secret encryption key used in the received response. This might be prevented by restricting the requests to authorized claimants only, e.g., the request has to be encrypted by a secret key as well.

7. *Known-plaintext attack*

   The attacker should not be able obtain corresponding plaintext ciphertext pairs by simple eavesdropping the communication. To prevent this attack it

is recommended to encrypt as many messages as possible.

3.4 Strong Authentication Protocols

As we are looking for strong authentication we will use challenge-response identification. According to [30] the idea of cryptographic challenge-response protocols is that one entity (the claimant) "proves" its identity to another entity (the verifier) by demonstrating knowledge of a secret known to be associated with that entity, without revealing the secret itself during the protocol[1]. This is done by providing a response to a time-variant challenge, where the response depends on both the entity's secret and the challenge. The *challenge* is typically a number chosen by one entity (randomly and secretly) at the outset of the protocol. If the communication is monitored by an adversary, the response from one execution should not provide useful information for a subsequent identification, as subsequent challenges will differ. Strong authentication can be achieved by three different authentication schemes, namely symmetric techniques, asymmetric techniques and zero-knowledge concepts. Zero-knowledge proofs are not considered any further since they are not suited for the regarded problems due to their high computation cost.

3.4.1 Challenge-Response by Symmetric-Key Techniques

Challenge-response protocols using symmetric encryption are based on the assumption that all participants share a secret key K a priori. Furthermore it has to be assumed that the keys are secret and thus are only known by authorized participants. All participants hold an authentic copy of K. Using challenge-response and the made assumptions about the used keys prevent the attacks (1) and (2) described in the previous section.
The basic challenge-response for unilateral authentication, Alice versus Bob, looks as follows:

1.	A: $\longleftarrow$ B: r_B	A: $\longrightarrow$ B: $E_K(t_a)$
2.	A: $\longrightarrow$ B: $E_K(r_B)$	
	using random numbers r	using timestamps t

[1] in some mechanism, the secret is known to the verifier and is used to verify the response; in others, the secret need not actually be known by the verifier

The disadvantages of this protocol are its vulnerabilities to known-plaintext (7) and reflection attacks (4). Malory obtains r_B and $E_K(r_B)$ by simple tapping the communication bus and holds a corresponding plaintext-ciphertext pair. If sessions could be executed parallel and Malory is challenged with r_B, he could start a second session by simple returning this message and would receive the valid response from Bob. This response is used by Malory again as valid response in the first session. Latter attack can be prevented by the use of the verifier's ID in the response as presented in the extended protocol:

1.	A: $\longleftarrow$ B: r_B	A: $\longrightarrow$ B: $E_K(t_a, ID_B)$
2.	A: $\longrightarrow$ B: $E_K(r_B, ID_B)$	
	using random numbers r	using timestamps t

The protocol using random numbers is still vulnerable to known-plaintext attacks (7). This can be prevented by encrypting both protocol messages which leads to our "final" version of the challenge-response protocol for unilateral authentication using random numbers:

1. A: $\longleftarrow$ B: $E_K(r_B)$
2. A: $\longrightarrow$ B: $E_K(f(r_B), ID_B)$

using random numbers r

According to [23] the purpose of function $f()$ is to ensure that message 1. and 2. do not have identical input, whereby f() is a function such as increment. Due to the identifier used in message 2, only a part of the encrypted message would be equal. This version is used for every unilateral authentication in the later presented protocols.

The challenge-response protocol for mutual authentication, Alice versus Bob *and* Bob versus Alice, resisting known-plaintext and reflection attacks looks as follows:

1. A: $\longleftarrow$ B: $E_K(r_B)$
2. A: $\longrightarrow$ B: $E_K(r_A, f(r_B), ID_B)$
3. A: $\longleftarrow$ B: $E_K(f(r_A), f(r_B)$

using random numbers r

As stated in [30] does the execution of two unilateral authentication protocols, one in each direction, not automatically lead to a mutually authentication in ad-hoc networks. *While mutual authentication may be obtained by running any of the above unilateral authentication mechanisms twice (once in each direction), such an ad-hoc combination suffers the drawback that the two unilateral authentications, not being linked, cannot logically be associated with a single protocol run.*

3.4.2 Challenge-Response by Asymmetric-Key Techniques

According to [30] public-key techniques may be used for challenge-response based identification, with a claimant demonstrating knowledge of its private key in one of two ways:

1. The claimant C decrypts a challenge encrypted under its public key P_C.
2. The claimant C digitally signs a challenge by its secret key S_C.

Ideally, the public-key pair used in such mechanisms should not be used for other purposes, since combined usage may compromise security. Usually session keys are established for this reason.
It is assumed that all used keys are authentic. Besides that all private keys, the keys used for signing, need to be secret. This assumption precludes all kinds of impersonation attacks, and the use of timestamps or random numbers precludes the replay of messages. Since only unilateral authentication is needed in the later presented protocols mutual are not considered at this point. The unilateral challenge-response authentication protocol based on public key encryption can look as follows:

1. A: $\longleftarrow$ B: $P_A(r, ID_B)$
2. A: $\longrightarrow$ B: r

using random numbers r

The reflection attack is not possible here, since a message encrypted by P_A can only be decrypted by A and thus not returned to the originator to receive the valid response from him.
The unilateral challenge-response authentication protocol based on digital signatures can look as follows:

1.	A: $\longleftarrow$ B: r_B	A: $\longrightarrow$ B: $cert_A, t_A, B, S_A(t_A, B)$
2.	A: $\longrightarrow$ B: $cert_A, r_A, B, S_A(r_A, r_B, B)$	
	using random numbers r	using timestamps t

The reflection attack is again precluded due to the use of the secret key. The random number r_A in the second message is used to prevent chosen-text attacks.

4 Solution Categorization

We are looking for solutions which provide either piracy protection, system protection or theft protection. These objectives can be achieved separately or concertedly. It turns out that system and theft protection can be solved at once. Thus we have three different categories of solutions.
To support the user in finding the right solution for her problem, or more precisely system, the solutions are categorized. The different categories will be described in the following, especially regarding their objectives, the main idea the solution is based on and requirements on the system environment. For choosing the right solution the user first needs to determine the objectives she wants to achieve by implementing one of the protocols. Besides that she needs to evaluate some characteristics of her system such as the restrictions of her system due to the environment and the network architecture. The below presented categorization should help to find the best solution for a particular system. Several suited applications will be given for each solution to illustrate possible realizations. The solution proposed for providing piracy protection, and thus also the later presented protocol in Section 7, is referred to as *proof of origin* in the following and described in the first section. The solution for providing system and theft protection as well as the proposed protocol is referred to as *system check* in the following. This solution is introduced in Section 4.2. The combination of both previous solutions is presented in the last section. This solution provides piracy, system, and theft protection at once. It is referred to as *proof of origin and system check* in the following.

4.1 Proof of Origin

Definition The *proof of origin* is a protocol which can be executed in a composite system by a verifier (a server, HSM, or system component) to proof the origin of a claimant (a component). After the execution the verifier knows if the verified component is an original part or a counter-

feit.

The objective of the *proof of origin* is to distinguish between original parts and counterfeits. Since only authorized components can be authenticated successfully the *proof of origin* provides piracy protection. The basic idea is that piracy protection can be achieved by labeling the original parts in a specific manner and checking for these characteristics during the assembly of components into a system. Therefore it needs to be ensured that those characteristics cannot be copied and used in counterfeits to pass the checks. The characteristics of systems which are suited for the implementation of this protocol are (1) the origin of a component needs to be verified just once and (2) the single components are part of the system for a short period only. After passing the check the component is authenticated until its disassembly, usually one session. Thus the solution is suited for systems with transient associations. Suited applications are usually systems with a client-server architecture. With one of the participants changing all the time. From the view of the server the components are changing and for the components the server is changing. The components (clients) have to authenticate themselves for using a special service provided by the server or being used for special services by the server. In the first scenario the component uses a provided service during one session. The component is removed out of the system right after using the services and all associations are obliterated. One typical application are credit cards. The credit card is the client and has to authenticate itself versus a automated teller machine (short ATM) which acts as the server. After the successful authentication the credit card or in fact the card holder can use confidential services provided by the terminal like transferals or drawing statements of account. After the use of the services the credit card is taken out and all associations cease to exist. The credit card can be used with many ATMs and one ATM is used by many different credit cards.

In the other scenario the server might use some services or data provided by the authorized component. The authentication could be then used to observe the copyrights, for instance if the server is a game console and the client the disc of a game. The disc is the client and has to authenticate itself to the game console. This ensures that only original game run on the game console. The used data is provided by the disc.
Examples of some applications are the following, whereby many more are imaginable:

- Credit cards

  The credit cards are the clients which have to authenticate themselves versus the ATMs, which represent the server.

- Phone-cards

  The phone cards are the clients which have to authenticate themselves to make use of special services offered by a provider which acts as the server.

- Film reel versus projector

  In this application the client has to authenticate itself to provide data to the verifier. If the film reel is proved to be an original one the projector plays back the movie. This could ensure that the owners of theaters can only show original movies.

- DVDs and CDs versus player

  Players of all kinds only play back data of originals.

- Games versus Play-Station

  Only original games can be used in the console.

4.2 System Check

> **Definition** The *system check* is a protocol which can be executed in a composite system by one or more system components to check the integrity of the entire system. After the execution the verifier know if the verified system is modified by unauthorized entities or not modified since the last *system check*.

The *system check* provides monitoring of the system integrity. Thus all systems which should be monitored for unauthorized modifications are suited for this solution. All changes of the system, i.e. adding, removing or exchange of components, are monitored by the *system check*. An alarm is set when any of these changes are performed by an unauthorized person or entity. The main idea is that all parts of a system share a secret which is assigned to a particular system. Only components which are in possession of this secret are valid system members. All other are not authorized and will be detected during the *system check*. This provides system

protection, since all components which are not from the considered system will be detected, and implies theft protection since these detected components cannot be used in the system. The idea of providing theft protection is based on the assumptions that nobody would steal components if those components could not be used in any other than their original system. Thus it is more an implicit theft protection since stealing components might still be easy, but it is just not worth it because the stolen components are useless. Another feature provided by this solution is the possibility of confidential communication within the system by using symmetric encryption with the shared secret as key. This might be of special interest for component's interconnected by wireless links, since they are prone to eavesdropping.
In some systems piracy protection is not necessary or ensured implicitly. In the first scenario the label or origin of a component is meaningless but the system integrity has to be kept, e.g., if you want to monitor if somebody has tampered with your system. In the latter scenario the piracy protection is provided by an independent process or by the assumption that the assembly of the system took place in a secure environment. This could be warranted by ensuring that only authorized personnel or entities are allowed to carry out modifications to the system. Thus either the people who are building in the components have to take on the full responsibility to distinguish between original parts and counterfeits or another autonomous procedure, e.g. by the use of one of the security feature presented in Section 2.1, might ensure the originality of the component before new components are built into the system.
The *system check* would be suited, e.g., for machine tools, since new components are only assembled by specialized staff (secure environment) and the execution of the protocol could be used to ensure the occupational safety by stopping the machine in the case counterfeits are used. The solution is also suited to be used for airbags. As in the previous example airbags are only built in by specialized personnel. If the system is altered in an unauthorized manner afterwards, the safety of all occupants would be imperiled. The system check would detect these modification and could take measures as to stop the engine or disable the ignition.

Examples for suited applications are as follows, whereby many more are imaginable:

- Specific machines like machine tools

See description of the example in the text above.

- Computers and their components

  All components of a computer share a secret. Only components which hold the secret can be used. All other parts will be disabled. The system integrity is ensured and stolen parts, e.g. hard drives, cannot be used in other computers.

- Taximeter, tachographs and speed limiter

  These devices are considered in detail in [5]. Those components and the assigned vehicle should form one unit, or in other words one system. All unauthorized modifications of the system, e.g. performed by malicious mechanics, would be detected during the *system check*, e.g, executed by the police during an inspection.

- Airbag

  See description of the example in the text above.

4.3 Proof of Origin and System Check

Definition The *proof of origin and system check* is a combination of both previous protocols.

This solution provides piracy, system and theft protection by combining both previous solutions. Piracy protection is warranted by the *proof of origin* protocol and the system and theft protection by the use of the protocol of the *system check* protocol. Due to the implementation of the *system check*, the confidential communication among all system components is provided as well. All systems which should detect bogus parts *and* unauthorized modifications in the running system, or have to prove the origin of the components frequently in the running system, are suited for the implementation of the protocol. This solution could be suitable for almost all systems which consists of two or more parts, whereby the components are capable to communicate with each other. A possible scenario could be a car. Every time the ignition key is turned the system integrity is checked, by the execution of the *system check*, and only in case of an unaltered system the engine starts. Due to the execution of the *proof of origin*, only original parts can be built into the car. A display could show the status, e.g. valid/invalid, of all components which are currently part

of the car. This could be useful in the case that a car has to be fixed in a car garage. The owner could prove then if the mechanics have really used originally parts only. With an option in the display to manually start the *system check* the owner of the car could also prove to third parties, e.g. the police or a potential buyer, that all components in the car are original parts.

Examples for suited applications are:

- Cars, trucks and other vehicles

  Example is described in the text above.

- Trains, street cars, subways

  Similar to previous application.

- Ships

- Airplanes

- Helicopters

- Printers

  The ink (or toner) cartridge has to authenticate itself versus the printer. Since the cartridge is used for a longer period and usually in one printer only, this application is rather suited to this combined solution than for the single *proof of origin*. At the first time of its assembly the cartridge is checked for its label, after then the *system check* is executed instead of checking the authentication.

- Vacuum cleaner

  The vacuum cleaner bags have to authenticate themselves versus the vacuum cleaner. This example is an application with a highly restricted technical environment, since the vacuum cleaner usually is not connected to the Internet.

- Weapons

  The bullets have to authenticate themselves versus the weapon. This could be useful to prevent the use of stolen police weapons and to prevent the use of stolen weapons or shots in a war by the enemy. Lately this could also be suited for the weapons of so called *Sky Marshals* or armed pilots to improve the safety on flights.

- Cell phones

  The antenna and the battery have to authenticate themselves to the cellphone body. After a successfully first authentication during the *proof of origin* the parts form a system which is then checked for its integrity by the execution of the *system check*.

- Single components

  Sometimes a special monitoring of highly sensitive system components might be necessary. This can be one single component out of an entire system or the system only consists of two components, the sensitive one as claimant, the other one acts as verifier.

  Examples for single components which are typical targets for fraudulent manipulations are:

  - Tachographs
  - Speed limiter
  - Electronic toll systems
  - Airbag
  - ABS, ESP
  - Active under-carriage
  - Tire pressure monitoring

5 Getting Prepared

In this chapter we will first introduce the general categories, objectives and design criteria of the later presented protocols. Furthermore all used terms are defined and all required assumptions and parameters are given. Section 5.1 provides a rough survey about the categories of systems. The following Section 5.2 deals with the general objectives of cryptographic protocols which leads to the design criteria of the protocols in Section 5.3. In Section 5.4 all important terms are defined and explained. Section 5.5 deals with the assumptions which have to be made for securely executing all protocols. Next general parameters which have to/can be set for the particular scenarios are presented.

5.1 General Categorization of the Systems

Cryptographic protocols depend on the encryption scheme and the network architecture of the system. Both affects the protocol flow and the assumptions and preparations to be made. All solutions, presented in the following chapters, can be split in two categories according to the encryption scheme and the network's architecture, respectively, which makes a total of four possible combinations.
The encryption schemes are subdivided into the following:

Symmetric encryption schemes are implementations using symmetric encryption and decryption by the use of secret keys which are shared by all participants a priori. The system relies on the secrecy and authenticity of the used keys. Symmetric encryption is considered secure when used with keys of adequate bit length. The major advantage of symmetric encryption is the fast execution. The main problem is the secure distribution of the symmetric key.

Asymmetric encryption schemes are implementations using asymmetric encryption and decryption or signing and verifying by the use of public and private

key pairs. Each participant holds its own key pair and uses the receiver's public key for encryption or its own private key for signing a message. The receiver decrypts or verifies the message with the corresponding key. The advantage of asymmetric encryption is that the potential communication partners do not have to share a secret before communicating, and thus does not even need to know each other a priori. Since, according to [40], correctly verifying the certificates is not sufficient to guarantee the validity, the verifier also needs to check whether that particular certificate has been revoked. It follows that the verifier has to contact a trusted third party which is in possession of a current certificate revocation list (CRL). For this the authentication procedure requires on-line access to centralized services if want to completely exclude that revoked certificates are being accepted. Thus the use of asymmetric encryption in a peer-to-peer network, which is explained below, would be impossible. It might be implemented with certificates containing a specified validity interval under the assumptions that the certificate has not been revoked in this interval. Another approach is an infrequent checking of a CRL.

The architecture of implementations is classified in:

master-slave networks consist of a central knot referred to as master and at least one entity called slave. The master could be an external or internal server. The server may hold extra data, e.g. a database of critical key material, and is usually more powerful, e.g. computational power, greater data storage etc., than all other members in the network. Due to the importance of the master it cannot be compromised. Systems with server normally rely on its trustiness. Hence the vulnerability of the server represents a single-point of failure. An example for a master-slave network is an automobile where the central board computer acts as master and all connected devices are slaves. An example for an external master is an on-line connection to a computer which is not part of the system, e.g., via the Internet. The advantage of an external master is that it is easier to protect against attacks.

peer-to-peer networks or short P2P consist of several components. These components are all connected with each other, either in a direct manner or via some other nodes. There is no server. All components provide data or services to each other. The associations between the nodes are transient and components might be added, removed, or exchanged all the time.

5.2 General Security Objectives

The four common security objectives of networked systems are as generally known:

1. Availability
2. Authentification/Identification
3. Confidentiality
4. Integrity

Availability is the one of greatest relevance for the user [41, 40], whereby *availability* means ensuring that the service offered by the node will be available to its users when expected. Possible attacks on the availability of a system are denial of service attacks, radio jamming in wireless systems and battery exhaustion in battery powered systems. Ensuring availability is a main concern of systems since, as stated in [41], else counts little if the device cannot do what it should. We do not consider system availability in our protocol design. It is left to the underlying data protocol to deal with those kinds of attacks.
The *Authentification/Identification* property is core objective since all others become useless if an entity does not know whom it is actually talking to. Talking confidentially to Malory while thinking it is Bob does not meet the goal. Providing authentication means that one party is assured about the identity of another party. For achieving our goals, i.e. the providing of piracy, system and theft protection, ensuring the authenticity of components is the main issue. Verifying the authenticity of a component is the primary objective of all later presented protocols. The identity of a component can be used to prove if it is an original part, or a part of a particular system, or if this component is a stolen part or a legally bought one. In systems with frequently changing components, e.g. a credit card - ATM system, or changes due to consumption goods as the system printer-cartridge, a new authentication problem occurs which is called *secure transient association* [41, 40]. All associations between the components are transient since the components of a system are changing frequently. For instance, if a component is sold all old associations should cease to exist and new associations should be create to the components of the new system.
Confidentiality of messages or data ensures that only authorized entities are able to read or access the message or data, respectively. As said in [41, 40] the real issue

is providing authenticity. Protecting *confidentiality* is by then simply a matter of encrypting the session using available key material. Thus the main problems have to be solved before the encryption can start, namely ensuring the authenticity off all participants and the authentic and secure distribution of the keys.
Data integrity ensures that the messages cannot be altered by malicious participants without being noticed by the system or at least by the recipient of the altered message. It can be achieved by encrypting messages under the assumption that all used keys are secret and authentic. The integrity of all entities has to be considered as well which means no unauthorized entities can tamper with them. This can be achieved by physical protection of the entities, e.g. by restricting the access to the entities to authorized persons/entities only, and physical tamper resistance of the entities themselves. Later could be realized by the use of physical tamper-evidence mechanisms, such as seals, or the use of electronic mechanisms, such as tamper sensing switches that zeroize memory. We consider this in more detail in Section 10.3.

5.3 Design Criteria of the Protocols

The design criteria form the basis of all protocols which are presented in the next chapters. The criteria are directly derived from the general objectives of identification protocols in Section 3.2 and the additional security properties we want to achieve by implementing our system. The latter provide following piracy, system and theft protection and are described in the previous section. Some criteria are derived implicitly from the prevention of the attacks considered in Section 3.3. Some criteria are according to the properties of identification protocols in [30]. The primary goal of the design criteria is to achieve all claimed properties.

1. Reciprocity of identification.

   Either one or both parties may corroborate their identities to the other, providing, respectively, unilateral or mutual identification.

2. Computational efficiency.

   Avoid expensive computations when possible. To avoid expensive computations the use of symmetric encryption schemes is preferable, which is for instance realized by the use of symmetric encryption in the running system

in all different scenarios. Additionally all expensive computations should be performed by an external verifier (if any), which is more powerful than the system components. The number of operations should be kept as small as possible.

3. Communication efficiency.

   Use as few protocol messages as possible. This induces faster execution of the protocol and can be achieved by interleaving of messages and avoiding of redundant protocol steps.

4. Encryption of protocol messages.

   To prevent eavesdropping all protocol messages should be secured by encrypting them.

5. Reveal no secret data.

   This goes hand in hand with the previous criterion. Additionally all secrets have to be stored securely.

6. The server knows as little information about the system as possible.

   In an ideal implementation the server does not know any secret system data. It should be considered carefully in the case that the server is providing services to more than one system if the server needs to know which component belongs to which system. The less secret information the server holds the less has to be protected and the more independent is the system.

7. Avoiding the use of messages of the same content.

   Thereby it should be taken into account to avoid the encryption of the same content. The same protocol messages should never contain exactly the same content and be encrypted by the same key to avoid attacks by crypto analysis. At best the content would be not predictable by just knowing the protocol flow. For instance the *hello* message at the beginning of each procedure of *assembly* should always contain ever-changing data instead of just the *hello* message. Otherwise Malory could try a *known plaintext attack* to compromise the secret key.

8. Supporting non-interruption of the protocol flow.

This can be achieved by cleverly interlacing the protocol messages of one or consecutively executed procedures. A possible scenario could be that a component sets an alarm after waiting for a particular message for a certain time period.

9. All associations are transient.

   It has to be ensured that all components can be re-used in other systems. Therefore it is necessary to distinct between components that were removed unauthorized, i.e. stolen, and components that were removed by an authorized person, e.g., for the purpose of resell. It has to be ensured that associations can only be obliterated by authorized persons or entities, e.g., by running an extra procedure for the *disassembly*.

10. The execution of the system check should, to the greatest possible extent, be independent of external entities.

    The *system check* is executed without any external help such that it is suited to off-line systems. This gives the required flexibility for practical application.

5.4 Definitions and Notations

All terms, protocol items such as lists, keys etc., participants and items of the presented protocol are introduced and described here in general. The participants and protocol items are mentioned again briefly in Chapter 6, 7 and, 8 in case they take part in the protocol or affect it in any way.

A System is a network of n nodes. In some realizations a high security module (HSM) is used as replacement for a server. But in contrast to a server, the HSM is considered to be a part of the system. The nodes, i.e. the components, all share a secret for secure communication. The HSM does not share a secret with the other nodes but uses individual keys for securely communicating with the network nodes.

A Solution includes all protocols necessary to provide our goals. For example, the *proof of origin* and the *system check* are solutions.

A Protocol is a self-contained process which consists of single procedures. A protocol has a specific start and end point, and all possible intermediate and end

results of the protocol run are anticipated and lead to a defined state. Examples of protocols are the protocol of *assembly*, the *system check*, and the *disassembly*.

A Procedure is a self-contained process within a protocol. All possible results which might occur during runtime are anticipated and lead to a defined state. Since some procedures are executed consecutively the results, i.e. the outputs, of one procedure might be passed to the following one. In some protocols a procedure starts only after being notified by the previous procedures about their successful execution. In other cases the result might be used as input of the next procedure, e.g. an once selected verifier could still act as verifier in the next procedure. The interaction between procedures allows to save cost, e.g., by avoiding to choose a verifier each time, and helps to provide non-interruption, e.g. by setting an alarm after a node was waiting for a certain time to get notified.

An Implementation is the use of a protocol or an entire solution in a real environment. The implementation is independent of all parameters. Whereby parameter are values predetermined by the environment of the particular system or have to be set by the user before installing the protocol. The parameters help to customize a solution for a particular system. The implementation is the protocol of one of the solution before customizing it for individual needs. The symmetric solution of the *system check* in an automobile is an example for an implementation.

A Scenario or Realization is an implementation with a particular set of parameters.

Lists *Certificate Revocation List, Secret Key Revocation List, Used List, Ready for Assembly List, ID-List*

All lists have to be secured in a way that they cannot be altered by unauthorized entities. The content of some lists could be non-confidential and might be readable by anybody, but the lists are write protected. Thus the provider of lists has to protect the data in a manner that records cannot be added, altered or erased by unauthorized entities. In some cases the lists' data is confidential and thus the lists have to be read and write protected.

The *certificate revocation list*, short *CRL*, consists of records of all revoked certificates. The list has to be hold by the verifier during the authentication of a newly assembled component. The certificates of all known fraud components are put on the list. For example, all duplicates which are detected during the *proof of origin* are added to the list. If a component's certificate is verified successfully and does not hold a record on the *CRL* it is assumed that the certificate's owner is an original part.

The *secret key revocation list*, short *SKRL*, is used in symmetric implementations and is used in a similar way as the CRL of implementation using asymmetric encryption schemes. All revoked secret keys of fraud components are put on the list. The same counts as for the above described CRL.

The *used list* contains records of all components which are currently part of a system. In some realizations it might be a list of all components ever used in a system, i.e. even the one which are not currently part of a system anymore. The list is usually stored on a server and might additionally stored on a HSM or a particular system component in systems which are temporarily on-line only. In off-line realizations it is hold by an entity, a HSM or system component which takes over the duties of the server. The list is used for the detection of duplicates in asymmetric solutions and for the secure communication between the list holder and system components in symmetric solutions. There are scenarios where the *used list* is global and thus contains the components' data of all systems or others where the list is split into local lists which contain the records of a particular system only. A record includes the component's ID, it's secret or public key, and optional data, e.g., the *status* of the component within system, a *description*, the *priority*, a *policy* and much more. The system status describes if the component is a valid system member or if it is waiting for authorization. The description describes the type of the component, e.g., the brakes in an automobile, the airbags, the radio. The policy might be used by all system components during the *system check* to decide when a rule is broken and when to set an alarm.

The *ready for assembly list* contains records of all components which are authorized for the assembly into a system. The list is held by the server or representatives as a HSM or a system component. This list is essential for symmetric solutions since the components cannot authenticate themselves ver-

sus the verifier without being known by it a priori. The list could also be used as a replacement for the *used list* and the *CRL* in an asymmetric solution. The records on the list consist of at least the component's ID and its secret or public key. It might also contain additional data. A static *ready for assembly list* is suited for off-line systems. Whereby static means that *all* authorized components are known a priori and are recorded on the list and the list cannot be altered after the implementation of the system.

The *ID-list* is system specific and hold by all system members. In some scenarios it is also hold by external entities. The list is mandatory for all protocols. It contains a list of data records of all system components. These records contain at least the component's ID. It can also contain additional data like the component's status within the system, the component's priority, a description etc.. The *ID-list* needs to be updated every time the system is altered, e.g., due to a newly assembled or disassembled component.

Keys *System Key, Secret Key, Public Key, Private Key, Certificate, Session Key*

The *system key* is essential for the *system check* and the combined solution *proof of origin and system check*. This key is system specific and used in symmetric encryption schemes. Each system uses a system key K which is hold by all system components. The key is secret and only components which are authenticated successfully during their assembly obtain the system key. The key is securely stored by the components, i.e., the key memory is read protected. Thus all components which hold the key are assumed to be trustworthy. The key is used for securing the communication among the system components. Additionally it is used as proof of identity within the system, i.e., as proof that a component belongs to a specific system.

A *secret key* is hold by each component in a symmetric solution a priori, i.e., the keys are set during their manufactory. This key is used for authentication purposes and also for encrypting the communication between the component and its verifier. The key has to be secret, i.e., it is only known by the particular component and the verifier. All potential communication partner have to hold a secret key a priori, which is realized on the verifier's side by the use of lists, namely the *ready for assembly list* and the *used list*. The key memory of the component needs to be read protected and the verifier's key lists must be stored securely.

The *public and private key pairs* are used in asymmetric implementations. A particular key pair is assigned to each component and is used for its authentication versus the verifier. It might also be used for securing the communication with external entities. The public key is part of a certificate and the private key is only known by the assigned component. Thus the component's key memory needs to be read protected. Since the certificate is send to the verifier nothing has to be set up a priori. The verifier only needs to hold the *CRL* and an authentic copy of the public key of the certificate's issuer.

Certificates are used in all asymmetric implementations. They are issued by the manufacturer of the entire system or by superior organizations. In an implementation of a protocol in an automobile, *Audi* could be the issuer of the certificates. *Audi* issues certificates to all authorized suppliers, e.g., *Bosch* and *Siemens*. The certificates contain all data necessary for the component to authenticate itself versus a verifier. A certificate contains at least the component's ID, the components public key P_C and further data. The verifier needs to hold an authentic copy of the issuer's public key P_m.

Session keys might be used in systems where several messages need to be exchanged after the successful authentication of a claimant. To avoid using the same keys for authentication and encryption, which might reveal some information, the use of session keys is suggested. The session key K_{ses} is generated and distributed securely by one or more trusted participants before confidential data needs to be exchanged. Thus all messages encrypted by this key are assumed to be confidential. K_{ses} is a symmetric key and consequently hold by all communication partners. The session key is only used during one session and does not need to be stored any longer.

Miscellaneous *ID, Priorities, Policies, Clones*

Each component holds *unique identifier*, short *ID*, which is used to distinguish between components. The ID is a public name and used to uniquely address all component. When a component sends its ID to a verifier, the verifier is able to search for the ID on his lists. The ID is used for the distinction of the components within a system, but also within the global context.

Priorities can be assigned to all components and used in combination with policies. Priorities enable to set corresponding levels of alarms and sanctions if an authentication of a component holding a certain priority is rejected.

The rejected authentication of low priority components cause an alarm of low urgency such as a flashing light or a system message. A failed authentication of a component holding a high priority could, for instance, follow the shut down of the entire system. Thus the taking of measures and the setting of alarms are not only dependent on protocol failures, e.g., invalid or missing responses, but also on the priority of the component which caused the failure. The priority of a component could be set during its manufactory, or set by an authorized entity before its assembly into a system. The priority has to be stored write-protected, otherwise Malory could set the priorities of components that low that hardly any alarms occur. In realizations using priorities the system's ID-list contains the priority of each component. Each component has a specific priority, e.g. in an automobile, the airbags would have a higher priority than the turn signals. For instance, airbags would hold a high priority and thus the system would stop immediately when a failure occurs. It would take more failing components of a low priority to set an alarm.

A *policy* might be used to fine tune the security level of a system. Policies come together with priorities. If components without holding any priorities should still be accepted, it is suggested to set a default priority. Each system component holds its own policy. The policy defines how many components of a specific priority have to fail until an alarm and/or other measures are taken. Furthermore, it defines what kinds of measures have to be taken. Thus the use of individual policies for each component constitutes a security policy for the entire system. The policies are either set up during the manufactory and thus independent of the system, or they set-up takes place right before the assembly into the system. In latter case a scenario using a default policy for the entire system is possible. The policies could be used, e.g., during the *system check*, to decide when a security rule is broken and when to set an alarm. In an automobile, for instance, the rejected authentication of a single airbag would cause the shut down of the entire car whereby the rejected authentication of several components of lower priority would be necessary to cause strong sanctions like that.

Now we consider all entities which might participate in the protocols. It is assumed that all mentioned components are authorized.
The possible participants of the protocols are:

- *New components*

  These components are authorized for assembly into s new system. Thus these components are either brand new and from an authorized brand, or used and authorized for re-assembly and of an authorized brand.

- *System components*

  These are all current components in the running system (n in total), i.e., they have already passed the *proof of origin* and are part of the system. In some temporary on-line or off-line scenarios a system component is used as a replacement for the server. Thus this component has to hold necessary data to verify the requests of all potential claimants. Since a system component has limited protection for higher level of security a realization with a HSM is recommended.

- *The server*

  The server is permanently or temporarily connected to the system and provides a database which contains all required data for verifying the requests for authentication. Since the server is outside of the system it is easier to protect than a system component. We assume that the server is physically and logically protected against unauthorized access. The communication between the server and the system is secured by the encryption used in the protocol.

- *High Security Module*

  In some implementations a High Security Module (HSM) is used in the system as replacement for the server during the time the system is not connected to the server or in server-less realizations. However, the HSM cannot provide current data from the outside of the system like the newest CRL. The HSM provides a high level of security due to his specific architecture. It is especially designed to protect the confidential data it holds.

5.5 General Assumptions

Below we give our fundamental assumptions for our protocols. We explain the assumptions and give brief hints how they can be ensured. The impact in case of failed assumptions is considered in Chapter 10. The general assumptions are as follows:

1. The server is trustworthy and immune to attacks

   All data provided by the server can only be accessed by authorized entities. The provided data is write-protected and if necessary read-protected. Malicious actions will never be performed by the server.

2. The HSM is trustworthy and immune to attacks

   If a HSM replaces the server it is trustworthy and as unbreakable as a server.

3. Each component which holds a key K is trustworthy

   Since only successfully authenticated components obtain the key these components can be assumed to be trustworthy.

4. The keys K are secret

   The key is only known by system components which have been successfully authenticated by a trustworthy verifier. The key is stored read-protected from outside parties. All protocol messages encrypted by K are confidential.

5. The keys K_i/ S_i are secret

   The key is stored read-protected from outside parties. Thus all protocol messages encrypted by these keys are assumed to be confidential.

6. The key K_{ses} is secret

   All session keys which might be generated during a protocol execution are distributed securely among all participants. Thus all protocol messages encrypted by K_{ses} are confidential.

7. The ID-list is authentic

   The integrity of the ID-list is ensured. The entire list and its single records can only be altered by authorized entities. This is achieved by storing the list write-protected and accepting altering commands only from components which are proved to be in possession of the system key K.

8. Each ID is unique

   The uniqueness has to be ensured by the manufacturer of the component, or an organization which determines the ID. The ID is comparable to a MAC address of network cards. By keeping track of assigned IDs or by choosing large random IDs, e.g., 128 bits, it can be assumed that the IDs are unique.

5.6 Parameters

Before a protocol can be implemented in a real environment parameters have to be set. The parameters might be predetermined by the system's environment, the required security level, or the budget. The best result can be achieved for each particular realization by a neat combination of parameters.

5.6.1 Parameters which Affect the Protocol Flow

1. **On-line/ off-line connection**

   a) *Permanent on-line connection*

   The system is always connected to a server, hence it has always access to the current (fresh) CRL or SKRL and the *used list*.

   b) *Temporary on-line connection*

   In this scenario entities are taking over the part of the server for the time the system is off-line. Each time the system goes on-line the system's data is synchronized with the server's data. Temporary on-line connection could also mean that an update is performed manually by using a CD-ROM to refresh the lists of the system. All cases of data integrity between off-line verifier and on-line server which have to be distinguished in the protocols are given in Table 5.1.

   This table shows all cases of systems with a verifier which is neither an on-line server nor an entity with permanent access to an on-line server. This entity is named *off-line verifier* in the table. We distinguish between four cases of data integrity of components' records. In the table we consider the record of a component C_i. Either the component's record of C_i matches on the server's and verifier's side, case 1a) and 2b) of the table, or the record not matches, case 1b) and 2a) of the table. In the case of data integrity the server and the off-line entity both hold a valid record of the component, or a invalid one or no record of this component. Whereby valid and invalid records describe records of authorized and unauthorized components, respectively.

   The two cases 1b) and 2a) where both, server and verifier, hold different data records have to be taken into special account, since this might

		Off-line verifier	Server
1	a)	invalid/no record C_i	invalid/no record C_i
	b)	invalid/no record C_i	valid record C_i
2	a)	valid record C_i	invalid/no record C_i
	b)	valid record C_i	valid record C_i

Table 5.1: All cases of data integrity of an off-line system

enable attacks on the protocol. Case 2a) of the table enables unauthorized components to pass checks during protocol executions. This could happen, for example, when cloned components are detected and their records put on the CRL, but the CRL is not updated in all systems. That follows that clones holding the "invalid" record could be still used in all systems which CRLs are not updated since the last changes. Case 1b) might cause the rejection of authorized components. Note that the entity acting as verifier does not know if it holds a current version of the components' data or if its data differs from the current server data. According to Table 5.1 the entity can distinguish between case 1) and 2), but not between variant a) and b). For ensuring data integrity and prevent fraudulent use of the system all actions performed while the system is off-line are recorded and checked the very next time the system has access to the server. Therefore the verifier could provide a *waiting list* where all component actions are recorded. Either critical actions are put on this list or are denied by the system. In the first case these actions are checked during the next on-line connection to the server, in the second case all actions can be re-tried during the next on-line connection to the server. For the first variant further security measures have to be taken into account such as an update of the system key (see Section 8.3.2), after detecting that a component was detected falsely. The second variant is less convenient since you always have to wait for the next time the system is connected to the server, but it provides a higher security level and is easier to implement, in reference to avoid attacks which are based on the non integrity of the data.

i. *with High Security Module (HSM) as part of the system*
   When the system is not connected to the server during a protocol execution the HSM takes over the tasks of the server.

ii. *without additional components*

When the system is not connected to the server during a protocol execution, a system component takes over the tasks of the server. This scenario does not have to be regarded separately for the protocol flow. The only difference between a HSM and a system component as server replacement is the security level they provide. As it has already been mentioned a HSM provides a higher protection.

c) *Off-line*

An off-line system specifies a system where the data cannot be updated at all. Hence all lists have to be set up a priori and cannot be changed during life time of the system. Note that revocation lists can still be part of the system but cannot be updated in a component's life time. New lists can only be set for a new generation of a product. In an off-line system one entity has to take over the tasks of a server, this entity can either be a HSM or a regular system component. We discuss both variants below.

i. *with High Security Module (HSM) as part of the system*

The HSM takes over the role of the server. Since there is no on-line connection the HSM cannot update its data. The entire system could be delivered with the HSM holding all information about the current system components and a list about all future components which can be assembled into the particular system.

ii. *with no additional components*

In this solution no server or HSM is involved. The system components have to take over the role of the server and have to perform *all* steps of the protocol on their own. No *current* revocation lists are used. Either the same solution as in the paragraph before could be set up or it could be suitable for some applications to implement a static CRL and other lists which are updated for every new component generation.

2. **The client's data that the verifier holds**

For the authentication of components in the assembly procedure the verifier needs to hold specific data to verify the component's request. Three different variants are presented below.

a) *Verifier holds a "used-list" and a "CRL" of all components of all systems*

Since a new component is neither recorded on the *used list* nor on the CRL, the verifier and the claimant do not share a secret a priori. Thus they cannot communicate securely. Consequently, this variant is only suited for asymmetric solutions. The *used list* is for detecting clones and it holds the IDs and further information as, e.g., the certificates of all components which are currently used. Thus a server which knows the component's ID can perform a table look-up to get the component's public key for securely communicating with the particular component. If the ID/public key pair of a newly assembled component matches a pair on the *used list*, the new component is a clone or a forged component. Cloned or forged components might hold the identical data of authorized components which are currently part of a system. These valid components hold records on the *used list*. The matching records would be detected during the assembly of the fraud components and thus cause an alarm. Stolen components were not properly disassembled by an authorized entity. Thus they still hold a record of their original system on the *used list*. These records would be detected during the assembly of the stolen components into the "new system and cause an alarm. Measures are taken by the verifier, for example, by putting the fraud component's record on the *CRL*.

b) *Verifier holds a "ready for assembly-list" and "used list" for all components and of all systems*

This variant is suited for symmetric and asymmetric solution. In asymmetric solution the *ready for assembly-list* tells the verifier which components are authorized for assembly in contrast to a scenario using a *CRL* telling the verifier which components are not authorized. The *ready for assembly-list* contains all records of all components which are authorized for assembly. The *used list* is needed in the symmetric solutions for secure communication between the verifier and "old" system components.

c) *Verifier holds a "ready for assembly-list" for all components and of all systems*

The *ready for assembly-list* contains all records of components which are authorized for their assembly. This variant is suited for the asymmetric

solution, since no keys are provided for securely communicating between the verifier and "old" system components as it would be necessary in a symmetric solution. After the successful authentication the component's record is deleted from the list and put again on it after the successful execution of the procedure of the *disassembly*. Thus the lists restricts the use of clones.

If the *ready for assembly-list* would contain the records of all components which are authorized for the assembly in one particular system, the solution would be suited for an off-line solution.

3. **The system's data that the verifier holds**

The choice of this parameter is important for the process of choosing a random component out of a system which is necessary in most procedures and protocols. The reason for using random components as verifier in the authentication steps is to avoid a single point of failure. Since Malory does not know which component is acting as verifier during the protocol execution manipulating a particular component would not gain an advantage. Either a verifier holds data of all systems at once or it stores the data separately for each system. This highly affects the implemented protocols. In the first case the verifier is not able to distinguish between single systems and hence does not know which components belong to which system. In the latter case the verifier knows which components are grouped together and form a system. If the verifier holds data for one system only, it obviously knows all single members of the system and thus is able to choose directly one component out of the entire system.

a) *The verifier does not know which components belong to which systems*

Due to the missing knowledge about single members of the system the verifier can communicate with the entire system only. To choose a random system component out of one particular system the verifier sends a message to the entire system. The system components need to support the verifier by performing additional computations for selecting a random component among them. The protocol flow could look as follows:

V: $\longrightarrow$: C_i: x, $\forall i \in 1, \cdots, n$, whereby x is a random number

Protocol action:

The verifier V broadcasts a random number x to all system components C_i. Each C_i computes: $ID_i \oplus x$, whereby $\oplus$ denotes the operator for binary adding without remainder also known as XOR operation. The component C_i with ID_i which obtains the smallest result is the chosen random component.

Now we consider how and by whom the random number x is chosen, therefore three different scenarios are presented below:

Generating the random number x

i. *Generated by the verifier*
The verifier able to generate a random number.

ii. *Derived from trigger event*
In manually started procedures the random number x could be derived by the trigger event, e.g., the time a button was pressed to start the procedure.

iii. *Jointly computed by system components*
In off-line applications all components may jointly compute a random number.
An example for computing a random number x jointly is given below:
$C_i \longrightarrow C_j : r_i, \forall i, j \in 1, \cdots, n$ and $i \neq j$
Each component C_i sends to all system components C_j a random number r_i. After receiving all random numbers each component C_i computes x by binary adding (XOR) all numbers:
$x = r_i \oplus r_j, \forall j \in 1, \cdots, n$
Now all system components hold the same random value x and can derive a random component from it.

b) *The verifier does know which components belong to which systems*

The verifier knows all single components which belong to a specific system, hence the server can directly choose a random component out of the current system. There could be realizations where the server indirectly chooses a random component by broadcasting a random number to the entire system as described in the previous paragraph.

4. **Environment in which the first assembly takes place**

First assembly denotes the first time single components are put together to

form a system, i.e., the initialization of the system. The first assembly takes place in a secure environment or a non-secure one. An environment is *secure* if all entities involved in this process are trustworthy. This includes all components and the server as well as the people who are carrying out this procedure. For example a car is put together at a protected production place of original parts by authorized workers. Contrariwise the term *non secure* is used if one of the above mentioned assumptions cannot be made. For instance, a cell phone is put together by its owner. She puts the battery, the antenna and the actual cell phone compound together. Since it is not ensured that the owner is trustworthy, the system (the entire cell phone with all of its components) cannot be assumed to be initialized in a trusted environment. It is important to be aware of the assumptions which can be made about the environment of the first assembly for choosing the proper realization for the implementation of the protocols.

a) *Secure environment*

If the first assembly of the whole system takes part in a secure environment, the entire system can be initialized at once right after all parts are built in. Since the system or a single component is initialized in a secure environment the system key can be distributed in plaintext. The key might be transmitted by a system component or an external entity like a server. For instance, in an automobile the entire system is assembled in a secure environment. Before getting shipped the automobile could be initialized by turning on the ignition key. In this moment all components would receive a random value as system key K by a server which is connected to the data bus of the car. Alternatively the key could be distributed by a special component, such as the engine, which holds the system key a priori and is used as "root" for further initializations of components. This component is referred to as *master component* and replaces the server in an implementation. The *master components* should be hard to clone and forge. The *master component* acts as root for the entire system. If Malory can build a cheaper counterfeit of the *master component* all components would receive the malicious system key of the counterfeit and thus Malory could also use more forged parts. Consequently, it has to be assumed that the *master components* of a system

are main components. For example, in an automobile this could be the engine. A key would be set in the engine during its manufactory. When all parts of the car are put together all components receive the system key from the engine after the ignition key is turned for the first time. A *master component* can be seen as the basic system to which further components are added. All protocol can run on a system consisting of the *master component* only.

b) *Non-secure environment*

Since the entire system or additional single parts are initialized in a non-secure environment, the system key K has to be transmitted securely by encrypting it. Otherwise Malory could eavesdrop the communication and easily obtain K. We consider how K is determined and how it is transmitted to the other components. The problem of an environment which cannot be assumed to be trustworthy during the initialization process of the system can only be solved by implementing a system using asymmetric encryption or by realizations providing a server. In the first case a *master component* could distribute the system key securely by using the components' public keys. If there is no *master component* a system component generates the system key and distributes it securely to all other components. In a server solution both, the use of asymmetric and symmetric encryption schemes. The server distributes the system key either encrypted by the component's public keys or by their secret keys which the server would need to share with all components a priori. One application could be cell phones. Alice puts all parts of her cell phone together. The cell phone then connects to the server of the provider which initializes all components. If there is no server connection one component needs to generate a system key. The component then encrypts this system key with the public key of all other system components and sends it to them.

5. **Use of timestamps or random numbers**

As mentioned in Chapter 3 the use of timestamps reduces the number of protocol messages. Instead of a challenge and a response message, only one message containing a timestamp is necessary. Unfortunately timestamps are not easy to implement at all. According to [23] a lot of assumptions have

to be made which are not easily to ensure. Especially the synchronization of the clocks off all participants is hard to achieve. We cite [23] the following: *... maintaining the synchronization of clocks in a secure and practical way generally requires a secure authentication scheme, basing the latter on the availability of synchronized clocks is somewhat dubious.* Furthermore clocks represent a point for attacks, e.g., by manipulating the clock time to execute replay attacks. Providing a secure clock involves a high level of tamper resistance of the component and a time distribution protocol which withstand malicious attacks. Thus only the use of random numbers in challenge-response based authentifications is considered in our presented protocols. However, if a secure implementation of timestamps can be guaranteed the protocols can easily be customized by exchanging the challenge-response messages by one timestamp message.

5.6.2 Parameters which do not Affect the Protocol Flow

Now we consider the parameters which do not affect the protocol flow. They can be used to optimize the implementation of a solution to reduce the number of protocol steps can be reduced, to accelerate the execution or to the provide a higher security level.

6. **Protocol is executed in a serial or parallel fashion**

   In some realizations it might be possible to execute protocol steps in parallel. This is highly dependent on the system's environment, e.g., if the data bus is able to send more than one message at the same time. The concurrently execution of protocol steps leads to an acceleration of the protocol execution time but may be vulnerable to new attacks (see attack 3 in Section 3.3).

7. **Sanctions in the case of failed authentications**

   In our protocols we do not consider measures after a failed authentication. Before implementing a system it has to be defined when measures are taken and of what kind they are. This policy regulates the security level of an implementation. It allows for flexible security levels. For example, a high security level enforces a complete stop after one failure. It has to be regulated who is setting the alarm and who takes further measures, who will be alarmed,

how do these measures looks like, and how can be guaranteed that an attacker cannot interrupt those sanctions.

8. **Trigger for the execution of the "system check"**

   The procedures *assembly* and *disassembly* will always start as soon as a component is built into or removed out of a system. However, a trigger for starting the *system check* has to be defined since no particular event has preceded the *system check*. The choice of this parameter defines how often the procedure is executed. It does not affect the protocol flow. Because there is no trustworthy person in most systems for starting the check manually, another trigger is necessary. An event within the system has to be used for starting the *system check*. Note that all following variants can be combined with each other. For instance, the check could be executed when the system is started, periodically in the running system, and every time an authorized person wishes to check the system integrity.

   a) *Every time the system is started*

      This option is only for systems which are running temporarily. The system only starts when it has not been altered in between the last and the current check. For instance, this solution could be convenient for vehicles. The safety of the occupants can be guaranteed, and malicious owner using stolen parts can be punished in a way that their car will not start anymore. Unauthorized changes in the running system, e.g. by exchanging parts, would be detected the next time the system starts. However, this solution is not satisfactory if an attacker can easily exchange parts in the running system. Malory would first use the original part to pass the check, then exchange this component with a bogus one, and finally re-build the original one before the next system check is started. This is feasible in vehicles but is hard to accomplish because Malory would have to hold both, an original and a bogus part. Therefore, in an implementation with this parameter it should be unpractical and expensive to exchange parts in the running system.

   b) *The system check is done periodically*

      This solution is suited to the case where the users can easily exchange components in the running system. For instance, consumer goods, as

printer cartridges, are worth it to exchange them all the time. The original component could be held for authentication purposes only and the forged one is used in the running system. Malory owns a printer and keeps the original ink cartridge which the printer is delivered with. Then he buys cheaper counterfeits for printing. The original cartridge is used to pass the *system check* and then replaced by its counterfeit. Every time the *system check* is executed Malory has to exchange the components. By a periodic check it becomes highly inconvenient for the attacker to exchange the components all the time. The higher the frequency of the check the higher the security level of the system. The length of a period can be expressed by time or clock cycles as introduced in the following.

i. *Started periodically dependent on the time*
In systems with timer the protocol could start after a certain time. This scenario is pretty rare.

ii. *Started periodically dependent on the clock cycles*
The verifier or a system component could start the procedure after a certain amount of clock cycles after the last execution of the check. This implementation is securer than the one described in the previous paragraph because the clock cycles are harder to tamper with than with a system clock.

c) *Procedure is triggered manually*

In this scenario the user starts the system check. The user must be trustworthy because an attacker would never start the test. This scenario could be combined with the choice of one of the previous variants 8a and 8b. For instance, Alice could perform a system check of her car. If Alice wants to check if the mechanics really did what they charged her for she starts a system check. If Alice wants to sell her car to Bob she can to him that her car consists of authorized components only.

5.6.3 Dependencies Between the Parameters

Not all parameter can be set independently of the choices of other parameters, i.e., the selection of one particular parameter may restrict the choices of another parameter. We consider now the dependencies which exist between some parameters, which are also given in the Tables 5.2, 5.3 and 5.4.

Parameter		Encryption schemes	
		symmetric	asymmetric
2	a	o	x
	b	x	x
	c	o	x

Symbols

x: possible
o: impossible

Table 5.2: Dependencies between the encryption scheme and the provided component data

The first table shows the dependencies between the used encryption scheme and the possible choices of parameter 2, namely the client's data that the verifier holds. As shown in the table, the use of an asymmetric encryption scheme is always possible. This is due to the fact that in realizations using certificates the knowledge of keys a priori of all potential participants is not necessary. Since this is required for symmetric schemes, scenarios with chosen parameter 2a and 2c are impossible.

Table 5.3 shows the recommended scenarios for off-line systems. Again this depends on the choice of parameter 2. An off-line realization with parameter 2a is not recommended since the CRL cannot be updated. The *used list* would only prevent the use of more than one clone in the same system. A realization with parameter 2b is better since the *ready for assembly-list* allows only components to pass if they are explicit on the list, in contrast to the CRL which allows all components to pass if they do not have a record on the list. The most recommended choice is variant 2c since it provides the same properties as the previous scenario with avoiding the redundant implementation of the *used list*. The *ready for assembly-list* in an off-line realization would only contain the records of components which are authorized for the assembly in a particular system, i.e., the considered off-line system.

The last table shows the dependencies of the used encryption scheme, namely asymmetric or symmetric encryption, parameter 1 and parameter 4. For the choice of parameter 1 only two different cases are considered, the system is either on-line or off-line whereby it does not matter if the connection to the server is provided permanently or temporarily. Parameter 4 describes if the environment the first assembly of the system takes place is assumed to be secure or non-secure. It can be seen in the table that all realizations except of one are possible. Only the implementation of a solution in a server-less system using symmetric encryption is impossible. To establish the system key in a non-secure environment the key has to be distributed securely. Since there is no server to distribute the key the components have to per-

Parameter		1c
2	a	-
	b	o
	c	+

Symbols

+: recommended
o: ok
-: not recommended

Table 5.3: Recommended scenarios for off-line systems

Parameter			Encryption schemes	
			symmetric	asymmetric
1 a,b	4	a	x	x
		b	x	x
1 c	4	a	x	x
		b	o	x

Symbols

x: possible
o: impossible

Table 5.4: Dependencies between the encryption scheme, on-/off-line connections, and the environment of the first assembly

form this among each other without the help of external entities. The components do not share a secret and thus are not able to transmit the key encrypted.

6 Protocol "System Check"

The protocol provides the monitoring of the system integrity as already described in Section 4.2. The protocol is based on the idea that all system components share a secret which they can use for authenticating themselves among each other and for securely communicating within the system. The protocol of the *system check* covers all periods of a component's life cycle, whereby the life cycle comprises the entire duration a component is a part of the same system. The periods describe the different stages the component passes through in one system. In all protocols which are presented in the following the life cycle of a component is divided into three periods, the first period is the *assembly* of the component, the second one is named *running system* and spans the time a components is member of one system, and the third period is the *disassembly* out of the system. The three periods and the assigned procedures of the entire protocol can be seen in Figure 6.1. For the assembly only one procedure has to be executed, namely the *key initialization* which is a simple key transmission and is described in detail in Section 6.3.1. During the execution of this procedure the component is initialized with the system key. In the *running system* the procedure of the *system check* is executed. This procedure deals with the integrity check of the system by searching for modifications and proofing if all system components are valid members of the system. All components are checked for their knowledge of the system key and it is referred to Section 6.4 for all details. In the last period the procedure of the *disassembly* is executed. This procedure is needed to obliterate all (or at least some) associations between the component and the system to enable the legally re-use of the component in other systems. The procedure is explained in Section 6.5.

6.1 The Participants

Before introducing the actual protocol we consider all kinds of participants which might take part in it. All participants are briefly introduced by listing their nota-

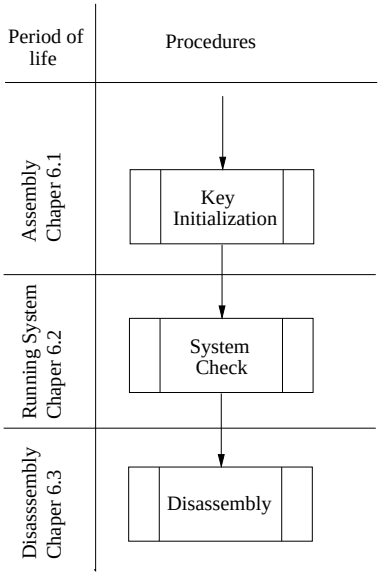


Figure 6.1: "System check": Life cycle of a component

tions and the data they hold. In addition all kinds of communications between the participants are considered, i.e. who is communicating securely with whom and who does communicate in plaintext.
The possible participants are:

- *New components*

  are components which are to be assembled into the system and referred to as C_{new} in the protocol hold the following data:

Component	Data
C_{new}	ID_{new}

  ID notifies a unique and also public identifier which is used to address the component. Since the component does not hold any keys it can only communicate unencrypted with system components and other entities.

- *System components*

  are components which are already part of the system. They are referred to as C_i, C_{old}, or C_{sys} respectively in the protocol and holding the following data:

Component	Data
C_i	K, ID_i, ID-List $\forall i \in 1, \ldots, n$

The ID is still from the basic state of the component (see new component), whereas K and the ID-list are received during the initialization process of the entire system or of the single component. K is the secret key of the system and static (except of key updates). The ID-list consists of the IDs, and perhaps status and more information about all components which are currently in the system. This list is refreshed every time a system modification is recognized, e.g. during the *system check* or the assembly of a new component. All system components and other entities which are in possession of the system key K are communicating securely with each other by encrypting and decrypting the messages with K.

- *The secret holder*

  is an entity which initializes the system components with the system key K during their assembly. Thus the secret holder needs to hold the secret system key K. It is responsible to either start the procedure of the *key initialization* or to pass K directly to all new authorized components.

Entity	Data
SH	K, ID-list[1]

  The entity has to be trustworthy because it is responsible for the distinction between authorized and unauthorized components. This distinction is not based on secret data and does not involve the use of cryptographic protocols. The secret holder might be divided in one instance which actually holds K and one who decides if the component is authorized for assembly or not. The secret holder could be a smart card personalized with a PIN, so that the card holder has to decide which components are authorized and then type in the PIN to start the procedure of initialization. Another possible scenario could be a password protected server holding K. It is assumed that the procedure of the key initialization can only be started by an authorized entity, however this is not part of the protocol and has to be ensured before. The *secret holder* might also hold the ID-list which would enable it to directly address all system members. This could be used for selecting a random component out of the system. Note that the ID-list has to be updated every time the system is altered. The SH can communicate securely with all system members by using K and in plaintext with new components.

[1] The ID-list is optional

- *The master component*

  A *master component* is necessary in realizations where the *secret holder* can start the *key initialization* but not perform the actual key transmission. If the *secret holder* is able to perform both, the use of a *master component* becomes redundant.

Entity	Data
MC	K, ID-list

6.2 Getting Started

For performing the *system check* all system components need to be distinguishable and hold a secret as proof that there are a valid member of the system. First is realized with the use of unique IDs which are set during the manufacturing of all components and hence are hold a priori before any of the protocols are executed. Latter is realized by the knowledge of the secret system key K. The system key is set during the very first initialization of the entire system or the initialization of a single component which is added subsequently to the system. For preparation purposes at least one component or one external entity has to hold the secret key to initialize others with this key data. This entity or component which holds the secret key data is referred to *secret holder* or *master component* respectively. Since all assemblies are assumed to take place in a secure environment the secret is passed in plaintext to the new components.

6.3 Assembly of a Component

For the assembly of a new component into a system the execution of only one procedure, namely the *key initialization*, is required. Since the verification if a component is authorized or not for the assembly is performed before, the protocol of the assembly deals only with the set up of the component. By performing the procedure of *key initialization* the new component is set up for a specific system.

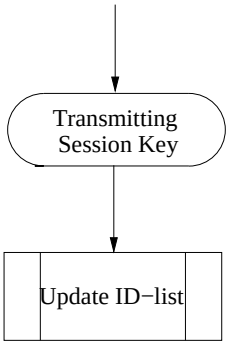


Figure 6.2: “System check”: Protocol flow *key initialization*

6.3.1 Key Initialization

Short overview
Objective: Initialization of all authorized components with the system key K
Purpose: Providing secure communication within the system and enabling component authentications among all system components
Executive entity: SH or system component

The procedure of the *key initialization* consists of the actual key transmission and the update off all ID-lists hold by the system components which can be seen in Figure 6.2. Since the environment in which the initialization takes place is assumed to be secure, the system key is transmitted in plaintext to the new component. The key initialization can take place outside of the system or within the new system. In the first realization the initialization is performed by the SH. All system components need to be informed about the new member to update their ID-lists. In the latter case, the initialization within the system, the system is first notified about a new authorized component which is newly built into the system. The system, or more precisely one random system component, then performs the actual key initialization and the update of the ID-lists hold by each system component.

The general protocol is described in the following table, whereby general means the implementation of all protocol steps independent of all selectable parameters. The entire procedure consists of two protocol steps only. The first one is the execution of the key transmission, where the secret key is send to the new assembled component. The second step is performed to update the ID-list of all current system components by adding the record of the new component to the list. The particular parameters which can be chosen for this procedure are given below the general protocol. For

clarity an example of a realization with a specific set of selected parameters is given.

General protocol:

1. Transmission of the system key K.
2. Secure (encrypted with the system key) and "fresh" update of the ID-list of each system component.

Assumptions:
The following assumptions have to be made in addition to the general ones given in Section 5.5.

1. Only trustworthy entities can start the *key initialization*.
2. The assembly takes place in a secure environment.

Parameters:
We consider all parameters which need to be set for this procedure. Fragments of the protocol flow of how the protocol would look like for possible choices of the parameter is given. Note that in the combination of those fragments some steps might become redundant or some extra steps become necessary. Which combinations are most favorable or not recommendable can be seen in Table 6.1. The criteria for the suggested implementation given in this table are based on the number of protocol steps or the computations which can be saved due to a clever combination of the parameters. In this procedure all combinations are possible, but sometimes only with large expense. Since the environment the execution of the procedure takes place is assumed to be secure no encryption and no use of challenge-response protocols for ensuring fresh updates are necessary.

1. **Transmitter of the system key**

   This parameter defines which entity actually transmits the system key K to the new component.

   a) *the secret holder*

      An authorized person/entity starts the procedure and the *secret holder* transmits the system key K as plaintext to the new component. The new component stores the received key.

1. $SH :\longleftarrow C_{new}$: hello(), ID_{new}
2. $SH :\longrightarrow C_{new} : K$

b) *a system component*

This parameter has to be set when the secret holder is responsible for the first assembly of the entire system only but not for the initialization of subsequently added parts. The choice of this parameter is mandatory if there is no SH at all. When the procedure is started by an authorized person/entity a random system component has to be chosen and prompted to execute the key transmission. The random component can be selected by authorized external entity or among the system components themselves without any external help. How to select a random component is explained in Section 5.6.1 in parameter 3. One realization with SH which holds the system's ID-list is that the SH could directly select a random system component out of the system and prompt it to execute the *key initialization* with the requesting component.

1. SH: $\longleftarrow C_{new}$: hello(), ID_{new}
2. Selection of a random system component C_{sys}
3. $C_{sys} :\longleftarrow SH : KeyInit, ID_{new}$
4. $C_{sys} :\longrightarrow C_{new} : K$

2. **Update of the ID-list**

After C_{new} : received K all components of the system have to be notified about the new member to update their ID-lists. To prevent attacks using fraud ID-lists the update of the ID-list can only be performed right after the key transmission and not separately. Additionally to the update of the ID-lists hold by "old" system components, the new one needs to receive the entire ID-list of the system. For this transmission the selection of a random system component for acting as transmitter is required. The considered variants differ only in the entity which notifies the system about the new component.

a) *executed by the secret holder*

For a scenario with set parameter 1a this is the favorable choice for this parameter since the secret holder has been already in possession of ID_{new}. Otherwise (parameter 1b) the *secret holder* has to be notified

about ID_{new} by the server or C_{sys} before, as can be seen in step 1 of the below presented protocol segment.

The *secret holder* sends ID_{new} and maybe further data of the new component to all system components. The message can be send encrypted or as plaintext. Each system component then updates its ID-list by adding the record of the new component. One system component then sends the current ID-list to C_{new} which stores it.

1. $S/C_{sys} \longrightarrow$ SH: ID_{new}
2. SH $\longrightarrow C_i : ID_{new} \forall i \in 1, \ldots, n$
3. Selection of a random system component C_{sys}
4. $C_{sys} \longrightarrow C_{new} : ID - list$

b) *executed by the new component*

In combination with parameter 1a a random system component has to be chosen (step 2 of the below presented protocol), in the other case this has been already performed during the steps of the key transmission.

The new component which has already received K notifies the system about its assembly by sending its own record. Each system component then updates its ID-list by adding this record to the list. After that one system component C_{sys} sends the current ID-list to C_{new} which stores it.

1. $C_{new} \longrightarrow C_i : ID_{new} \forall i \in 1, \ldots, n$
2. Selection of a random system component C_{sys}
3. $C_{sys} \longrightarrow C_{new} : ID - list$

c) *executed by the server*

This scenario is again favorable for set parameter 1b since a random system component has been already chosen and step 2 of the presented protocol would be redundant. The server not only performs the notification of the particular system component, it also informs all system components about the new member ID_{new}. Each component then updates its ID-list. After that C_{sys} sends the new ID-list to C_{new} which stores it.

1. $S \longrightarrow C_i : ID_{new} \forall i \in 1, \ldots, n$
2. Selection of a random system component C_{sys}

Parameter		1	
		a	b
2	a	++	o
	b	o	+
	c	o	+
	d	o	++

Legend

++: very recommendable
+: recommendable
o: possible
–: impossible

Table 6.1: parameter dependencies: *key initialization*

3. $C_{sys} \longrightarrow C_{new} : ID - list$

d) *executed by a system component*

This scenario is highly recommended for set parameter 1b since a random system component C_{sys} has been already chosen *and* is in possession of ID_{new} and thus step 1 and 3 of the below presented protocol fragment would become redundant.

1. $C_{sys} \longleftarrow SH : ID_{new}$
2. $C_{sys} \longrightarrow C_i : ID_{new} \forall i \in 1, \ldots, n$
3. Selection of a random system component C_{sys}
4. $C_{sys} \longrightarrow C_{new} : ID - list$

Example

In this example a protocol for the chosen parameter 1a and 2a is presented, because this scenario is one of the two preferred realizations (see Table 6.1). This could be a scenario with a personalized smart card which acts as the *secret holder* and is used by the card holder to start the procedure and initialize the new component. The new component is connected to the smart card, for instance C_{new} is assembled in to the system and the smart card is also connected to the system via an interface, which is provided for this purpose.

Protocol messages

1. $SH :\longleftarrow C_{new}$: hello(), ID_{new}

   $SH :\longrightarrow C_{new} : K$

2. SH $\longrightarrow C_i : ID_{new}, x \forall i \in 1, \ldots, n$

   $C_{sys} \longrightarrow C_{new}$: ID-list

Protocol steps:

1. C_{new} send a *hello* message together with its identifier to the smartcard to start the procedure.

   The smartcard returns the secret system key K which is then stored by the component.

2. The smartcard then sends a message to all system components to inform them about the new system member by sending ID_{new}. It also sends a random value x to select a random component out of the system.

   For updating their ID-list all components add the ID_{new} to their ID-list. For determining a random C_{sys} each component C_i computes:

   $ID_i \oplus x \forall i \in 1, \ldots, n$

   The component C_i which assigned ID_i obtains the smallest result is the selected component C_{sys}. C_{sys} sends the current ID-list of the system to the new component which stores it. C_{new} is now a valid member of the system.

6.4 The Running System

Short overview
Objective: Recognition and notification of unauthorized system modifications
Purpose: Preservation of the system integrity
Executive entity: System component

In the running system the system is checked for integrity by monitoring unauthorized modifications. Therefore all system components are checked after a specific time period or after a random time period to check if they are still in the system and still

active. The objective is to notice all components that were removed or exchanged in an unauthorized fashion. Unauthorized added parts do not pose a threat, since they will not receive the system key and are not listed in the ID-list. First follows that the component cannot listen to the communication within the system because all system messages are encrypted by the system key. Secondly, the fact that the added components do not have a record on the ID-list, it follows that these components will not be challenged during the *system check*. Regardless of the realization (e.g. with server or without server) the *system check* is always executed by a randomly chosen system component because the knowledge of the system key is required to perform the check. Due to the design criteria of the protocols (Section 5.3) the check should be executed independent of a server or other external entities. The check makes use of the fact that all system components are in possession of the system key K. A component holding K is trustworthy since K can only be received after its successful authentication. All components hold the ID-list which specifies the ID and status of all valid system members. The chosen verifier has to check all components on this list to make sure that the entire system is unaltered. All components which hold a record on the ID-list are challenged by the verifier. If the component is still a part of the system it responses in a correct manner. An invalid response or no response means that the component is missing or exchanged by an unauthorized component. In these cases the verifier sets an alarm. The notification of violations to the system or to the protocol flow are as important as the recognization of these violations itself. The system might set an alarm, stop the system, and take further measures. The non-interception of the alarms has to be ensured. When the *system check* is started, the first step of the protocol deals with choosing a random system component for executing further protocol steps. In the preferred realization no server (or equivalent entities) is involved and thus the selection of this particular component has to be performed without any external help. Independent of the actual scenario, all system components on the current ID-list have to be checked if they are still present and in possession of the system key. This is realized using a challenge-response protocol. This is described in step 2 of the general protocol and can be realized in two different variants as shown in Figure 6.3 and 6.4. After successfully executing the protocol all components of the system either have been authenticated successfully or have been added to a list of non-authenticated IDs, referred to as !ID-list in the protocol. Step 3 of the general protocol deals with the update of the ID-lists of all valid system members, i.e. all components which passed this check, have to be updated. It has

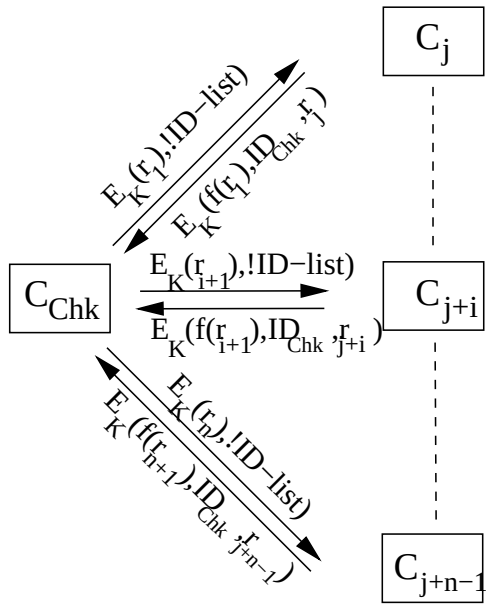


Figure 6.3: "System check": Protocol flow *system check* with one verifier

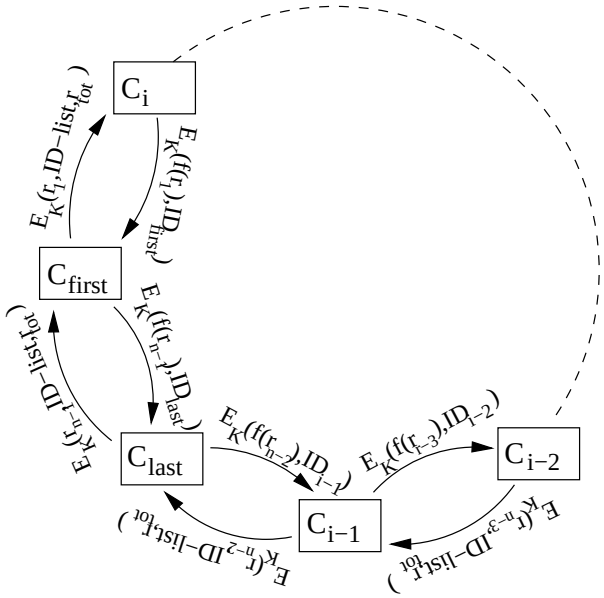


Figure 6.4: "System check": Protocol flow *system check* with rotary verifiers

to be ensured that this "update message" cannot be altered (by encrypting it with K) and not be replayed (by using fresh challenges every time).

General protocol:

1. A system component is randomly chosen as verifier C_{first}.
2. All system components are challenged.

   All system components respond.
3. The ID-list of all system components is securely updated.

Parameters:

We consider now all parameters which need to be set up for the implementation of the *system check*. There are some which do affect the protocol flow, namely 3, 5, and some which do not, 1, 4, 6. Only three parameters are interdependent, the choice of the structure of the verification, the fashion of the protocol execution and the sanctions in the case of one or more failed authentication/s. The interdependencies of these parameters are shown in Figure 6.2.

1. **Periodicity of execution**

   The *system check* procedure needs a specific trigger to be started. It is referred to the general descriptions of this parameter in Section 5.6.2 parameter 8 to read more about all variants and to see which one is best suited for a system.

   a) *Every time the system is started*

   b) *Periodically*

   c) *Procedure is triggered manually*

2. **Selection of a random system component for starting the check**

   It is referred to the Section 5.6.1 parameter 3 for details.

3. **Structure of verification**

   a) *One verifier*

   One component acting as verifier C_{Ver} is challenging all the others. After receiving all responses the verifier sends an update of the ID-list to all participants. To be able to ensure the freshness of the update, all responses of all components contain again a random number. These numbers are added to the update message. Hence each component can verify if its challenge is part of the received update and thus proved to be fresh.

   1. choosing random component C_{ver}
   2. $C_{Ver} \longrightarrow C_i : E_K(r_j,$!ID-list$), \forall i \in 1, \ldots, n$
      $C_{Ver} \longleftarrow C_i : E_K(f(r_j), ID_{Ver}, r_{j+1})$, $j++$ after each challenge
   3. $C_{Ver} \longrightarrow C_i : E_K($!ID-List$, f(r_n))$, with $f(r_n)$ as a combination of all used r_{j+1}

   b) *Rotary verifiers*

   One randomly chosen component C_{first} is starting the procedure. All components are challenging each other subsequently until C_{first} is challenged by the "last" component. During the loop all participants are adding a random value to a so called collective random number which is used as proof for freshness of the ID-list update. This collective random number, referred to as $f(r_n)$ is part of the update message send by C_{first} to all components which passed the check.

1. choosing random component C_{first}
2. $C_i \longrightarrow C_{i+1} : E_K(r_i, !ID - list, f(r_j))$, $\forall i \in 1, \ldots, n$, with $f(r_j)$ as a combination of all previous used r_j
   $C_i \longleftarrow C_{i+1} : E_K(f(r_i), ID_i)$, $i{+}{+}$ after each challenge
3. $C_{Ver} \longrightarrow C_i : E_K(!\text{ID-List}, f(r_n))$, with $f(r_n)$ as a combination of all used r_j

4. **Protocol is executed in a serial or parallel fashion**

   This parameter slightly affects the protocol flow, but increases the speed of the execution. It has to be kept in mind that new kinds of attacks might become possible due to the parallel operations. This parameter is affected by the choice of parameter 3 and 6.

   a) *Parallel protocol steps*

   Note that for executing several steps at once, the technical environment, e.g. the communication bus, must be suited for the parallel sending of protocol messages. In the variant with one verifier (parameter 3a) all challenges could be performed at the same time, as well as the responses and the update of all components' ID-lists. This would highly reduce the execution time of the check. For scenarios with rotary verifiers 3b the protocol flow would have to be modified since the total random number for ensuring a fresh update cannot be computed in the presented manner as well as the update of the list of failed authentications would have to be performed differently.

   b) *Serial protocol steps*

   This is mandatory for all systems where the parallel sending of messages is not supported. In this case each component is challenged separately. Although the sending of parallel messages are not possible, *multicast messages* might be supported by the system. That means the same messages is send to all components, e.g. all entities connected to the data bus. The update of the ID-list could then be performed as *multicast message*.

5. **Ensuring the freshness of the ID-list update**

   Without ensuring the freshness of the list, an attacker could replay an old ID-list to all system components. This entails that all components assembled

after the last update would not be recorded on the list. Thus these components would not being checked during the *system check*. Malory could simple exchange or modify all unrecorded components unnoticed by the system.

a) *List of challenges*

This realization is suitable for systems with a small number of components. A list of all challenges could be generated during the protocol execution, e.g. by adding the respective challenge to the list before sending it to the next challenged component. The list, referred to as $f(r_n)$ in the protocol segments presented in parameter 3, could be part of the updating message of the ID-list. Thus each system component would be able to prove the freshness of the received update by performing a look-up on this list.

b) *Challenge-response*

This realization is also suitable for systems with a small number of components. One component, e.g., the verifier C_{Ver} in scenarios with parameter 3a, could execute a challenge-response protocol with each system component. This assures all components the freshness of the received ID-list, i.e. that the list is not replayed.

c) *Total challenge*

This scenario is practical for implementations where the used challenges r are small. Each component in a realization with set parameter 3b or the single verifier in a scenario with parameter 3a multiplies the challenge by all previous challenges. After receiving the new ID-list each component is able to prove the freshness of the message by dividing the total challenge, i.e. the product of all single challenges, by its own challenge. If all components can calculate this without remainder, the message is proven to be fresh.

6. **Sanctions in the case of failed authentications**

This parameter defines how the system reacts to failures during the execution of the protocol. The parameter depends on the structure of the verification as defined in parameter 3. It also depends on the kind of execution of the protocol steps defined in parameter 4. Latter is not directly shown in Figure 6.2 but

Parameter		3	
		a	b
6	a	o	++
	b	–	+
	c	o	o
4	a	o	o
	b	+	–

Legend

++: very recommendable
+: recommandable
o: possible
–: impossible

Table 6.2: Parameter dependencies: *running system*

can be derived from the dependencies between the choice of the structure and the kind of execution.

a) *highest security level*

The protocol execution stops immediately after the component which acted as verifier sets an alarm and takes measures due to an authentication failure. In the case of implementation with parameter 3b the current verifier "knows" the ID of the component which authentication was rejected and can alarm all other components and take measures.

In the other variant with parameter 3a, one verifier could check all received responses one by one and stop immediately after detecting one failure. In contrast to the previous case, a component which is not responding could first be recognized after receiving all other responses and a certain time-out when the procedure stops.

b) *medium security level*

This variant is suited to assumptions made as parameter 3b only. Instead of stopping when *one* authentication fails, each claimant (i.e. in each step) checks the current ID-List update (!ID-list in the challenge message) and proves if it has to take measures by checking its policy.

c) *lowest security level*

All components check the ID-list after completely checking all components, i.e. completely executing the system check. Each of them takes measures as regulated in their security policies depending on the test result.

Example

The protocol is highly affected by the many parameters which can be set. For the below presented example we assume that the components take turns in challenging each other (parameter 3a) and that the freshness of the ID-list update is guaranteed by generating a list of challenges of all components which take part in the check (parameter 5a). Furthermore the system supports sending *multicast messages*.

Notation:
i: consecutive index of the system components, $i \in (1, \ldots, n)$
j: consecutive index of the challenges, $j \in (1, \ldots, 2n)$
$list(r_j)$: the collective random number, which is in this example a simple list of all challenges r_j. Each challenge r_j is added to the list by the verifier

Protocol messages

1. $C_i :\longrightarrow C_j : E_K(r_i), \forall i, j \in 1, \cdots, n$ and $i \neq j$
2. $C_i :\longrightarrow C_{i+1} : E_K(r_j, !\text{ID-list}, list(r_j))$
3. a) $C_i :\longleftarrow C_{i+1} : E_K(f(r_j), ID_i)$
   b) until $i < n$: $i++, j++$, jump to step 2)
   Loop is executed until C_{first} is authenticated by the "last" component.
4. $C_{first} :\longrightarrow$ *multi-cast message*: $E_K(!\text{ID-List}, list(r_x))\ \forall x$

Protocol steps:

1. Each system component generates a random number r_i and sends it to all other components on the ID-list. It then computes the value x which depends on all r_i ($x = g(r_i)$).

   This value x is taken to select a component by XOR-ing it to all IDs on the current ID-list. The component with the ID which obtains the lowest result is chosen to start the system check.

2. Starting the loop:

   The verifier C_i selects a random number r_j and puts it on the list $list(r_x)$. For challenging the "next" component, C_i encrypts r_j combined with the current

list of failed authentications !ID-list and $list(r_j)$. It then sends the result to C_{i+1}.

3. a) C_{i+1} decrypts the received message and encrypts the modified challenge r_j together with the verifiers ID and sends it back to C_i.

   b) In each iteration i and j have to be incremented. A jump to step 2) is executed until the "last" component has verified the identity of C_{first}.

4. After the execution of the system check C_{first} holds a complete list of all components which failed the check and also a list $f(r_j)$ of all challenges r_j used by the verifiers C_i of each step.

   C_{first} sends a list of all components which failed the test as an update for the ID-list. It also sends a list of all challenges $f(r_j)$ encrypted with the system key K to all components.

   The system components decrypt the received message and compare if their challenge value matches one of the r_j's on the list. If so, they update their such that they change the status of all IDs which are on the !ID-list.

   After the update each component compares the current ID-list with its policy. Measures are taken by all single components according to the individual security policies.

6.5 Disassembly of a Component

The implementation of the *disassembly* highly depends on the check for authorization before a component is assembled into a system. The procedure of *disassembly* is only required if a component should re-usuable in other systems, i.e. the component is removed out of its system and then re-build into another one. If the components needs to meet some criteria for the re-assembly, these requirements have to be set during the component's disassembly. From the pre-definition of the authorization check, which is executed before the actual assembly starts, follows the implementation of the *disassembly*. In some realizations the component might need a specific key value, e.g. a specific initial value, or has to hold other pre-defined data. In this case the procedure of *disassembly* would have to be executed to re-set or modify the key memory and/or other memories of the component. If the authorization of a component is verified by a person or machine who/which is building it into the

system, no procedure for the disassembly is necessary. The fresh assembled component would just receive the system key of the new system and overwrite the key of the previous system, the component was a part of before. Since the criteria to distinguish between authorized and un-authorized components were not regarded in this solution, the procedure of the *disassembly* is not considered any further here as well.

7 Protocol "Proof of Origin"

The protocol *proof of origin* provides piracy protection. After the protocol execution the user knows if the checked component is an original part or a counterfeit. For an overview about the solution and suited applications we refer to Section 4.1. One characteristic of suited applications is that the participants form transient systems only. Thus nothing like a specific system's secret, e.g. a secret key, has to be set up among the system component. For more complex services which require many exchanged protocol messages a session key for the particular sessions is generated and used during the session to secure the communication. Otherwise the keys hold by the components for authentication purposes anyway could be used for securely communicating as well. The protocol for this solution can be implemented with the use of symmetric or asymmetric encryption. When using symmetric encryption the component holds a secret key for its authentication, which might be also used for encrypting further messages. The server needs to hold a list of all authorized components a priori. If asymmetric encryption schemes are used the components prove their authentication by using private and public key pairs. The server needs to verify the certificates by using the issuer's public key. The server holds a list of all revoked certificates. For the implementation of both encryption schemes the protocol mainly consists of the authentication of the component to the verifier. The authentication process is considered for the symmetric and asymmetric solution in the Sections about the *assembly* 7.1.3 and 7.2.3 respectively. Figure 7.1 shows all necessary procedures assigned to the period of a component's "life" in one system. As above mentioned a procedure for establishing a session key might also be a part of the protocol of the *assembly*. During the period the component is part of the same system, referred to as *running system*, no execution of extra procedures are required due to the transient associations. Dependent on the actual realization a procedure of *disassembly* might be needed.

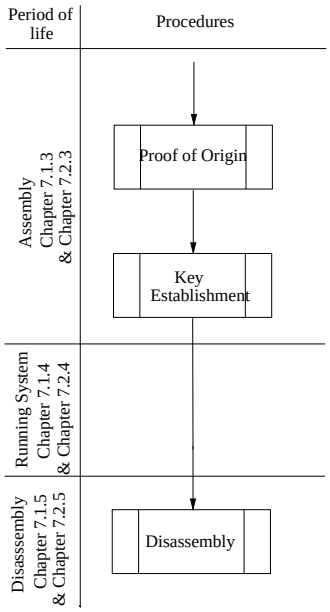


Figure 7.1: “Proof of origin”: life cycle of a component

7.1 Symmetric Solution

The symmetric implementation of the *proof of origin* provides piracy protection by verifying the authenticity of components *and* the checking for cloned components. The first check ensures that the component holds data from an authorized trademark or organization. The second check then proves if the component is really an original part and not a clone holding the original data.

7.1.1 The Participants

In this solution components are a part of a system for a short period only. In most applications there is one server or a representative entity acting as verifier and a component acting as client. The solution deals mainly with one component asking one verifier for temporary authorization to use a special service. The server acts as verifier and proofs the request. There might be some realizations without a permanent on-line server. In these scenarios a representative for the server is necessary to act as server. These representatives could be a High Security Module (HSM) or another entity holding the required data for performing the authentication process. In the following a HSM represents all entities which are required in an temporary on-line or off-line realization for taking over the role of the server.

The participants are:

- *New components*

  are components which are to be assembled into the system and referred to as C_{new} in the protocol. They hold the following data:

Component	Data
C_{new}	$ID_{new}, K_{new}, counterreading(t)$[1]

  The ID notifies a unique public identifier which is used to address the component. K_{new} is the component's secret key used to prove its authenticity. The key is also used for the secure communication between server and this component.

- *The server*

  is referred to as S or V (verifier). It is connected to the system and provides the following data of the considered system[2]:

Entity	Data
S	ID_{server} *ready for assembly-list*$(ID_j, K_j, \ldots)$

  The index j is a consecutive index to number the components on the list. Since the list is permanently altered, i.e. records of components are added, altered, or removed, the number of components is not determined.

- *The High Security Module*

  or short HSM, is referred to as V for verifier in the protocols and serves as a replacement for the server. The HSM is a permanent part of the system in temporary on-line or off-line systems and holds the server's data from the last time t_x the HSM was connected to the server. Thus it cannot be guaranteed that the data hold by the HSM equals the current data on the server. The HSM holds the following data:

Entity	Data
HSM	ID_{HSM}, *ready for assembly-list*$_{t_x}(ID_j, K_j)$, with $t_x \leq t$

  The provided data consists of an "old" ($t = t_x$) subset[3] of the servers data. Either the HSM has to be connected to the server any time a table look-up is

[1] Optional, required in realizations providing clone detection by using counters
[2] The server holds this kind of information for all other systems
[3] only the data of the system the HSM is a part of

performed or it is assumed that the data on the HSM is still equivalent to the current data on the server. In first case the HSM would be redundant because the server could directly perform all look-ups and other necessary steps for the verification of the component's authentications. Second bases on a strong assumption and reduces the security of the solution.

7.1.2 Getting Started

Since there is no permanent system consisting of the same components, no initialization of a system or its members to run the protocol is needed. The only preparation is to set up the components and the verifiers with the required data for the authorization process. Because of the transient associations no particular system key and thus no initialization of it is needed. The components are initialized with the required data during their manufactory. The verifiers are initialized before the protocol is started the first time. The required data for the execution of the entire protocol has already been introduced in Section 7.1.1.

7.1.3 Assembly of a Component

Since we are mostly talking about one client and one server temporarily building a system, it is rather a client getting connected to a server or another verifier than the assembly of a component into a system. The headline of this section is used for consistency and better comparison with the solutions in Chapter 6 and 8. When a component gets connected to a server it is prompted to identify itself. If the component can successfully prove its (valid) identity it is allowed to use services provided by the server to all authorized components. The realization is described in the next sub section which explains the procedure of the *proof of origin*. In the case that the exchange of many confidential messages is needed a session key might be generated and used during one session. This procedure is explained in Section 7.1.3.2.

7.1.3.1 Proof of Origin

Short overview
Objective: To verify if a component is from an authorized trademark or organization
Purpose: To avoid the use of confidential services by bogus parts
Executive entity: External verifier such as a server or HSM or random system component as internal verifier

After the execution of this procedure it is proven if the requesting component is authorized for its assembly or not. The requesting component C_i holds a secret key K_i to prove its authenticity to the verifier. The verifier holds a list of all authorized components (*ready for assembly-list*) to perform a look up if the component's record is valid. This action can be seen in Figure 7.2 in the first decision of the flow chart. The verifier performs a table look up to see if the component has a valid record on the *ready for assembly-list* and if the component is using the dedicated secret key. Besides this the verifier checks the freshness of the messages by using challenge-response which is referred to as *Authentication* in the general protocol and the second part of the first decision in the figure. Only if both checks are successful the protocol either stops or the protocol continues with generating the session key. In the first case the component starts to use the services. In the latter variant the procedure *key establishment* is started. If one or both check/s failed, an alarm is set by the verifier.

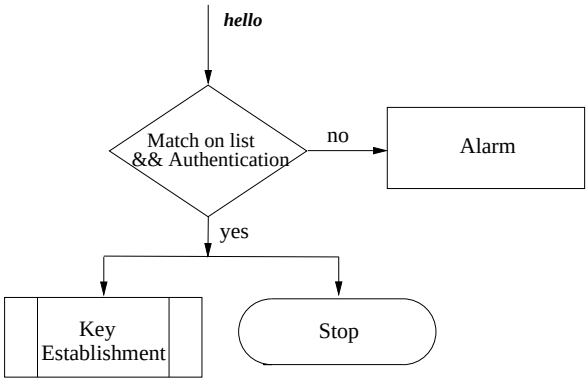


Figure 7.2: "Proof of origin", symmetric solution: Protocol flow *proof of origin*

General protocol

1. New assembled component says *hello* to the verifier.
2. Verifier checks if the component's record is on the *ready for assembly-list*:
   i. if there is a **match**:
      Authentication new component vs. verifier by challenge-response
      α) authentication accepted:
      → The protocol either stops and the authentication is accepted **or** continues with the *establishment of a session key*; see Section 7.1.3.2
      β) authentication rejected:
      → The protocol stops and further measures are taken.
   ii. if there is **no match**:
      → The protocol stops and further measures are taken.

Assumptions:
The following assumptions have to be made for the *proof of origin*.

1. The data hold by the HSM or other entities in a temporary on-line or off-line solution is assumed to be equivalent to the current data on the server.

Parameters:
There are only two parameters to choose for the *proof of origin*, namely the connectivity of the claimant to a server and the possibility of clone detection. Since the protocol flow will not change for the choice of the connectivity the example given below covers all cases. The dependencies of these two parameters are given in Table 7.1 and show, among others, that a detection of clones is impossible to implement for an off-line solution.

1. **On-line/ off-line connection**

   This parameter affects all protocol steps where a look-up on the *ready for assembly-list* is needed. In this procedure a look-up for checking if the component is authorized for the assembly is executed right in the beginning of the protocol and thus affects the protocol steps.

a) *Permanent on-line connection between the server and the system*

Since the system is always on-line look-ups of data is not an issue here.

b) *Temporary on-line connection*

The HSM or another entity acting as verifier performs a look-up on its *ready for assembly-list*. As mentioned in sub chapter 5.6.1 all cases of data integrity or non integrity between off-line verifier and server as given in Table 5.1 have to be considered.

Case 1a) and b) of the table would lead to step 2II. of the above presented general protocol. Since step 2II means that the component is not authorized and the protocol stops these cases do not have to be taken under special account. If the component is supposed to be authorized, the procedure simply has to be re-started after the data hold by the verifier is updated, e.g. the next time it is connected to the server.

Case 2a) and b) lead to step 2I. of the general protocol. Since this means the successful authentication of the component these cases are critical. All components are a part of a system for a short time only, thus no *waiting lists* as described in Section 5.6.1 parameter 1b can be provided. The next time the system might be connected to the server the component has been already disassembled and maybe built into several other systems. Thus nothing else remains to be done than to assume that the list hold by the HSM is still equivalent to the current list provided by the server. The more frequent the HSM is connected to the server and hence the more frequent the *ready for assembly-list* is updated, the higher is the security level provided by the system.

c) *off-line*

This implementation is not recommendable because the components record is usually not static. Since all components are frequently used in different systems it cannot keep track about the currently used original components, e.g. by the use of a *used list*, like in other solutions. But this would be necessary for detecting cloned components. To distinguish between original and cloned components without knowing where the originals are, additional arrangements have to be made for instance using the private key only once and then establish a new one, or using a counter for the times a component is used. Since an off-line scenario provides

only static data these changes could not be performed and the detection of clones is impossible. Hence this realization is not suited for the *proof of origin* if you want to provide this detection.

If Alice still wants to implement this scenario she could try to make the components impossible to clone, or at least that hard that it is not worth it for Malory to do so. Thus it could be assumed that there are no clones. All other counterfeits can still be detected. In realizations on this condition no doublets have to be detected and taking measures like performing key updates or incrementing counters are not necessary. This is regarded in Section 7.1.5 about the disassembly.

2. **Providing clone detection**

The parameter defines if the solution provides the detection of cloned parts or not. The solution always checks the authorization of components, but clones would pass this check, since their data equals the one hold by the originals. To prevent the use of clones, the data of an original part need to be changed after each successful authentication.

a) *Solution providing clone detection*

This scenario provides the detection of clones and is thus the recommended choice. For the realization two possible variants are given below.

i. Key update
In this variant the secret key K_i of the component C_i is updated every time the component is disassembled. Thus the protocol flow of the *assembly* is not affected by this choice at all. It is referred to the respective parameter in Section 7.1.5 to read about the effects this variant has on the procedure of the *disassembly*. The protocol flow of this variant can be seen in Figure 7.5.

ii. Counter
In this variant a counter is increased, or changed in another predefined way, when a component is disassembled. The counter reading is checked every time a component is built into a system. Thus the modification of the data takes place in the procedure of the *disassembly*, as in the previous variant. This leads to the protocol flow shown in Figure 7.5. The *proof of origin* has to be changed in a way

Parameter		1		
		a	b	c
2	a	++	+	–
	b	o	o	o

	Legend
++:	very recommendable
+:	recommendable
o:	possible
–:	impossible

Table 7.1: Parameter dependencies: *proof of origin*

that the counter reading is checked during the authentication. The counter reading is stored in the memory of each component and also part of the component's record on the *ready for assembly- list*. The protocol fragment for this scenario would look like follows, whereby $counterreading(t)$ terms the counter reading of a component at the time t:

C: $\longleftarrow$ V: $E_{K_C}(r), ID_V$

$C :\longrightarrow V : E_{K_C}(counterreading(t), f(r), ID_V)$

The verifier challenges the claimant with an encrypted random value. The claimant responses on its part with an encrypted message containing its current counter reading, the modified challenge and the verifier's ID. The verifier checks the response. If the claimant responded in the right manner it is proven to be in possession of its secret key K_C and the current counter reading. Both pieces of information plus the ensured freshness of the message successfully identifies the component.

b) *Solution without clone detection*

This variant is generally not recommended but the choice mandatory for off-line scenarios. Since for providing clone detection the altering of data on the client *and* the verifier side is needed this is the only possible realization for off-line systems. For all other realizations it is suggested to implement the clone detection. The protocol flow differs little from the above presented protocol in parameter 2(a)ii. Only the counter reading in the claimant's response is missing.

C: $\longleftarrow$ V: $E_{K_C}(r), ID_V$

$C :\longrightarrow V : E_{K_C}(f(r), ID_V)$

Example

The protocol flow for a realization with an on-line server as verifier (parameter 1a) and providing clone detection by the use of counters is presented below.

Protocol messages

1. C_{new}: $\longrightarrow$ S: $hello, ID_{new}$
2. C_{new}: $\longleftarrow$ S: $E_{K_{new}}(r_1), ID_{Server}$
   C_{new}: $\longrightarrow$ S: $E_{K_{new}}(counterreading, f(r_1), ID_{Server})$

Protocol actions

1. The new component sends a *hello* message together with its ID to the server.

2. The server performs a table look-up on the *ready for assembly list* to check the records for ID_{new}. If there is no match the protocol stops.

   In the case of a valid match the protocol continues with a challenge-response protocol for the authentication of C_{new} to the server. Therefore the server uses the secret key K_{new} which was assigned to the matching record on the *ready for assembly list* to encrypt a random number r_1 as challenge together with its own ID and sends the result to C_{new}.

   The component responses with its counter reading, the challenge $f(r_1)$ and the servers ID encrypted with its secret key.

   The server verifies the response and in the case of a successful authentication the protocol continues with the procedure of the *key initialization*. Otherwise the protocol stops and the server takes further measures, for instance erasing the component's record from the *ready for assembly list*.

7.1.3.2 Key Establishment

Short overview
Objective: Distributing of a session key K_{ses}
Purpose: Secure communication between component and verifier during one session
Executive entity: A verifier

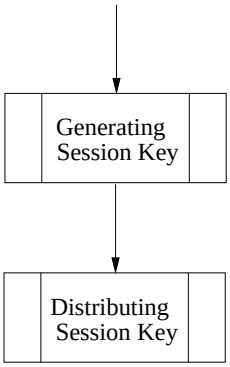


Figure 7.3: "Proof of origin", symmetric solution: Protocol flow *key initialization*

To provide confidential communication during one session it is recommended to use a session key, i.e. a secret key which is only used for encryption during the current session. A secret key should be never used for authentication purposes and for securing communication at the same time, because it might reveal some information about it. This and further motivations for the use of session keys can be found in Section 12.2.2. of [30]. The secret key is used for symmetric encryption and after establishing the key all following messages are encrypted by it. The procedure of the *key establishment* follows seamlessly the *proof of origin*, in the case it was successful. The procedure consists of two steps, first the generation of the session key K_{ses} and second the distribution of it, as it can be seen in Figure 8.3 and in the general protocol below.

General protocol:

1. Generating a session key K_{ses}.
2. Distributing K_{ses} and acknowledging the receipt

Parameters:

Two parameters have to be set for this procedure. The first one is the connectivity, i.e. if the component is directly connected to the server or to an off-line proxy. The second parameter deals with the realization of the generation of the session key.

1. **On-line/ off-line connection**

The parameter defines which entity acts as verifier and thus generates and distributes the session key. The choice has no direct effect on the protocol flow.

2. **Generating the session key**

   Many different realizations may be found in literature [30] and elsewhere, whereby the following general cases can be distinguished:

   a) *Verifier generates the session key*

      The verifier generates the session key K_{ses} alone and distributes it to C_{new}. Since the verifier is usually capable to perform complex computations this is the common realization. The protocol flow could look like follows.

      1. Verifier generates the key
      2. C_{new}: $\longleftarrow$ Verifier: $E_{K_{new}}(K_{ses}, r)$

   b) *Component generates the session key*

      The new component generates the session key and distributes it securely to the verifier.

      In this variant the component has to be capable to compute a secure session key. Since the component has already proven its authorization to the verifier, it has to be ensured that the transmitter of the session key is equivalent to the authorized component. The next problem is, that the component cannot be sure to whom it is talking to, because the server never authenticated itself to components. Both problems are solved with the use of the component's secret key, since it has been assumed that K_{new} is only known by authorized verifiers and the component C_{new} itself. A possible protocol flow fragment for this scenario is presented in the following.

      1. Component generates the key
      2. C_{new}: $\longrightarrow$ Verifier: $E_{K_{new}}(K_{ses}, r)$

   c) *Verifier and component generate the session key together*

      Both, the verifier and the component add some information to the session key. The key is distributed by the server. A protocol flow of this variant is given below.

1. Component selects some key data and sends it to the Verifier
   C_{new}: $\longrightarrow$ Verifier: $E_{K_{new}}(keydata_{C_{new}}, r)$
2. Verifier selects some key data and generates session key
   $K_{ses} = f(keydata_{C_{new}}, keydata_{Verifier})$
   C_{new}: $\longleftarrow$ Verifier: $E_{K_{new}}(K_{ses}, f(r))$

Example:
An example for the protocol flow of the procedure in an scenario with server as verifier and the providing of clone detection is given below.

Protocol messages

1. Key generation K_{ses}
2. C_{new}: $\longleftarrow$ S: $E_{K_{new}}(K_{ses}, r_2)$
   C_{new}: $\longrightarrow$ S: $E_{K_{ses}}(f(r_2), ID_{Server})$

Protocol actions:

1. A session key K_{ses} is generated
2. The server sends K_{ses} and a challenge encrypted by the component's secret key to C_{new}. C_{new} then decrypts the message and returns the modified challenge and the sender's ID encrypted by the session key. The server checks by decrypting the received message if the component is in possession of K_{new} and K_{ses}. The data transfer can be started in the second message!

7.1.4 The Running System

In this solution a client and a server form a system for one single session only. Thus the *running system* describes the period of one session. Since the component is checked before a session starts, there is no need to re-check it during the same session. Thus the execution of additional procedures in the *running system* is redundant. The use of the freshly distributed session key ensures that both participants are authentic during an entire session, because K_{ses} is generated and especially distributed in a secure manner. During one session the entire data transfer is encrypted with the session key K_{ses} and is thus assumed to be confidential.

7.1.5 Disassembly of a Component

Short overview
Objective: Enabling authentic re-use in other systems
Purpose: Provide tamper-proofness
Executive entity: Verifier

This procedure is started for properly terminating the communication between the two participants at the end of a session. Furthermore, this procedure is necessary for updating data on the clients and the servers side if clone detection is provided. Both participants can start the procedure by sending a message asking for the termination, referred to as *goodbye* message in the following. All following protocol steps are depending on the choice of parameter 2, namely if the realization is providing clone detection or not. If not the *goodbye* message just has to be confirmed by the other participant, as can be seen in variant I. of step 2 in the below presented general protocol. In variant 2 II., the update of data of both participants has to be performed. The update of the data of both, the client and the verifier is a precondition for the detection of clones. If no data is altered an original component is not distinguishable from its clone. Alice and Malory could both use their original and counterfeit respectively at the same time and it could not be detected. If the data is refreshed on the client's site only, the verifier would not have a criterion to decide which of the two presented values (Malory's and Alice's) is the valid one. Thus Malory could still use its cloned component. Only if both sites are updating their data in the same manner, the verifier can check if the data is authentic or not. For clarity another example, Malory clones Alice's credit card, but this time a solution providing clone detection is used. The very next time *one* of the two credit cards is used, independent if the original or the cloned one, the two become distinguishable due two their differing data. The fraud one will be detected. Consequently, Malory could use his forged credit card one time at maximum.

General protocol:

1. Starting the procedure by sending a *goodbye* message.
2. I. Confirmation of the goodbye message
   II. Updating of key material or counter on the claimants and the verifiers side

Parameters:

As for the *assembly* two parameters have to be set for the disassembly. For the dependencies between both of them it is again referred to Figure 7.1.

1. **On-line/ off-line connection**

   Since the records of components which are to be disassembled have to be altered on the *ready for assembly-list* the parameter affects the protocol flow of this procedure.

   a) *Permanent on-line connection between the server and the system*

   Since the system is always on-line all records can be altered immediately when necessary.

   b) *Temporary on-line connection*

   Since the components are removed out of the system the data integrity of the current *ready for assembly-list* and the one hold by the HSM or another entity representing the server has not be regarded here. Possible discrepancies would have only effects of the assembly, when the component is built into a system again. But this case has already been considered in Section 7.1.3.1. Because of this a HSM or another entity representing the serving during the off-line period has to alter the particular record of the component on the *ready for assembly-list* without paying attention on data integrity.

   c) *off-line*

   For an off-line implementation the *disassembly* procedure is redundant, because the data on the list is static and could not be altered anyway. The components in this realization are just built into and removed out of the system without providing the ability to detect cloned components.

2. **Providing clone detection**

a) *solution providing clone detection*

To prevent the use of cloned components, the component's data has to be modified on the verifier *and* the component side. Therefore two realizations are presented below where more are conceivable. It has to be kept in mind that doublets and clones can be detected with this preventive measures, but the verifier cannot distinguish between the counterfeit and the original component. This is a weakness which could be helpful for an attacker. Malory makes a copy of a original component, let's say of Alice's credit card, with giving back the original one. If he makes sure to use his counterfeit before Alice is paying with her credit card, his card will not be detected as fraud. But the next time Alice will use her card again an alarm will be set since the data on her credit card and the assigned record on the server differ. At this time Alice has to prove that her credit card is the original one, the account has to be erased, and a new account has to be opened. Thus the presented measurements restricts the possible attacks strongly. Malory can only use one clone and he can only use it until the original component is used again. First is due to the fact that after using one clone the others become invalid. Latter restriction is for the case that the original was copied and then put back to the owner. Thus Malory should rather steal components than clone them, because he can only use one single component anyway and no direct theft protection is provided by the solution. Two variants for realizations providing clone detection are given below.

i. *Key update of the secret key*

Each secret key K_i can only be used once for the authentication otherwise the components could be cloned, i.e. the key is read out and stored in another component, and it would be impossible to distinct between the original and the counterfeit. Before the component C_{dis} is removed out of the system its secret key K_{dis} is updated by the verifier. The new key K_{new} is stored in the key memory of the component and on the *ready for assembly-list* hold by the verifier. In the component's record the new key only or the entire history of used keys might be stored. The new key could be generated randomly

or derived from the old one, e.g. by using a MAC with a secret. From latter case follows that each component with the same initial value would pass through the same states. The protocol flow for this scenario could look like the following.

1. Verifier or component sends a *goodbye* message to start the procedure
2. $C_{dis} \longleftarrow V : E_{K_{ses}}(keyupdate, K_{new}, r)$
   $C_{dis} \longrightarrow V : E_{K_{ses}}(f(r), ID_{Verifier})$

One participant starts the protocol by sending a *goodbye message* to the other. The verifier sends the command to start the *key update*, the new secret key and a challenge r encrypted by the session key to the component. The component stores the K_{new} by overwriting its old key memory and returns the challenge together with the verifiers ID again encrypted by K_{ses}.

ii. *Counter of authorized disassemblies*

Each time a component is a legally removed from a system a counter in the components memory and on the *ready for assembly-list* is incremented. Besides an incremental counter scenarios using decrementing counter or randomly chosen counter values are possible as well. This provides clone detection, since the counter reading would differ from the one hold by the verifier. This selection of the parameter has also effects on the procedure of the *assembly* since the counter reading would be a necessary part of the authentication message.

$C \longleftarrow V : E_{K_{ses}}(counterupdate, counterreading(t+1), r)$
$C \longrightarrow V : E_{K_{ses}}(counterupdate, counterreading(t+1), f(r), ID_V)$

b) *Solution without clone detection*

Even without providing clone detection the session between client and server should be terminated properly. Therefore one participant has to start the procedure by sending a *goodbye* message and the other one has to confirm this message. The protocol steps to be performed could look like follows:

$C \longleftarrow V : E_{K_{ses}}(goodbye, r)$
$C \longrightarrow V : E_{K_{ses}}(f(r))$

Example

An example of a protocol implementation for the chosen parameter 1a and 2(a)i follows.

Protocol messages

1. C_{dis}: $\longrightarrow$ S: $E_{K_{ses}}(disassembly)$
2. C_{dis}: $\longleftarrow$ S: $E_{K_{ses}}(update(K_{new}, r_3)$
3. C_{dis}: $\longrightarrow$ S: $E_{K_{ses}}(f(r_3), ID_{Server})$

Protocol action

1. The component wants to terminate the communication. Therefore it sends a *goodbye* message to the server.
2. The server generates a new secret key K_{new} for C_{dis} and sends it together with a challenge encrypted with the session key to C_{dis}.

   C_{dis} decrypts the received message, stores the new key in its key memory and sends a confirmation message consisting of the modified challenge and the servers ID back to the server.

7.2 Asymmetric Solution

Since piracy protection is the main purpose of the *proof of origin* it is provided by the asymmetric solution. But different from the symmetric implementation the asymmetric solution does not provide the detection of cloned part in its basic protocol. Due to the use of static certificates on the client side and the missing records of authorized potential components on the server side, the hold data cannot be changed as easy as in the symmetric implementation. But as mentioned before the altering of data is essential for the distinction between clones and original parts. Thus a protocol is presented in this section which provides piracy protection under the assumption that the components' data cannot be cloned. A realization of the asymmetric solution providing clone detection can be find as feature in Section 7.3.1.1. In the basic protocol the components authenticate themselves by their certificates and signed messages and the verifier holds a list of all revoked certificates, referred to as CRL in the following, and verifies the certificates and proves if they are recorded

on the CRL. Thus the verifier does not need to know a priori which components are authorized to get access.

7.2.1 The Participants

Only two participants take part in the protocol, namely the claimant and the verifier. The claimant is a component requesting for using provided services and the verifier is an entity holding the necessary data for checking the identity of the claimant. The verifier could be a server in an on-line scenario. In a temporary on-line realization a High Security Module (short HSM) or another entity could take over the role of the server, during the period the system is off-line. Their data is updated by the server every time the system goes on-line. In an off-line scenario the verifier could be again a HSM or another entity, whereby their data is never updated, i.e. the CRL is never updated in those systems. When talking about the server's representatives in the protocol we only mention the HSM, but this always includes all kinds of entities which could take over the role of the verifier.
The participants are:

- *New components*

  are components which claim for authorization to use a specific service provided by a server or another verifier. They are referred to as C or C_{new} in the protocol and hold the following data:

Component	Data
C_{new}	ID_{new}, $cert_{S_m}(P_m)$, $S_{b_{new}}$, $cert_m(P_{b_{new}})$

  ID_{new} notifies a unique identifier which is used to address the component and $cert_{S_m}(P_m)$ is used to verify signatures signed by the server or its representatives. The components private key for signing messages is $S_{b_{new}}$ and its associated public key is contained in the certificate $cert_m(P_{b_{new}})$. For securely communicating the component can sign messages with its private key and other parties can use the components public key for secure encryption.

- *The server*

  is connected to the system which provides the following data of the considered system[4]:

[4]The server also holds this kind of information for all other systems

Entity	Data
S	S_m, $cert_{S_m}(P_m)$, CRL

- *High Security Module*

  is a replacement for the server. The HSM is a permanent part of the system, but not always connected to the server. Thus it holds the server's data from the last time t_x it was connected to the server. The HSM holds the following data:

Entity	Data
HSM	S_m, $cert_{S_m}(P_m)$, $CRL(t_x)$ with $t_x \leq t$

  The provided data consists of an "old" ($t = t_x$) subset[5] of the server's data. Either the HSM has to be connected to the server any time a table look-up is performed or it is assumed that the data on the HSM is still equivalent to the current data provided by the server. In first case the HSM would be redundant because the server could directly perform all look-ups and other necessary steps for the authentication. Thus it has to be assumed that the data hold by the HSM still equals the current data hold by the server. This is a strong assumption and reduces the security of the solution.

7.2.2 Getting Started

In this solution there is no actual system. Thus nothing has to be set up before the protocol is executed, except of the component's initialization during their manufactory.

7.2.3 Assembly of a Component

As already been mentioned in the respective Section 7.1.3 of the symmetric solution the assembly of a component is rather to get the component connected to verifier as the assembly into a system. This process could be the insertion of a credit card into the slot of a automated teller machine or of a disc into a paddle. The protocol of the *assembly* consists of the procedure *proof of origin* where the claimant's authenticity is checked and may be an additional procedure for generating a session

[5] only the data of the system that the HSM is a part of

key, namely the procedure of the *key establishment*. Latter is necessary if the secure communication between client and server during the current session should be provided.

7.2.3.1 Proof of Origin

Short overview
Objective: To verify if a component is from an authorized trademark or organization
Purpose: To avoid the use of confidential services by bogus parts
Executive entity: verifier

As the name of the procedure implies the claimant is proven for originality. After executing this procedure the verifier knows if the component is authorized to be a part of the system or not. Authorized are only components from specific trademarks or organizations. The component has to prove its identity to the server and must also fulfill the required criteria. The protocol flow of the procedure is very similar to the one of the respective procedure of the symmetric implementation, which can be seen by comparison of the Figures 7.2 and 7.4 of the symmetric and asymmetric solution respectively. The verifier checks the identity by using challenge-response and performing a table look-up on its CRL. Only if the authentication is successful and *no* matching record is found on the CRL the protocol stops successfully or continues with the establishment of a session key. If one or both of the checks fail an alarm is set by the verifier and other measurements might be taken.

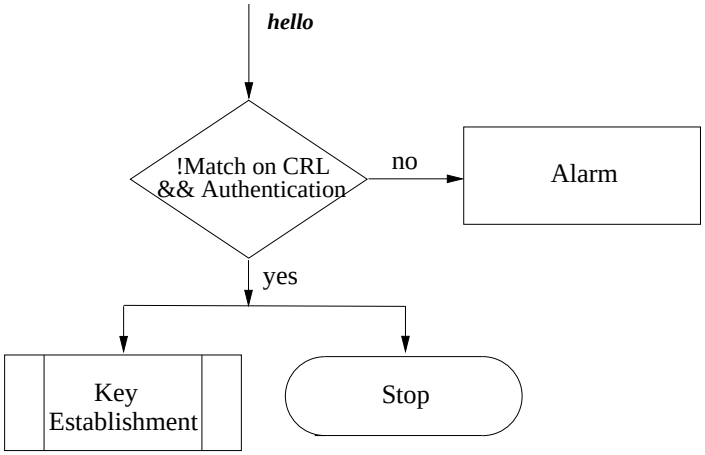


Figure 7.4: "Proof of origin", asymmetric solution: Protocol flow *proof of origin*

General protocol

1. New assembled component says *hello* to the verifier.
2. Verifier checks if the component's certificate is on the *CRL*:
   i. if there is a **match**:
   The protocol stops and further measures are taken.
   ii. if there is **no match**:
   → Claimant authenticates vs. verifier by using challenge-response
   α) authentication accepted:
   → The protocol either stops and the authentication is accepted **or** continues with the *establishment of a session key*; see respective sub chapter
   β) authentication rejected:
   → The protocol stops and further measures are taken.

Assumptions:
The assumptions which have to be made are the same one as of the symmetric implementation in Section 7.1.3.1, plus the additional assumption given below:

2. No clones of original components exist, since in this basic solution no clone detection is provided.

Parameters:
As in the symmetric implementation of the *proof of origin* the connectivity of the system has to be chosen. One additional parameter occurs due to the use of asymmetric encryption. The challenge can either be encrypted by the claimants public key P_{claim} or the claimant has to sign a challenge.

1. **On-/off-line connection**

   It is referred to read about this parameter in Section 7.1.3.1. Therefore the *ready for assembly list* and the matching records have to be replaced by *CRL* and mismatching records respectively.

2. **Challenge-response variants**

The two common variants of proving someone's identity by using challenge-response protocols are presented below. An additional variant combining the previous ones for ensuring encrypted communication in all protocol steps is given as well.

a) *Proof by decrypting the challenge*

The Verifier V challenges the claimant C by using the claimants public key P_C, obtained from its certificate, for encrypting a random number r. The claimant then proves its possession of the secret key S_C by returning the challenge in plaintext.

C: $\longleftarrow$ V: $P_C(r)$

C: $\longleftarrow$ V: r

b) *Proof by signing the challenge*

In this variant the claimant proves its knowledge of S_C directly by signing a challenge. The verifier sends a random challenge r and the claimant returns it signed by its secret key. The verifier then verifies the signature by the use of the certified public key of the claimant.

C: $\longleftarrow$ V: r

C: $\longleftarrow$ V: $S_C(f(r))$

c) *Encrypted challenge-response*

This variant ensures that all protocol steps are encrypted and thus secure and confidential. It is a combination of both previous variants.

C: $\longleftarrow$ S: $P_C(r)$

C: $\longrightarrow$ S: $S_C(f(r))$

Example:

For better comparison a realization with server is considered as in the respective example of the symmetric procedure. A fully encrypted challenge-response protocol (see parameter 2c is used for the authentication.

Protocol messages

1. C_{new}: $\longrightarrow$ S: $hello, cert_{S_m}(P_{new}, ID_{new})$
2. C_{new}: $\longleftarrow$ S: $E_{P_{new}}(r_1), ID_{Server}$

   C_{new}: $\longrightarrow$ S: $E_{S_{new}}(f(r_1), ID_{Server})$

Protocol actions

1. The component sends a *hello* message together with its certificate to the server to start the procedure.

2. First the server verifies the certificate by using P_m. If $ver(cert_{S_m}(P_{new}) = true$, the server performs a table look-up for matching records on the *CRL*. If the server cannot find a match the protocol continues, otherwise an alarm is set.

   Now C_{new} has to authenticate itself to the server. For this purpose the server encrypts a challenge with the public key of the component, which it obtained of the certificate. The result and the server's ID is send to C_{new}.

   C_{new} decrypts the received message and signs the challenge together with senders ID and returns the result.

   The server verifies the signature and in the successful case the protocol either stops at this point and is re-started for the procedure of the *disassembly* or it continues with the procedure of the *key establishment*.

7.2.3.2 Key Establishment

Short overview
Objective: Distributing of a session key K_{ses}
Purpose: Secure communication between component and verifier during one session
Executive entity: a verifier

The same reason for executing this procedure counts here than in the symmetric solution. Only the transmission of the session key differs due to the use of other secret keys for ensuring confidentiality. It is referred to read the Section 7.1.3.2 of the symmetric solution and see the Figure 8.3 for further information.

General protocol:

1. Generating a session key K_{ses}.
2. Distributing K_{ses} and acknowledging the receipt.

Parameters:
The same parameters as in the respective procedure of the symmetric solution in Section 7.1.3.2 have to be set. One additional parameter occurs due to the usage of asymmetric encryption.

3. **Encryption of the key transmission**

   The use of asymmetric encryption enables two variants for the encryption of the session key K_{ses}.

   a) *using* S_m

   The verifier signs the session key by the manufacturer's private key.

   C: $\longleftarrow$ V: $S_m(K_{ses})$

   b) *using* P_C

   The verifier encrypts the session key with the claimant's public key.

   C: $\longleftarrow$ V: $P_C(K_{ses})$

Example:
An example is given for a scenario with server and securely distributing the session key by signing it.

Protocol messages

1. generating session key K_{ses}
2. C_{new}: $\longleftarrow$ S: $S_m(K_{ses}, r_2)$

   C_{new}: $\longrightarrow$ S: $E_{K_{ses}}(f(r_2), ID_{Server})$

Protocol actions:

1. K_{ses} is generated

2. The server signs K_{ses} together with a challenge and sends it to C_{new}.

   C_{new} confirms the receipt by encrypting the challenge and the sender's ID with K_{ses}.

7.2.4 The Running System

Once a session key K_{ses} is established between both participants, this key is used during the entire session. Since this key is a secret key all realizations are using symmetric encryption schemes and there is no difference between the symmetric and the asymmetric solution anymore until the communication is to be terminated and the procedure of *disassembly* started. It is referred to read Section 7.1.4 which deals with the running system in the symmetric implementation.

7.2.5 Disassembly of a Component

Since no clone detection is involved in the basic implementation of the asymmetric *proof of origin*, no data has to be updated neither on the claimant nor on the verifier side. Because of that a simple connection termination is performed. Therefore one of the two participants sends a *goodbye* message to the other. The communication partner confirms this message. This simple dialog can be seen in the protocol fragment given below. It is independent of parameters and no further assumptions have to be made. As soon as the detection of copied components should be provided additional protocol steps have to be performed. One realization providing clone detection can be seen in the next Section 7.3.1.1.

General protocol:

1. Starting the communication termination by sending a *goodbye* message.
2. Confirmation

A connection termination could look like follows:

1. $C :\longrightarrow V : E_{K_{ses}}(goodbye, r_3)$
2. $C :\longrightarrow V : E_{K_{ses}}(f(r_3))$

7.3 Features

This section deals with possible features for the protocol of the *proof of origin*. The realizations are considered for both solutions, the symmetric and the asymmetric one.

7.3.1 Clone Detection

This feature is necessary to provide protection against frauds with clones. Clones are malicious components holding a copy of all data, i.e. inclusive the secret data, of an authorized component. Thus they are not distinguishable from the original parts. If the hold secret data can be secured in a manner that it cannot be readout and copied, the detection of clones becomes redundant. According to [5] providing this feature could be harder than it sounds like. One possible solution to provide the ability of clone detection is to alter some data every time a component is used. The data is altered on the component and on the verifier side to re-establish the uniqueness of the data again. The problem left is that the component which is still usable after this data modification should be the original component and not its clone. This problem cannot be solved. B we can set an alarm and take further measures if one doublet is detected, independent if this is the original component or the cloned component. In the case of a credit card this would lead to the following scenario. Malory copies Alice's credit card and gives it back to her. Now Malory goes shopping and pays with his forged credit card using Alice's account. Nobody will detect the fraud until Alice uses her card again. Her card will be rejected, she can prove that her card is the original one by presenting extra information to her bank such as her passport to identify herself. At this point measure can be taken, as for instance, blocking her bank account and open a new one. By then Malory's copy becomes useless. How to realize clone detection in the protocol of the *proof of origin* has been already presented for the symmetric implementation in Sections 7.1.3 and 7.1.5. The protocol flow of a scenario providing clone detection by the use of counter can be seen in Figure 7.5 and can be used for direct comparison with the respective asymmetric scenario, which protocol flow is given in Figure 7.6.

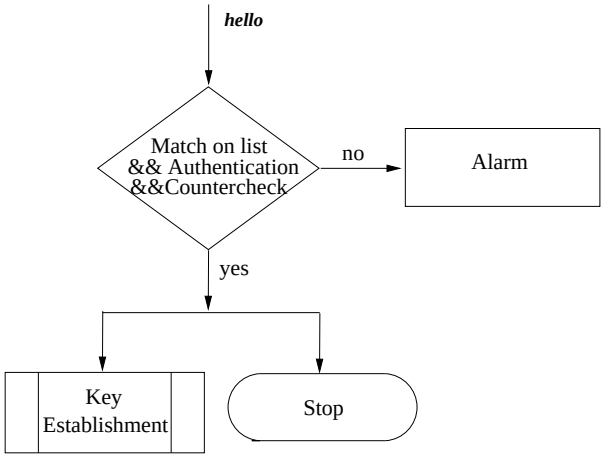


Figure 7.5: “Proof of origin”, symmetric solution: Protocol flow *proof of origin* providing clone detection

7.3.1.1 Asymmetric Solution

Short overview
Objective: Enabling clone detection **Purpose**: Provide fraud resistance **Executive entity**: A verifier

For providing clone detection in an asymmetric implementation of the *proof of origin* some additional expenses are necessary. The usual advantage of asymmetric encryption schemes that the verifier does not need to hold any information about the client a priori turns out to be a disadvantage in the implementation of this feature. As it has already been mentioned before, the altering of data on the client and the verifier side is required. Since certified information cannot be altered without issuing a new certificate each time, some extra information has to be stored independently of the certificate on both sides. The variant with a key update to enable clone detection as described in parameter 2(a)i of the symmetric solution is not feasible here, since the asymmetric keys are certified and an additional symmetric key would make the entire asymmetric system redundant. Thus variants using counters, as described in parameter 2(a)ii of the symmetric solution, are considered here. The additional information are stored in extra memory of the client side and on an extra list on the verifier side, namely the *used list*. The used list contains the records of all ever used components, i.e. all parts which ever authenticated themselves to the verifier.

The records contain, among other information as the component's ID, the current counter reading of the components. The verifier still does not need to know all potential claimants a priori. When no matching record can be found on the list, it is assumed that the requesting component is assembled into the system for the very first time. The procedure of the *assembly* and of the *disassembly* have to be customized for providing the clone detection. During the execution of the first procedure the counter reading of the requesting component has to be checked as can be seen in Figure 7.6 and in the below presented general protocol. The procedure of the *disassembly* has to deal with the updating of the counter every time a session is terminated and is described in the respective part of the general protocol.

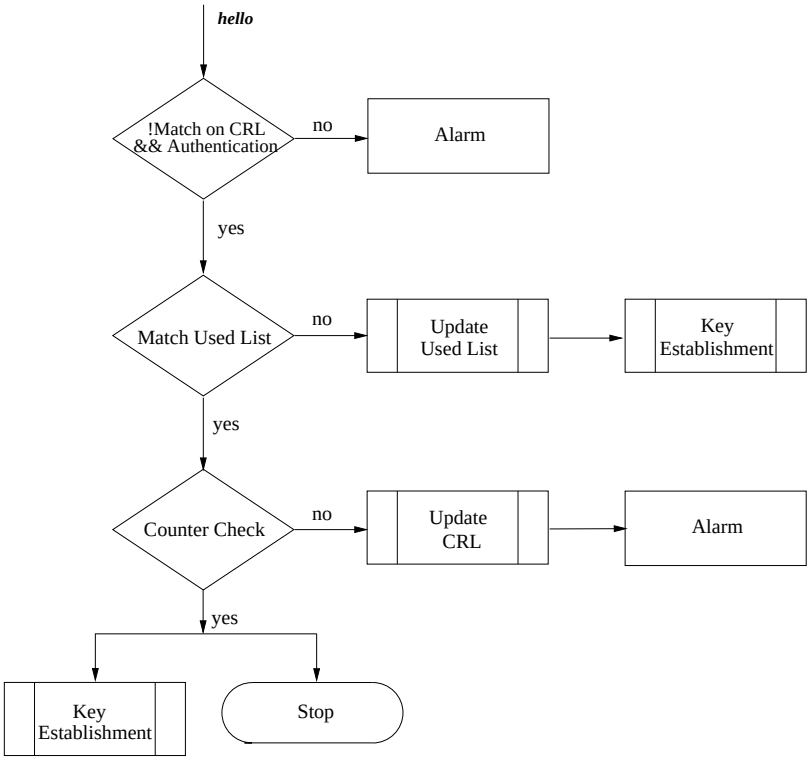


Figure 7.6: "Proof of origin", asymmetric solution: Protocol flow *proof of origin* providing clone detection

General protocol:
Assembly

1. New assembled component says *hello* to the verifier.
2. Verifier checks if the component's record is on the *CRL*:
   i. if there is **no match**:
   Authentication new component vs. verifier by challenge-response
   α) authentication accepted:
   → The protocol continues
   β) authentication rejected:
   → The protocol stops and further measures are taken.
   ii. if there is a **match**:
   → The protocol stops and further measures are taken.
3. Verifier checks if the component's record is on the *used list*:
   i. if there is **no match**:
   → Update of the *used list*,
   protocol continues with the *key establishment* or stops successfully.
   ii. if there is a **match**:
   → Check of the claimant's counter reading
   α) Check fails:
   → Update of the *CRL*, the protocol stops and further measures are taken.
   β) Check successful:
   → Protocol continues with the *key establishment* or stops successfully.

Disassembly

1. Starting the procedure by sending a *goodbye* message.
2. Updating of counter on the claimant's and the verifier's side.

Assumption:
Additional to the assumptions given in 5.5, the following has to be assumed:

1. The *used list* contains the records of all components, which ever asked for their authentication. Hence components which are not recorded are used the first time and their counter readings does not need to be proven.

Parameter:
For implementing this feature, one parameter has to be set, namely the realization of the counter check. Since the needed protocol steps are given in the description of the parameter choices no additional example of a protocol flow is given for this feature.

1. **Variants for using counter readings to provide clone detection**

   a) *independent of the claimant's certificate*

   The counter reading is computed and stored independently of the certificate. The counter reading hold by the claimant is compared to the one stored on the *used list* hold by the verifier each time the procedure of *assembly* is started. During the *disassembly* the counter reading of both participants is altered in a pre-defined way, e.g. by incremental it. This variant is equivalent to the presented one of the symmetric solution (parameter 2(a)ii).

   b) *as a part of the claimant's certificate*

   The component's certificate contains an initial value as bases for all following counter readings. One scenario could be the following:

   The initial value is a MAC over the component's ID and further component specific data $H_K(ID_C, \ldots)$. The MAC is part of the certificate $cert_m(P_C, H_K(ID_C, \ldots))$. The current counter reading is $H_K^{t_{cur}}(ID_C, \ldots)$ and hold by the component. The secret key K' required for computing the MAC values is hold by the verifier only. The records on the *used list* contain at least ID_C and t_{cur} of all components C_i. During the authentication the claimant sends its current MAC value and its certificate to the verifier. The verifier performs a table look-up on the ID and computes, by using K and $H_K(ID_C, \ldots)$ obtained from the certificate, the current counter reading $H_K^{t'_{cur}}(ID_C, \ldots)$. The verifier then compares its computed

result with the sent counter reading of the component, if both matches the authentication is accepted.

When one participants asks for the termination of the session, the verifier computes the new MAC value $H_K^{t_{cur+1}}(ID_C, \ldots))$. The verifier stores t_{cur+1} and sends the new MAC value as new counter reading to the component. The component stores the MAC as its new counter reading. The protocol flow of such an implementation could look like follows:

Assembly:

C: $\longrightarrow$ V: *hello*, $cert_m(P_C, H_K(ID_C, \ldots)), H_K^{t_{cur}}(ID_C, \ldots)$

Disassembly:

C: $\longrightarrow$ V: *goodbye*

C: $\longleftarrow$ V: $P_C(H_K^{t_{cur+1}}(ID_C, \ldots))$

C: $\longleftarrow$ V: $S_C(H_K^{t_{cur+1}}(ID_C, \ldots))$

7.4 Differences between Both Solutions

There are two main differences between the symmetric and the asymmetric solution of the *proof of origin*. The first one is about the data the verifier needs to hold and the second is the realization of the clone detection. In scenarios without clone detection the protocol flow of all procedures are mostly equal, which can be seen by comparison of the respective protocol flow charts.

Data hold by the verifier

As typical for asymmetric encryption schemes, the verifier of the asymmetric solution does not need to know all potential clients a priori, since certificates are used for the client's authentication. That might make this solution more suitable for systems with a great number of potential clients. Furthermore the extension of the potential system members is easier, since no data on the verifier side has to be changed for that. Due to the hold data the asymmetric solution seems also more suitable for temporary on-line realizations than symmetric implementations, since only the CRL has to be updated.

Providing clone detection

The providing of the clone detection is easier to implement in the symmetric solution, as can be seen by comparison of the protocol flows shown in Figure 7.5 and 7.6.

8 Protocol "Proof of Origin and System Check"

The protocol *proof of origin and system check* provides piracy, system and theft protection by combining both previous presented protocols. The piracy protection is warranted by the use of the protocol *proof of origin* and the system and theft protection by the use of the protocol *system check*. The basic ideas of the solution and the characteristics of suited systems, including some illustrated examples of application, were introduced in Section 4.3. During the execution of the *proof of origin* all newly assembled components have to authenticate themselves to a verifier. This authentication ensures that the components are original parts and not cloned components or counterfeits. After passing the test a component becomes a valid member of the system by initializing it with the system key. This ensures the theft protection since a component is assigned to a particular system and cannot be used in any other than its origin system. A stolen component would be noticed by its original system and the new system it is built into. The system integrity is provided by the *system check*. During the execution of the *system check* all unauthorized changes, such as unauthorized added, removed or exchanged components, will be noticed by the system. These recognized unauthorized actions cause an alarm and other sanctions might be taken. The entire protocol of the *proof of origin and system check* is introduced as symmetric solution 8.1 using symmetric encryption schemes only and as hybrid solution 8.2 using symmetric and asymmetric encryption schemes. We consider some features which could be additionally implemented in our protocol in Section8.3. The differences between the symmetric and hybrid solution are considered in detail at the end of this chapter in Section 8.4.

8.1 Symmetric Solution

We consider now a symmetric implementation for both parts of the protocol, namely the *proof of origin* and the *system check*. This is realized by using secret keys and symmetric encryption only. Figure 8.1 shows the typical life cycle of a component and the associated procedures of the symmetric solution. The life-cycle describes the entire period a component is a part of a particular system, i.e. from its assembly to its disassembly. During the *assembly* period two procedures, the *system key check* and the *key initialization* are executed. After passing this checks a newly assembled component becomes a valid member of the system. In the running system the *system check* is executed to prove the system integrity. Finally, when a system component should be disassembled the respective procedure *disassembly* has to be executed. During the execution of this procedure all associations between the component and the system are obliterated to authorize it for a possible re-use in another system.

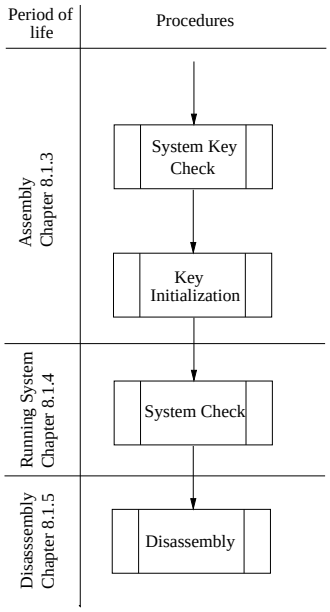


Figure 8.1: “Proof of origin and system check”, symmetric solution: Life cycle of a component

8.1.1 The Participants

First we introduce all participants which might act as claimant and/or as verifier in one or more of our procedures. Since these participants are already described in

general in Section 5.4, the main focus here is on the data they hold in this particular solution.
The possible participants are:

- *New components*

  are components which are to be assembled into the system and referred to as C_{new} in the protocol. They hold the following data:

Component	Data
C_{new}	ID_{new}, K_{new}

  ID_{new} notifies a unique identifier which is used to address the component. K_{new} is the component's secret key and used for secure communication between the verifier and the component.

- *System components*

  are components which are already part of the system and referred to as C_{old}, C_i or C_{sys} respectively in the protocol. They hold the following data:

Component	Data
C_i	K, ID_i, ID-List, K_i, $\forall i \in 1, \ldots, n$

  The identifier ID_i and the secret key K_i did not change after the component is built into the system (see new component), whereas K and the ID-list are received during the initialization process of the component. K is the secret key of the system and static (except of key updates), the ID-list consists of the IDs and perhaps more information like system status, priority etc. about all components which are currently in the system. The list is refreshed every time a system modification is recognized, e.g., during the system check or the assembly of a new component.

- *The server*

  referred to as S is permanently or temporarily connected to the system. The server provides the following data of a system[1]:

Entity	Data
S	*used list*$(ID_l, K_l, \ldots)$, *ready for assembly-list*$(ID_j, K_j, \ldots)$

[1] it also holds this kind of information for all other systems

The two consecutive indices l and j are used to number the components on both lists. Since the lists are permanently altered, i.e. records of components are added, altered, or removed, the number of components is not determined.

The *used list* holds the records of all currently used components (of all systems). The *ready for assembly-list* is a list of all components which are authorized for their assembly. The *used list* is necessary to secure the communication between the server and system components. The *ready for assembly-list* is used to secure the communication between the server and new components. It is also used to check the authorization of new components. Both lists contain at least the ID and the secret key of the components, which are the minimum required information needed to provide secure communication.

- *High Security Module (HSM)*

  is a replacement for the server in temporary on-line or off-line systems. In temporary on-line systems the HSM provides a copy of the server's data of the system the HSM is a part of. The data set is from the last time t_x, with $t_x \leq t$, the server was connected to the system and thus to the HSM. In off-line system the data provided by the HSM is static and the use of a *waiting list* is redundant. The HSM provides the following data to the system:

Entity	Data
HSM	*used list*$_{t_x}(ID_l, K_l)$, *ready for assembly-list*$_{t_x}(ID_j, K_j)$, *waiting list*

The provided data consists of an "old" (t_x) subset of the server's data and a current *waiting list*. Whereby subset denotes the data records of all components of the system that the HSM is currently a part of. The *waiting list* holds the data of all components which took part in a protocol action during the time the system was off-line. All these actions have to be regarded again the next time the system goes on-line. The *waiting list* is used to check the integrity of the data provided by the HSM and by the server. The current data hold by the server might have changed in a way that, for instance, a newly assembled component which properly passed the test, by the use of the old data of the HSM's list, would not pass the check anymore using the fresh list.

8.1.2 Getting Started

Before the first execution of our protocol the system has to be set up and prepared. All required data for executing the *proof of origin* is held by all components a priori since all components are provided with the necessary data after their manufacturing process is completed. Consequently, this data depends on the manufacture/ brand only and is totally independent of the system the component is built into later on. The required keys and data are described in the previous section and in Section 5.4. The preparation of the components differs for the *system check*. The *system check* is based on the secret K which is known by all components. This key cannot be set in the components during the manufacturing process because K is specific to a system. Besides the system key K more data is required for the successful execution of the *system check*, for example the ID-list of all system components. Both, K and the ID-list are set in the component during its assembly into the system. All other data also required for the check is set during the component's manufactory. The process of the initialization with the system key depends on whether the first assembly takes place in a secure or non secure environment (see Section 5.6.1 parameter 5.4 for more details).

1. *Secure environment*

   The system key K can be transmitted to all components in plaintext. The key K might be transmitted by a *master component* within the system or by an external entity such as a server or a smartcard.

2. *Non-secure environment*

   The system key K has to be transmitted encrypted to all system components. As described in the previous paragraph the key K might be distributed by an internal or external entity.

8.1.3 Assembly of a Component

Each time a component is build into a system the component is validated, i.e., authorized components receive the secret system key K and thus become a valid member of the system. The protocol for the assembly can be divided into the *system key check* and *key initialization* which are explained in detail in Section 8.1.3.1 and 8.1.3.3 respectively. First procedure is to check if the new component is

already a part of the system and only re-built into the system. If the component is proven to be already a member of the system the protocol stops. If the component's authentication was accepted and the component is new to the system the protocol continues with the component's initialization.

8.1.3.1 The System Key Check

Short overview
Objective: check if assembled component has already been a part of the system
Purpose: re-assembled component will not be checked any further
Executive entity: a random system component

The *system key check* proves if the newly assembled component has already been a part of the system before. If so, the protocol stops and no more checks are necessary. It is proven by then that this component has already passed the *proof of origin* before and thus is in possession of the system key. Assembled components holding the system key could be, for example, components which have been fixed and are re-assembled into the their system. Performing this check saves time because it avoids further expensive checks. To prove if a component belongs to the system, it has to know the system key K. The verifier has to hold K, thus the check can only be executed by a system component. Once a component is build into a system it sends a message to the system components or the server. This message initiates further protocol steps. A look-up on the *ready for assembly list* list is sufficient[2] for the authorization of a new assembled component. As a result a component with a valid entry on this list needs no further key check and the procedure of the key initialization starts as can be seen in Figure 8.2. If the newly assembled component is not on the list it could be a former system component and thus be authorized for the re-assembly into its system. This is checked by a challenge-response protocol with a randomly chosen system component acting as the verifier and the component to be assembled as the claimant. If the claimant responses in a right manner the component is proven to possess the system key. The protocol stops at this point. If the component responses in a wrong manner the component is neither authorized for its assembly nor a part of the system. Hence the verifier sets an alarm and takes further measures. The protocol flow of the key check is shown in Figure 8.2 and explained in general, i.e. independent of parameters, in the following.

[2]And the execution of challenge-response messages to prevent replay attacks is necessary

General protocol

1. New assembled component says **hello** to the verifier.
2. Verifier checks if the component's record is on the *ready for assembly-list*:
   i. if there is a **match**:
   Authentication of new component vs. verifier by challenge-response
   α) if authentication accepted:
   → The protocol continues with the *key initialization*; see respective sub chapter
   β) if authentication rejected:
   → The protocol stops and further measures are taken.
   ii. if there is **no match**:
   → The protocol continues.
3. A random system component for acting as verifier.
4. Authentication new component vs. system component by challenge-response
   i. if the new component knows K:
   - the verifier informs the server about the positive result of this check
   - the protocol (successfully) stops at this point.
   ii. if the new component does not know K:
   - the verifier informs the server about the negative result of this check
   - the protocol stops, an alarm is set and further measures are taken.

Parameter:

The parameters for the key check are:

1. **On-line/ off-line connection**

   This parameter affects all protocol steps where a look-up on the *ready for*

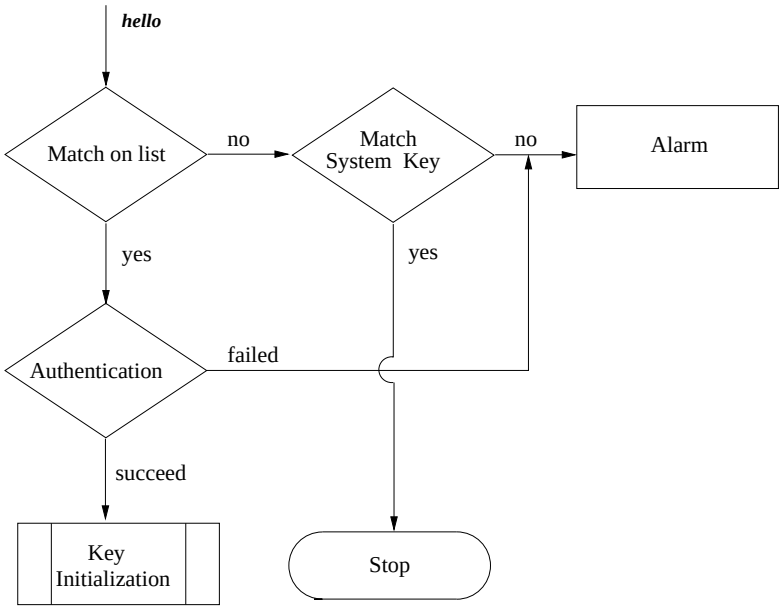


Figure 8.2: “Proof of origin and system check”, symmetric solution: Protocol flow *system key check*

assembly-list is needed. In this procedure a look-up for checking if the component is authorized for its assembly is executed right in the beginning of the protocol execution. Besides that the parameter affects the process of choosing the random system component which acts as the verifier in the further protocol steps.

a) *Permanent on-line connection between the server and the system*

Since the system is always on-line look-ups of data are not an issue here. Regarding choosing the random component we present a solution in the following. The server chooses a random component out of the system. The selected component executes the authentication of the new assembled component and notifies the server about the result.

b) *Temporary on-line connection*

i. *with HSM*

The HSM performs a look-up on its *ready for assembly-list*. As mentioned in Section 5.6.1 all cases of data integrity or non integrity as given in Table 5.1 have to be considered.

Case 1a) of the table would lead to step 4.i) or 4.ii) of the general

protocol, case 1b) to step 4.ii). Since step 4.i) means that the component has been a part of the system before these cases need not to be considered any further. In step 4.ii) the newly assembled component is rejected. The assembly could be performed again the next time the lists are updated.
Case 2a) of the table leads to step 2.i.α) or β), and case 2b) to step 2.i.α) of the general protocol. Since in step 2.i.α) the protocol stops, the process could be simple re-executed after updating the *ready for assembly-list*. Step 2.i.β) is continued by the key initialization. Since the initialization of an unauthorized component is the action to prevent it needs to be paid special attention on the last case.
Comprising it can be said that it needs to be paid special attention on step 2.i.β), since a malicious component could receive the system key. In one scenario the requesting component gets the system key, is then put on a *waiting list*, and gets re-checked the next time the list is updated. In another variant the component's request is put on the *waiting list* without receiving the system key. Note that the first scenario needs to implement further security measures in the case that a malicious part is detected during the re-check of the components on the *waiting list*. For example, an update of the system key within the entire system after removing the detected malicious component. In addition it has to be taken into account that after receiving the system key a malicious can only be detected when it is recorded on the *waiting list*. Otherwise the component cannot be distinguished between authorized and unauthorized components.
In all other cases the assembly of components could simply be re-tried after refreshing the lists and no further measures are necessary.
Since the key check is mostly executed by a system component the protocol flow differs only slightly from a implementation with server. The new assembled component registers at the HSM. The HSM then chooses the random system component as verifier. At the end of the protocol the verifier informs the HSM about the result of the key check.

ii. *without additional components*
Regarding the table look-ups on the *ready for assembly-list*, the same

counts as in the previous paragraph, i.e. all possible cases have to be considered separately and respective measures have to be taken for each case.

With respect to choosing the random component the protocol flow changes only slightly in this variant, for the same reason as mentioned above . The new assembled component registers at the randomly chosen (without any help of a server/HSM) system component. Either the result is transferred to another system component which continues with the brand validation or the same component continues in the protocol.

c) *off-line*

Since the lists are static in this implementations, no updates or *waiting lists* are necessary.

i. *with HSM as part of the system*
see b)i

ii. *without additional components*
see b)ii

2. **The system's data that the verifier holds**

The parameter affects the selection of a random system component, and also the communication between the server and the chosen system components.

a) *the verifier does not know which components belong to which system*

The server can only "communicate" with the entire system. The server has to choose a particular component out of the entire system, e.g., by sending a message to the entire system. All system components have to perform additional calculations to co-ordinate which component takes over the role of the verifier. Note that the server cannot encrypt its messages to the system but to single components.

b) *the verifier does know which components belong to which system*

The verifier directly chooses a random component of the system and communicates securely by using the secret component's key. This saves computations for choosing a random node and for confirming the result.

Example

For clarity an example for a particular case is given below. This is the detailed protocol flow for an implementation with an on-line server (parameter 1a) The server in this scenario holds a list of all system components arranged by systems (parameter 2b).

Protocol messages

1. C_{new}: $\longrightarrow$ S: $E_{K_{new}}(hello, ID_{new}), ID_{new}$
2. i. C_{new}: $\longleftarrow$ S: $E_{K_{new}}(r_{S_1})$
   C_{new}: $\longrightarrow$ S: $E_{K_{new}}(f(r_{S_1}), ID_{Server})$
   Start *key initialization*
   ii. Protocol continues
3. C_{old}: $\longleftarrow$ S: $E_{K_{old}}(r_{S_2}, keycheck, ID_{new})$
4. C_{new}: $\longleftarrow$ C_{old}: $E_K(r_1)$
   i. if C_{new} knows K:
   C_{new}: $\longrightarrow$ C_{old}: $E_K(f(r_1'), ID_{old})$
   C_{old}: $\longrightarrow$ S: $E_{K_{old}}(ID_{new}, systemcomponent, f(r_{S_2'}))$
   ii. if C_{new} does not know K:
   C_{new}: $\longrightarrow$ C_{old}: $Iamdumb()$
   C_{old}: $\longrightarrow$ S: $E_{K_{old}}(ID_{new}, !systemcomponent, f(r_{S_2'}))$

For better understanding of the protocol the steps are described in detail in the following.

Protocol actions

1. The new assembled component C_{new} sends a *hello* message together with its ID. The message is encrypted by the secret key K_{new}. Furthermore it sends ID_{new} as plaintext.

2. The server performs a table look-up on the *ready for assembly-list* to find an assigned secret key for the received ID_{new}.

i. If there is a match the server uses the assigned key to challenge the new component.

The claimant decrypts the messages, evaluates the received value in a defined way, encrypts it together with its ID and returns it.

The server verifies the message. If the authentication is successful the remaining steps of the current procedure are skipped and the protocol continues with the *key initialization*.

If the authentification is rejected, the protocol stops.

ii. If there is no matching record on the list the protocol continues.

3. The server chooses a random system component C_{old}, which acts as the verifier in the following. It then encrypts the ID of C_{new}, the command to start the key check and a challenge r_{S_2} with the secret key of C_{old} and sends it to C_{old}.

4. The system component C_{old} decrypts the received message and challenges the new component by sending a random number r_1. That number is encrypted by K.

i. if C_{new} knows K:

C_{new} encrypts the evaluated challenge r_1 together with the verifier's ID by K and sends the result back to the verifier.

The verifier checks the received values and checks to determine if ID_{new} is on the current ID-list of the system. If so, it sends the result of this check (component = system component) together with the ID of the new assembled component and the evaluated challenge $r_{S_2'}$ encrypted with its secret key K_{old} to the server. The assembly process is finished at this point and the protocol stops.

If the ID of the (re-)assembled component is not on the current *ID-list* the system component C_{old} sets an alarm and takes measures.

ii. if C_{new} does not know K:

it sends a *Iamdumb* message back to the verifier. The verifier sends the result of this check together with its ID, the ID of the newly assembled component and the modified challenge $r_{S_1'}$ encrypted with its own secret key K_{old} to the server. The server or the system component sets an alarm and takes further measures.

8.1.3.2 Proof of Origin

If the protocol is not stopped after the execution of the *system key check*, the component is proven to be authorized for its first assembly into the system. The authentication implies that the component is from an authorized trademark and thus authorized for the assembly. Since the component hold a record on the *ready for assembly-list* and the component's authentication was successful this check is not necessary. Consequently, no additional *proof of origin* is needed in the symmetric implementation to verify the origin of the component.

8.1.3.3 Key Initialization

Short overview
Objective: Initialization of all successfully authenticated components with the system key K
Purpose: Providing a key for secure communication within the system and for component authentication versus the system
Executive entity: A random system component

The procedure seamlessly follows the *system key check* in case that the new assembled component is proven to be authorized for the assembly and new to the system. This was proven having a valid record on the *ready for assembly-list*. Thus the component is authorized to obtain the system's secret key to become a valid member of the system. The secret key is used for secure communication between all components within the system. Furthermore, it is used for the *system check* which provides the authentication of components versus the system. Hence the knowledge of K is used as proof that components are part of the particular system. Since this secret key is only known by system components and the knowledge required for the execution of the *key initialization*, it is executed by one of them. After the verifier finds a record of the new assembled component on the *ready for assembly- list*, a random system component C_{old} is chosen for executing the *key initialization*. Since no key is shared between this system component and the new component a common secret key has to be generated and distributed before the actual key initialization can start. As soon as both parties hold this key C_{old} encrypts the system key K with it and sends it together with the encrypted system's ID-list. After the initialization all other system components have to be informed about the new system member

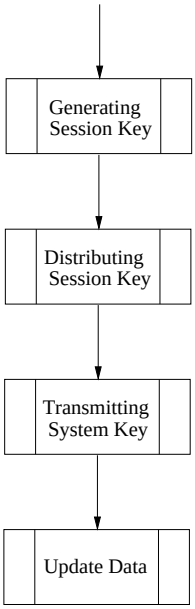


Figure 8.3: "Proof of origin and system check", symmetric solution: Protocol flow *key initialization*

by updating their ID-lists. Therefore C_{old} sends this update which consists of the ID, the status, and optionally any additional information of the new component, to all system components. This update must be guaranteed to be fresh, i.e., it is not replayed. The protocol flow of this procedure is shown in Figure 8.3. The single protocol steps are explained in the following, first in general and then for a particular choice of parameters.

General protocol:

1. A system component C_{old} is randomly chosen.
2. A session key is generated and distributed between the new component and C_{old}.
3. The system key K is encrypted with K_{ses} and send to the new component C_{new}. The ID-list is also send encrypted.

   $C_{old} :\longrightarrow C_{new} : E_{K_{ses}}(K), E_K(ID - list)$
4. The ID-list of all system components is updated in a secure (encrypted with the system key) and "fresh" manner.

Parameters:
The parameters for the *key initialization* are:

1. **On-line/ off-line connection**

   The parameter affects the process of choosing the random system component which acts as the verifier in the further protocol steps of the procedure.

2. **The system's data that the verifier holds**

   The choice of this parameter is important for the process of choosing a random component out of a system. Either a verifier holds data of all systems at once or it stores the data separately for each system.

   a) *The verifier does not know which components belong to which systems*

   To choose a random system component the verifier sends a message to the entire system. The system component needs to support the verifier by performing additional computations for selecting a random component among them.

   b) *The verifier does know which components belong to which systems*

   The verifier knows all single components which belong to a specific system, hence the server can directly choose a random component out of the current system.

3. **Communication between system component and "newly" assembled component**

   A shared secret between the participants for secret communication is needed. The server shares a secret with the new component (K_{new}) and each system component C_i (K_i). However, no secret is shared a priori between a system component and a new component. Thus no private communication between these nodes is possible although it is needed for the secure exchange of the system key during the key initialization. Either the server has to send the system key to the new component or an additional secret between a system component and the new one has to be generated. Either ways, one participant gets to know more information than it holds before the protocol execution. The server gets to know the system key K, or the system component gets to know the secret key of the new component. This is a drawback of the symmetric

solution. In both cases it is suggested to store the secret temporarily while executing these protocol steps. There is no reason to keep, since all members of the system can communicate securely by using the system key (system components with each other) or their secret keys (system component with server and vice versa).

a) *the server or HSM knows system key K*

If the server holds the system key the server could also execute the *system key check* and the initialization; no communication between "fresh" and system component would be necessary. Note that this variant is contradictory to the philosophy of the presented solution. The server should only be involved in the *proof of origin* but not in the inner system communication and the *system check* as described in item 6 of the design criteria 5.3.

b) C_{old} *knows secret key* K_{new}

The C_{old} knows the secret key K_{new} of the new assembled component C_{new} and can use this key to encrypt the system key.

Therefore the server has to send it encrypted to the system component which will execute the key initialization

C_{old}: $\longleftarrow$ S: $E_{K_{old}}(K_{new}, ID_{new})$

After the key initialization there is no need for using the session key for communication between system components because they all hold the system key K for this purpose. Therefore it is suggested that C_{old} just temporarily stores the secret key K_{new} while executing these protocol steps .

c) *Session key for the transmission is generated*

A session key K_{ses} for the key initialization is generated[3]. This key is generated by the server and distributed to C_{new} and C_{old} to exchange K. This is the recommended implementation.

4. **Generating the session key**

This parameter and parameter are interdependent. Many different solutions can be found in literature [30] and elsewhere. They include the following

[3]Independent of how the session key is generated, by the server only, or server and new component, or new and old component, the server always knows this key

variants:

a) *Server selects session key*

Verifier chooses the session key K_{ses} alone and distributes it to both participants. Therefore the verifier needs to be able to generate a key which meets the security demands. That follows that a implementation of this variant in combination with a server or HSM as verifier is recommended.

b) *C_{new} selects session key*

New component chooses the session key and the server forwards it securely to the system components.

c) *Information of all participants is used for the generation*

All participants add information to the session key which is distributed by the server.

5. **Update of the ID-list**

a) *executed by a system component (the verifier)*

Each system component challenges the random system component which was chosen to act as verifier C_{old}. The verifier returns the challenge combined with the new components ID_{new} and may be additional data encrypted with the system key.

In another variant C_{old} returns only one challenge instead of responding to each system component separately. Therefore C_{old} combines all responses, e.g., by multiplying them or by putting them on a list, and sends a multicast message to all system components. The claimants can check if their own challenge is part of the "total response". More examples of generating a "total response" are introduced in Section 6.4 parameter 5.

b) *executed by the server*

Each system component C_i sends the server a random value r_i as challenge and the server returns ID_{new} together with the challenge encrypted with the secret component key K_i.

C_i: $\longrightarrow$ S:r_i

C_i: $\longleftarrow$ S:$K_i(f(r_i), ID_{new})$

The function f() could be the product, or another function which depends on all inputs, or just a list of all inputs etc.

This parameter is depending on parameter 2. In scenarios with parameter 2a, i.e., the server does not know the single system components, the server can check if the IDs are valid but it needs to assume that every component of the system has challenged it. In variant 2b the server is able to proof if all system components have send a challenge.

Example

For clarity an example is given below. We assume that the server generates a session key for the transmission of the system key (parameter 4a). Furthermore a system component performs the task of updating the ID lists (parameter 5a).

Protocol messages

1. C_i: $\longleftarrow S : E_{K_{old}}(keyinit), ID_{old}, \forall i \in 1, \cdots, n$
2. generating a session key:

   C_{new}: $\longrightarrow S : E_K(r_{new})$
   C_{old}: $\longrightarrow S : E_K(r_{old})$

   distributing the session key:

   C_{old}: $\longleftarrow S : E_{K_{old}}(K_{Ses}, f(r_{old}), ID_{new})$
   C_{new}: $\longleftarrow S : E_{K_{new}}(K_{Ses}, f(r_{new}), ID_{old})$
3. C_{old}: $\longrightarrow C_{new} : E_{K_{ses}}(K, ID\text{-}list)$
4. C_{old}: $\longleftarrow C_i : E_K(r_i)\ \forall i \in 1, \cdots, n$
   C_{old}: $\longrightarrow C_i : E_K(\text{update(ID-list)}, f(r_i))\ \forall i \in 1, \cdots, n$

Protocol steps:

1. The server chooses one system component C_{old} and notifies all system components by sending a multi-cast message. This message contains the command to start the key initialization encrypted with the secret key of the chosen component and its ID in plaintext. Thus all system members know who will execute the key initialization, and they know whom to challenge for the ID-list update in step 4).

2. Both the new and the system component send a random number as a challenge to the server.

   The server generates a session key K_{ses} and sends it encrypted with the corresponding secret key to the new component and the system component. The message also includes the challenges and the ID of the communication partner in the key exchange.

3. C_{old} encrypts the system key K with the session key K_{ses} and sends it together with the encrypted ID-list. The new component decrypts the message and stores the obtained system key and the ID-list.

4. Each system component sends a challenge to C_{old}.

   C_{old} sends the ID-list update encrypted with the system key K to all components together with their challenges.

   Each system component decrypts the received message and updates its current ID-list by adding the new ID and all further information.

8.1.4 The Running System

Once a system consisting of original components is established, the system integrity needs to be ensured in the operating system. Therefore the procedure of the *system check* is executed. The procedure of the *system check* is identical to the procedure introduced in Section 6.4. Thus it is referred to read in this section everything about this procedure.

8.1.5 Disassembly of a Component

Short overview
Objective: Authorization of components for re-use in other systems
Purpose: Provide theft protection
Executive entity: The server and a system component

To provide the possibility of resell and re-assembly of components it is necessary to make a distinction between authorized and unauthorized (stolen) disassembled components. Because of this a procedure for the disassembly is needed. Since some component's records on lists have to be altered or erased the component cannot just be removed out of its system and re-built into another one. If a used component

passes all tests after the assembly in a new system it is proven to be authorized and thus not stolen. The presented procedure can be seen as a re-set of the component, but in some implementations the case between a brand new and a used component be distinguished. The difficulty of providing theft protection is that a stolen component holds a valid system key. Hence a server has to detect if a component which starts the disassembly process is still in its original system. Otherwise a stolen component could be build into another system and then legally start the procedure *disassembly* to get authorized for another assembly. The component is then authorized for a new assembly and could simply start the *assembly* procedure. To avoid this attack the component has to authenticate itself versus the server *and* the system during the procedure *disassembly*. It is authorized only when it passes both checks. In the case of a successful execution of the *disassembly* all system components are notified in order that they can adequately update their ID-list. But how can the server trust the system without knowing if this is really the original system of the requesting component. This problem is discussed in detail in parameter 2 and approached in the presented protocols.

When a component is to be disassembled, this component, referred to as C_{dis} in the following, sends a *goodbye* message to the server or the system. C_{dis} then has to authenticate itself versus the server to prove that it is a valid system component. Therefore the server performs a table-lookup of ID_{dis} and the secret key K_{dis} on the *used list*. In addition it has to be proven that C_{dis} is still part of its original system.

There are two variants for doing so. In the first variant, C_{dis} authenticates itself versus a random system component by proving that it holds the system key. For this variant the choice of parameter 2b) is necessary so it can be assumed that the system component acting as verifier is part of the original system. As a second variant the server challenges one or more random components and checks if they are all part of the same system as C_{dis}. The more system components are challenged the higher the security level.Therefore, all challenged components have to prove that they are in possession of the system key and a valid secret key. All claimants compute a value using the server's challenge and the system key as input and encrypting it with their secret keys. The value received by using a function over the two inputs might reveal some information about the secret key K, but this does not affect the security of the protocol since this data is encrypted and can only be decrypted by the server. Moreover, the server gets to know K or at least some information about it every time a new component is initialized.

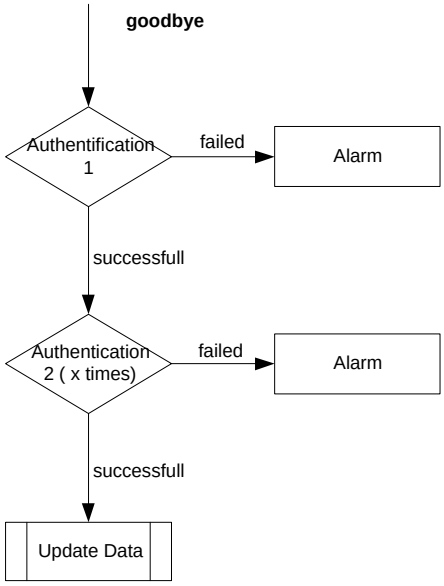


Figure 8.4: "Proof of origin and system check", symmetric solution: Protocol flow *disassembly*

After the successful execution of both authentications, the list held by the system, i.e. the ID-list, and the ones held by the server, namely the *ready for assembly* and the *used list*, have to be updated. The general protocol for the procedure of the disassembly is presented in the following. The two variants of the second authentication process can be seen in step 4 of the general protocol flow. The general protocol flow of the protocol is presented in Figure 8.4 and the single steps are explained in the following as well.

General protocol:

1. The component C_{dis} says **goodbye** to the server/system
2. One or more system components C_{sys} are randomly chosen
3. C_{dis} authenticates to server by challenge-response

   The server informs C_{sys} about the result of its check
4. i. Authentication C_{sys} vs. server by challenge-response

   or

   ii. Authentication C_{dis} vs. C_{sys} by challenge-response

   C_{sys} informs the server about the result of its check
5. Update of the lists:

   a) ID-list within the system

   b) *Ready for assembly- list* and *used list* of the server

Parameters:

1. **On-line/ off-line connection**

   The parameter is highly affecting the procedure since some data has to be altered, e.g., erasing the components record from the *used list* and creating a new record on the *ready for assembly- list*. A server-connection is essential for the disassembly of a component.

   a) *Permanent on-line connection*

   This is our recommended solution. The disassembled component can be re-assembled in a new system right away while data consistency of all lists is ensured.

   b) *Temporary on-line connection*

   As mentioned above data on the server-side has to be altered during this procedure. This cannot be performed until the system is connected to the server. The removed component can be legally re-used at the earliest after its "old" system is connected to the server. Then the removed component is approved to be removed out of its system in a authorized

manner. Thus an on-line connection to a server is recommended for this procedure. Otherwise it is hard to keep track for removed components when they are finally approved to an authorized assembly in another system.

i. *with a HSM (High Security Module) as part of the system*
The data which has to be altered on the server's database is temporarily stored in the HSM. The server updates its data the next time it is connected to the system.

ii. *with no additional components*
In this variant a system component temporarily stores the altered data.

c) *Off-line*

Since the *ready for assembly list* is static in the off-line solution, an authorized re-assembly in new systems cannot be supported. Thus the procedure of *disassembly* is impractical for this scenario. By using static list and providing the assembly of a component into different systems the detection of clones could not be supported. The list only holds information which components are authorized but not which components are currently used in a system. Nevertheless this variant is suited for applications where the assembly of components into other systems is not needed. For instance, vacuum cleaner bags would never be used in one system and then in another system.

2. **The system's data that the verifier holds**

As mentioned above, the server has to be able to distinct the original system of the component which wishes to be disassembled. The parameter affects the selection of the random system component.

a) *server does not know which components belong to which systems*

If the server only knows the ID and the key of a component it cannot map a component to a system. Hence the server cannot distinct between different systems. The server however, can detect if components are belonging to the same system. The server challenges n random components of a alleged system and assumes that if n component plus the component

which wishes to be disassembled belong to the same system the component has not been stolen and is still in its original system. The server has to select the random system components out of the entire system, since it cannot address single components within one particular system. Therefore it sends n messages to the entire system to select n random system components. An attacker would have to steal $n + 1$ components to pass this check. Unfortunately she could authorize all $n+1$ stolen components for legal re-use in any system. So the security of this solution increases by increasing n.

b) *server does know which components belong to which systems*

When a component asks for disassembly, the server can look-up its ID and knows immediately to which system the component belongs to. Furthermore, it can directly challenge another (randomly chosen) system component. If both components prove that they are from the same system, it is assumed that the claimant is still part of its dedicated system.

3. **Protocol steps executed in a serial or parallel manner**

If protocol steps are executed in parallel it has no direct affect on the protocol flow but could accelerate the execution of the procedure. Both authentications, for instance, could be performed parallel. In the case of challenging several system components to increase the security these authentications could be performed concurrently. For optimization as many operations as possible should be (in general) executed parallel. In doing so, it should be kept in mind that the update of all data should only happen if all authentications were successful and all entities which are responsible for updating data are informed about it.

4. **Update of the ID-list**

a) *executed by a system component (the verifier)*

The same method as introduced in the procedure *system check* for updating the ID-list can be used here.

b) *executed by the server*

Each system component C_i sends the server a random value r_i as a challenge and the server sends the ID of the component which should be disassembled C_{dis} with the modified challenge encrypted with the secret component key K_i.

C_i: $\longrightarrow$ S:r_i

C_i: $\longleftarrow$ S:$K_i(f(r_i), ID_{dis})$

If the server does not know the single system components (see parameter 2a)) it can only check if the IDs are valid and assume that all components of the considered system have challenged it. Otherwise the server can check if it received challenges from all nodes of the system.

Since the protocol is highly affected by the second parameter and the different possibilities for the realization of the authentication we give two examples below. The first one is an implementation with the chosen Parameter 1 a), 2 a), 4 a), and variant I for the authentication, the second is based on 1 a), 2 b), 4 b) and the second authentication variant.

Example 1

Protocol messages

1. C_{dis}: $\longrightarrow$ S: $E_{K_{dis}}(goodbye, ID_{dis}), ID_{dis}$
2. C_i: $\longleftarrow$ S: $x, r_{S1}, disassembly, ID_{dis}$ $\forall i \in \{1, \cdots, n\}$
3. C_{dis}: $\longleftarrow$ S: $E_{K_{dis}}(r_{S_1})$

   C_{dis}: $\longrightarrow$ S: $E_{K_{dis}}(f(r_{S_1}, K)), ID_{dis}$
4. C_{sys}: $\longrightarrow$ S: $E_{K_{sys}}(f(r_{S1}, K)), ID_{sys}$
5. if C_{dis} holds K_{dis} and K, the server informs C_{sys} about the successful authentication:

   C_{sys}: $\longrightarrow$ S: $E_{K_{sys}}(r_1)$

   C_{sys}: $\longleftarrow$ S: $E_{K_{sys}}(systemcomponent, ID_{dis}, f(r_1))$

   update ID-list:

   C_{sys} :$\longleftarrow$ $C_i : E_K(r_i)$, $\forall i \in \{1, \cdots, n\}$

   C_{sys} :$\longrightarrow$ $C_i : E_K(f(r_n), update(ID\text{-}list))$, with $r_n = f(r_i)$

Protocol steps:

1. The system component C_{dis} to be disassembled sends a *goodbye* with its ID

encrypted with its secret key together with its ID as plaintext to the server.

2. The server selects a random number x and sends it together with another random number as challenge r_{S_1}, the command to start the procedure, and the ID of C_{dis} to all components of the system.

   Each system component computes the representative of the system by, for example, XOR-ing all IDs of the ID-list with the received x. The ID which obtains the smallest result is assigned to the chosen component C_{sys}.

3. The server challenges C_{dis} with a random number encrypted with the component's secret key.

   The claimant decrypts the challenge, computes $f(r_{S_1}, K)$, and encrypts it with its secret key and returns it together with its ID as plaintext to the server.

4. The selected system component C_{sys} also computes $f(r_{S_1}, K)$ with the challenge received by the server and the secret system key as input. It encrypts the result using its secret key and sends with its ID in plaintext to the server.

5. The server decrypts the messages received from C_{dis} and C_{sys} and compares the first values. If they match, both components hold the same secret key. The server also checks if both components hold a valid secret component key which matches their IDs. If all checks are successful the affected lists can be modified.

   The server performs an update of its database by erasing the record of C_{dis} on its *used list* and putting it on the *ready for assembly- list*.

   For updating the ID-list of all system components, the server has to inform one system component[4]. Therefore this system component sends a challenge to the server and it responses with the modified challenge, the positive result of the authentication, and the ID of the component to be disassembled. The communication is encrypted with the secret component key.

   Now the component can inform all valid members of the system component about the modifications to be made on the ID-list. Again challenge-response (this time of all system components) is needed to guarantee freshness of the message.

[4]usually, since it is most convenient, it is the one which was chosen and authenticated before

Example 2

Protocol messages

1. C_{dis}: $\longrightarrow$ S: $E_{K_{dis}}(goodbye, ID_{dis}), ID_{dis}$
2. C_{sys}: $\longleftarrow$ S: $E_{K_{sys}}(r_{S_1}, disassembly, ID_{dis})$
3. C_{dis}: $\longleftarrow$ S: $E_{K_{dis}}(r_{S_2})$
   C_{dis}: $\longrightarrow$ S: $E_{K_{dis}}(f(r_{S_2}), ID_{server})$
4. C_{dis}: $\longleftarrow$ C_{sys}: $E_K(r_1)$
   C_{dis}: $\longrightarrow$ C_{sys}: $E_K(f(r_1), ID_{sys})$
   if C_{dis} holds K, C_{sys} informs the Server about the successful authentication:
   C_{sys}: $\longrightarrow$ S: $E_{K_{sys}}(f(r_{S_1}), ID_{dis})$
5. C_i :$\longrightarrow$ $Server : E_{K_i}(r_i)$, $\forall i \in 1, \cdots, n$
   C_i :$\longrightarrow$ $Server : E_{K_i}(f(r_n), update(ID_{dis}))$, with $r_n = f(r_i)$

Protocol steps:

1. The system component C_{dis} to be disassembled sends a general *goodbye* with its ID encrypted with its secret key together with its ID as plaintext to the server.

2. The server chooses a random system component C_{sys}, for executing the second authentication, and sends the command for starting the procedure, ID_{dis}, and a random value r_{S_1} as a challenge to it.

3. The server checks if the ID is on the list of valid system components (*used list*). If so, it looks-up the secret key of the component and challenges it by sending a randomly chosen number and encrypted with this key.

   The component responses with the modified challenge and its ID again encrypted with its own secret key.

   The server encrypts the received message and checks if the sent message is valid. If they are valid, the authentication versus the server is successful.

If so, it informs[5] the system component which acts as verifier in the second authentication about the positive result by sending the result plus the verifiers ID with the modified challenge.

4. The chosen system component C_{sys} checks if the ID is on the ID-list of the system. If so, it challenges C_{dis} with a random number encrypted with the system key.

   If C_{dis} is a member of the system it can decrypt the challenge. It then modifies the random number and encrypts it together with the verifiers ID, again with the system key, and sends the result back to the verifier.

   The verifier checks the received message and informs the server in the case of successful[6] authentication. Therefore, it encrypts the result, the ID of the claimant, and the modified challenge of the server (from the second step) with its secret key and sends it to the server.

5. When the server knows that *both* authentications were successful the update of all affected lists can be executed.

   a) Each system component sends a challenge r_i to the server for ensuring that the received update of the ID-list is fresh.

      The server sends the modified challenges with the update encrypted with the system key.

      The components decrypt the received message and update their current ID-list.

   b) The server updates its database in by deleting or altering the record of the component in the *used list* and putting it onto the *ready for assembly-list*.

8.2 Hybrid Solution

In this section the hybrid solution of the protocol *proof of origin and system check* is presented. The system check is always implemented by using symmetric encryption as already described in Chapter 6 as stand alone solution and in Section 8.1.4 as part of the symmetric solution. The *proof of origin* can be implemented for both

[5]For guaranteeing freshness of this message the recipient has to challenge the server before by sending a challenge

[6]otherwise an alarm is set and further measure are taken

kinds of encryption schemes and is here described in detail for the use of asymmetric encryption. In this chapter a combination of symmetric and asymmetric encryption schemes, i.e. a hybrid solution, is presented. In addition to a priori shared secrets, certificates for private and public key pairs are used, and consequently, an entire PKI is required. Due to the use of an asymmetric encryption scheme an entity has to hold a current list of all revoked certificates (CRL). The use of CRLs is generally known as one drawback of asymmetric encryption, but the advantage of it is that all participants can communicate securely without holding any secrets a priori. Hence no *ready for assembly -list* is needed as in the previous solution. In hybrid implementations a CRL and a *used list* for the detection of stolen parts is required.

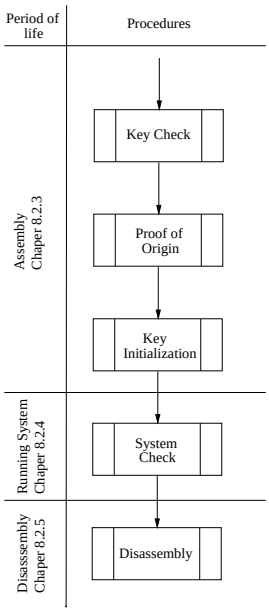


Figure 8.5: "Proof of origin and system check", asymmetric solution: Life cycle of a component

8.2.1 The Participants

The same participants take part in the protocol as in the symmetric solution presented in Section 8.1.1. The data they hold differs. In the following all participants with focus on their data are introduced.

- *New components*

  are referred to as C_{new} in the protocol and hold the following data:

Component	Data
C_{new}	ID_{new}, $cert_{S_m}(P_m)$, $S_{b_{new}}$, $cert_m(P_{b_{new}})$

  The $cert_{S_m}(P_m)$ is the manufacturer's certificate which contains the manufacturer's public key P_m. The certificate is "self issued by the m,manufacturer's private key S_m. New components also hold their own secret key $S_{b_{new}}$ and a certificate containing their public key $P_{b_{new}}$ signed by the manufacturer.

- *System components*

  are components in the running system and referred to as C_{old}, C_i or C_{sys}.

Component	Data
C_i	K, ID_i, ID-List, $cert_{S_m}(P_m), S_{b_i}, cert_m(P_{b_i})$

  In addition to the data hold by new components system components also hold the secret key K and the ID-list of the system.

- *The server*

  or short S in the protocol. The server provides the following data:

Entity	Data
S	S_m, $cert_{S_m}(P_m)$, CRL, *used list*

  The server knows the private and public key of the manufacturer. The server provides the *CRL* and the *used list* to the system. First list to detect components with revoked certificates and latter to detect cloned parts.

- *High Security Module (HSM)*

  is a permanent member of the system and holds the same data as the server from the last time the HSM was connected to the server. The HSM's data is updated every time the server is connected to the system. The following data are provided by the HSM:

Entity	Data
HSM	S_m, $cert_{S_m}(P_m)$, $CRL(t_x)$ with $t_x \leq t$, *used list*(t_x), *waiting list*

T_x denotes the last time t the server was connected to the system and the HSM could update its data.

8.2.2 Getting Started

As mentioned in Section 8.1.2 of the symmetric solution, no set up of the components is needed for the *proof of origin*, since they are set up with all required data during their manufactory. Thus the components need to be set up for the execution of the *system check* only. It is referred to Section 8.1.2 of the symmetric solution to read about these preparations, because this part of the protocol is realized with symmetric encryption in both solutions and thus equal in both solutions.

8.2.3 Assembly of a Component

Each time a part is assembled into the system, some procedures have to be executed to prove, if this component is authorized for the assembly or not. A component is authorized if it has been already a part of the current system, or it is new and of an approved brand. Used components which are re-setted and of an approved brand are authorized as well. If a component was a part of a system before can be proven by executing the *key check*. If a component is of an accepted brand and authorized for the assembly can be proven by the consecutive execution of the *key check* and the *proof of origin*. Components which are new to the system and pass those checks will receive the system key K in the procedure *key initialization*. All procedures of the assembly are described in detail in the following sections.

8.2.3.1 System Key Check

Short overview
Objective: To verify if a component has been already part of a particular system
Purpose: To avoid to check re-assembled parts again
Executive entity: A server and a system component

In the *system key check* it is verified if components were already approved parts of the system and thus still hold the system key K of the particular system. The procedure starts with a table look-up of the ID of the requesting component on the *used list*. If there is a match on the list the component is asked for its knowledge

of the system key. If the component can successfully prove this knowledge to the verifier it is ensured that this component was part of the system before.

Holding the system key means that a component was removed out of its system and is now re-assembled into the same system again, e.g. after fixing it. In this case the protocol stops and no further procedures have to be executed. If the component does not have a record on the *used list* the procedure of the *proof of origin* is started. Since the component is new to the system it needs to be checked is the component is authorized for the assembly and of a tolerated label. Performing the *system key check* avoids executing the entire protocol of the assembly for system components which are re-built into its system again. Thus components are only checked once if they are from a specific brand. The *proof of origin* is only performed once during the component's first assembly into a system and not again when the component is re-assembled into the same system. Hence, performing the *system key check* saves time and costs. The general steps of the protocol are explained in the following and are also presented in Figure 8.6.

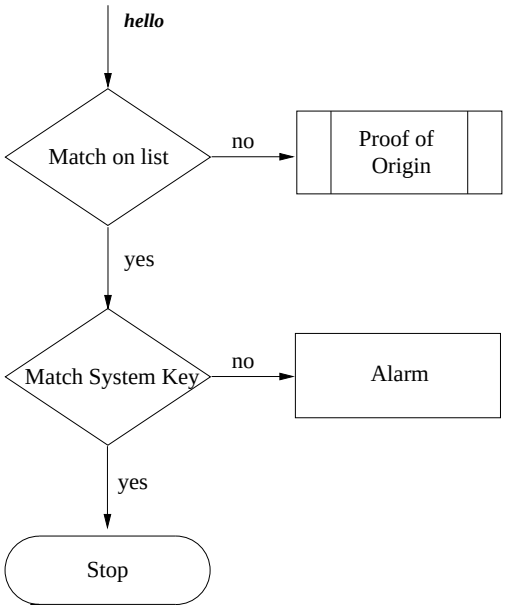


Figure 8.6: “Proof of origin and system check”, asymmetric solution: Protocol flow *key check*

General protocol

1. New assembled component says **hello** to the system/server.
2. Verifier checks if the component's record is on the *used list*:
   i. if there is a **match**:
   → The protocol continues
   ii. if there is **no match**:
   → The protocol continues with the *proof of origin*
3. A system component for acting as verifier is randomly chosen
4. Authentication new component vs. system component by challenge-response
   i. if the new component knows K:
   - the verifier informs the server about the positive result of this check
   → The protocol (successfully) stops at this point.
   ii. if the new component does not know K:
   - the verifier informs the server about the negative result of this check
   → The protocol stops, an alarm is set and further measures are taken.

Since the procedure is very similar to the symmetric solution presented in the previous section, similar assumptions have to be made and parameters have to be selected for the asymmetric implementation.

Assumption:
Only one additional assumption has to be made.

1. P_m is secret and hence only known by authorized parts

Parameters:

1. **On-line/ off-line connection**

[6] or a representative

The parameter affects all protocol steps where a look-up on the *used list* is needed. In this procedure a look-up for checking if the component is authorized for the assembly is executed right in the beginning of the protocol.

Besides this the parameter affects the procedure of choosing a random system component which acts as the verifier.

Since this has been already discussed before it is referred to Section 8.1.3.1 of the symmetric solution to see the details.

a) *Permanent on-line connection between the server and the system*

The server chooses a random component (see next parameter) and the selected component verifies the identification of the new assembled component. The system component notifies the server about the result of the authentication process.

b) *Temporary on-line connection*

The entity which represents the server performs the table look-up on the *used list* to check for any matches of the component's ID. As explained in 8.1.3.1 all cases for data integrity between the current *used list* hold by the server and the one hold by the representatives have to be considered.

Case 1a) and b) of Table 5.1 would lead to step 2.ii) in the general protocol, and thus end with the start of the *proof of origin*. Since the *proof of origin* can be passed holding a valid certificate and no record on the *used list*, both cases would lead to the key initialization. Hence, these actions have to be recorded by putting the components ID on a *waiting list*. This could be implemented in a way that the remaining protocol steps will be executed and the component is re-checked the next time the system's *used list* is updated by the server. Another implementation would be that the protocol stops in these cases and will be started again after the update of the *used list*.

Case 2a) would lead to 4.ii) and 2b) to 4i.) or 4ii.) of the general protocol of the *system key check*. Since the protocol stops in step 4i.) anyway, this case is not critical and the component could be just re-assembled and the protocol re-started again after a list's update. A protocol run which ends in step 4ii.) is not critical as well, since a component holding the system key has not to be regarded any further.

Concerning choosing a random system component without server access the procedure has to be executed without any external help, for details see Section 5.6.1 parameter 3.

c) *Off-line*

The *used list* is static in this implementations and thus there will not be any updates. Regarding the selection of a random component this procedure has to be performed again without any help of a server or another external entity and it is referred to Section 5.6.1 parameter 3 details.

2. **System data the server holds**

Either the server does or does not know which components belong to which particular system. This affects the selection of a random system component. It also affects the communication between server and system components. This parameter has been already discussed in Section 8.1.3.1 of the symmetric solution.

Note: In the case that the server knows all single members of each particular system the public key of the manufacturer P_m has not necessarily to be secret. In the fourth step of the protocol the notifying message could look as follows:

$$C_{old} \longrightarrow S : E_{S_{old}}(ID_{old}, ID_{new}, systemcomponent, f(r_{S'1})), cert_m(P_{b_{new}})$$

Since the server has chosen the component out of the considered system, it can verify the certificate and check if the ID matches the one of the chosen component. For decrypting the next protocol message the component's public key could be used.

For better comparison of the given examples the same parameters as in the symmetric implementation, presented in Section 8.1 are set here, namely an on-line server (1a), which holds the list of all system components sorted by systems (2b) under the assumption that P_m is secret.

Protocol messages

1. C_{new}: $\longrightarrow$ S: $hello, ID_{new}$
2. Server performs table look-up, in the case of a match the protocol continues
3. C_{old}: $\longleftarrow$ S: $r_{S_1}, keycheck, ID_{new}$
4. C_{new}: $\longleftarrow$ C_{old}: $E_K(r_1)$
   i. if C_{new} knows K:
   C_{new}: $\longrightarrow$ C_{old}: $E_K(f(r_1'), ID_{old})$
   C_{old}: $\longrightarrow$ S: $P_m(ID_{old}, ID_{new}, systemcomponent, f(r_{S_1'}))$
   ii. if C_{new} does not know K:
   C_{new}: $\longrightarrow$ C_{old}: $Iamdumb()$
   C_{old}: $\longrightarrow$ S: $P_m(ID_{old}, ID_{new}, !systemcomponent, f(r_{S_1'}), r_2)$

Protocol steps:

1. The new assembled component C_{new} sends a *hello* with its ID to the server.
2. The server performs a table look-up on the received ID. If no match can be found on the *used list* this procedure stops and the protocol continues with the next procedure *proof of origin*.

   If a match is found the protocol continues.
3. The server selects a random system component C_{old} for acting as verifier in the further protocol steps. The server sends then a random number r_{S_1}, the command for starting the key check, and the ID of the component which is to be assembled to the chosen verifier.
4. C_{old} sends a challenge r_1 encrypted by K to C_{new}.
   i. if C_{new} knows K:
   it encrypts the modified(f()) challenge r_1 together with the verifier's ID and sends the result back to C_{old}.

C_{old} checks the received challenge and if ID_{new} is on the current ID-list. If both checks are positive, it sends the result (component= system component) together with its ID, ID_{new} and the modified challenge r_{S_2} encrypted with P_m to the server.

The assembly process is (successfully) finished at this point and the protocol stops.

If the ID of the (re-)assembled component is not on the current ID-list the system component C_{old} sets an alarm and takes measures.

ii. if C_{new} does not know K:

it sends a *Iamdumb()* back to the verifier.

The verifier notifies the server about the negative result by sending a message which consists of its ID, ID_{new} and the modified challenge r_{S_2} encrypted with P_m.

The system component and/or the set an alarm and take further measure. The protocol stops.

8.2.3.2 Proof of Origin

Short overview
Objective: To verify if a component is from a authorized trademark
Purpose: To avoid the assembly of bogus parts
Executive entity: A server

The *proof of origin* verifies if components are from a licensed[7] trademark and authorized for their assembly into the system. For proving its authorization a component needs to be in possession of a certificate signed by the manufacturer of the system. In addition the certificate needs to be still valid and thus does not hold a record on the *CRL*. After passing these checks the new assembled component is authorized to receive the system key in the procedure of the key initialization to become a valid member of the system.

The *proof of origin* is based on an authentication of the new assembled component versus a verifier, which holds the required information to prove the validity of the used certificates. The Authentication is again implemented as challenge-response protocol whereby the authentication consists of several steps on the verifier side.

[7] by the manufacturer of the total system

The signature of the new component has to be proven to be “fresh” and to be verified successfully. Then table look-ups on both the CRL and the *used list* are performed by the verifier. After passing all these checks the component is authorized to receive the system's secret key. The verifier then informs the system, or more precisely one system component, about the positive result of the *proof of origin* of the new component. This step is the preparation for the procedure of the key initialization which is executed by a system component. For notifying one component the server has to choose a random system component which will start the next procedure.

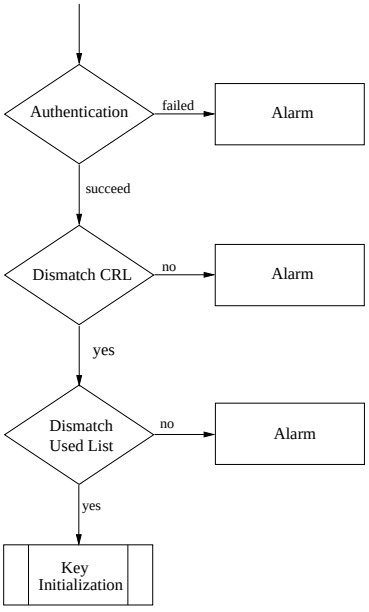


Figure 8.7: “Proof of origin and system check”, asymmetric solution: Protocol flow *proof of origin*

General protocol

1. Authentication new component vs. server
   a) Verification of the signature
      α) successful:
      → Protocol continues
      β) fails:
      → Alarm is set, further measures are taken; protocol stops
   b) Look-up on the CRL
      α) no match:
      → Protocol continues
      β) match:
      → Alarm is set, further measures are taken; protocol stops
   c) Look-up on the used list
      α) no match:
      → Protocol continues
      β) match:
      → Alarm is set, further measures are taken; protocol stops
2. Update of the used list, and if necessary of the CRL
3. Notification of the system about the successful proof of origin
   → Protocol continues with the *key initialization*

Parameters:

1. **On-line/off-line connection**
   a) *permanent on-line connection between server and system*
      All table look-ups can be performed immediately.
   b) *temporary on-line connection*
      i. *with HSM*
         When the system is not connected to the server during the brand validation, the HSM takes over the tasks of the server.

The HSM holds a version of the *used list* and the *CRL* which are updated every time the system is connected to the server.
The new assembled components are put on the *waiting list*. The next time the system is connected to the server all components on the *waiting list* will be checked again with by the use of a freshly updated data.
Either the protocol stops after putting components' records on the *waiting list* or it continues with the key initialization of the components. Latter means that a bogus part could possibly get the system key, but will be still detected after re-checking these components by an updated list.

ii. *without additional components*
When the system is not connected to the server during the brand validation, a system component takes over the tasks of the server.
The same counts for a system component acting as server replacement as for the HSM in the previous paragraph.

c) *off-line*
In an off-line implementation no *used list* and only static CRLs can be provided.

For better comparison with previous presented protocols and to show the recommended and most convenient solution the given example is a server implementation in which the server can distinguish between the single systems.

Protocol messages

1. C_{new}: $\longleftarrow$ S: r_S
   C_{new}: $\longrightarrow$ S: $cert_m(P_{new}), S_{new}(f(r'_S))$
2. update of the list(s)
3. C_{old}: $\longrightarrow$ S: r_1
   C_{old}: $\longleftarrow$ S: $S_m(f(r'_1), ID_{new}, cert_m(P_{new})$

Protocol steps:

1. The server sends a random number r_S as challenge to the new component

   The new component signs the received value r_S with its private key $S_{C_{new}}$ and sends the result together with its certified public key $cert_m(P_{new})$ back to the server

2. a) The server verifies the received signature.
   If $Ver(sig(r_S))$ =*true* the protocol continues.
   If $Ver(sig(r_S))$ =*false* it sets an alarm and takes measures.
   b) The server performs a table look-up for the certificate on the CRL.
   If there was *a match* on the CRL it sets an alarm and takes measures.
   If there was *no match* on the CRL the protocol continues.
   c) The server performs a table look-up for the certificate on the *used list*.
   If there was *a match* on the *used list* it sets an alarm and takes measures.
   If there was *no match* on the *used list* the protocol continues

3. The server performs an update of the *used list*, i.e. the certificate of the new component is added on the list.

   If necessary, in case 1.b), the CRL has to be updated by the server, in putting the certificate of the component which is to be assembled, on the CRL.

4. The server chooses a random component C_{old} out of the system. It then notifies C_{old} by sending a random number as challenge.

   C_{old} signs the modified challenge and a new challenge (chosen by its own) and sends it to the server.

   The server signs the modified challenge of the system component r_1 , the ID of the new component, the certificate of the new component, and its ID. The server returns the result to the system component C_{old}.

8.2.3.3 Key Initialization

Short overview

Objective: To initialize new assembled authorized components
Purpose: Secure communication among system components *and* proof of affiliation to a particular system
Executive entity: A system component

The *key initialization* follows seamlessly the *proof of origin* if the new assembled component passed all previous checks. It is the last procedure of the entire assembly process. After the key initialization the new component becomes a valid member of the system. Therefore the new component needs to securely receive the system's key and all member of the system have to be informed about the new member. Latter is done by updating the ID-list of all system components.
The system component which sends the system key to the new components has been already chosen in the previous procedure of the *proof of origin*. One particular system component C_{old} was notified by the verifier about the successful authorization of the new component. Since the notification contains the certificate of the new component, C_{old} can use the public key of C_{new} to send the system key K securely to it. Besides the system key the new components must also receive the current ID-list of the system. The ID-list needs to be updated. All ID-lists hold by each system component are updated by C_{new}. To guarantee the freshness of this list the use of challenge-response is again inevitable.

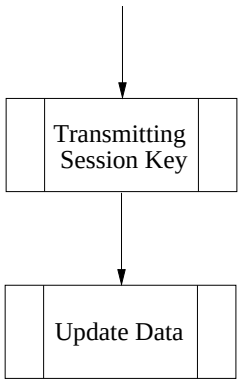


Figure 8.8: “Proof of origin and system check”, asymmetric solution: Protocol flow *key initialization*

General protocol

1. Sending the system key to the new component
2. Secure and fresh update of the ID-list of all system components.

Example
Since the protocol flow is not affected by parameters at all, this example represents a general scenario.

Protocol messages

1. C_{old}: $\longrightarrow C_{new} : P_{new}(K), E_K(ID\text{-}list)$
2. C_{old}: $\longleftarrow C_i : E_K(r_i)\ \forall i \in 1, \cdots, n$
   C_{old}: $\longrightarrow C_i : E_K(\text{update(ID-list)})\ \forall i \in 1, \cdots, n$

Protocol steps:

1. C_{old} encrypts the system key K with the public key of the new component and sends it together with the with K encrypted ID-list of the current system

   The new component decrypts the message and stores the obtained system key.

2. Each system component sends a challenge to C_{old}. This component returns the adjusted challenge together with the update of the ID-list.

   Each system component decrypts the received message and updates its current ID-list by adding the new ID and all further information.

8.2.4 Running System

Since all system components are communicating securely by using symmetric encryption with their secret system key the procedure of the *system check* does not differ from the respective protocol introduced in the symmetric solution. To read all details about this protocol it is referred to Section 8.1.4 of the symmetric solution.

8.2.5 Disassembly of a Component

The procedure of the *disassembly* has to be executed for the same reasons as the same procedure of the symmetric solution. It is referred to Section 8.1.5 to read more about it. All components which were a valid member of a system are holding a secret system key and are having a record on the *used list*. The component which wishes to be removed out of the system has to prove that it is still part of its dedicated system of which it holds the system key K of. This proof could be performed

by a trusted component of the considered system. Whereby trusted means, that we can assume that this component is really part of the system. If this assumptions cannot be guaranteed the entity which holds the *used list* has to perform this authentication with the help of one or more system components.

General protocol:

1. The component says **goodbye** to the server/system
2. One or more system component/s is/are randomly chosen
3. Authentication C_{dis} vs. verifier by challenge-response
4. i. Authentication C_{sys} vs. verifier by challenge-response
   or
   ii. Authentication C_{dis} vs. C_{sys} by challenge-response
   C_{sys} informs the server about the result of its check
5. If required the server informs C_{sys} about the result of its check
   Update of the lists:
   a) ID-list within the system
   b) *used list*

Parameters:

1. **On-/off-line connection**

   Important parameter of this procedure because some data has to be altered, e.g. erasing the component's record from the *used list*. Thus a server-connection is essential for the disassembly of a component.

   a) *Permanent on-line connection*

   Recommended solution, since the disassembled component could be re-assembled in a new system right away and data consistency could be ensured in this realization only.

   b) *Temporary on-line connection*

   As mentioned above, some data of the component has to be altered, this could not be updated until the system is connected to the server again. In

this case the disassembled component can first be re-used after connecting to the server and updating the particular data, which would be hard to trace.

i. *with a HSM*
The data which have to be altered on the server's database is temporarily stored in the HSM. The server updates its data the next time it is connected to the system.

ii. *with no additional components*
See previous paragraph with storing the data in one or more system components.

c) *Off-line*

Since the list for the off-line solution is static authorized re-assembly in new systems cannot be supported. Therefore the procedure of disassembly is impractical for this implementation.

Nevertheless this variant is suited for applications where assembly in other systems is not needed. For instance vacuum cleaner bags would never be used in one system and then in another system, or in general, in systems where no theft protection is required.

2. **Protocol executed in a serial or parallel fashion**

a) *serial fashion*

In this scenario all protocol steps are performed subsequently.

b) *parallel fashion*

First the server chooses a random system component C_{old}. While the server is verifying the identity of C_{old}, C_{old} is verifying the identity of the component to be disassembled at the same time.

3. **Update of the ID-list**

a) *executed by a system component* C_{old}

See presented method in the procedure of the *system check*.

b) *executed by the server*

Each system component C_i sends a random value r_i as challenge to the server. The server sends the ID of the component which should be dis-

assembled with the modified challenges encrypted with the secret key of the manufacturer

C_i: $\longrightarrow$ S:r_i

C_i: $\longleftarrow$ S:$S_m(f(r_i), ID_{old})$

whereby f() could be the product of the input or just a list of all inputs etc.

4. *Records of "used list*

a) *records of currently used components*

The list consists of certificates of all currently used components. For updating the list the certificate of the component which is to be disassembled has to be erased from the list.

b) *records of all used components*

The *used list* consists of all records of all components ever used. Therefore the components' records contain additional information such as data about the first assembly/ disassembly, second assembly/ disassembly etc..

During the update of the list the additional information has to be altered, e.g. by adding the new date of disassembly

Two examples of possible implementations of the procedure of the *disassembly* will be given below. The first scenario is a server solution in which the server challenges the component which is requesting its disassembly C_{dis} and a randomly chosen system component. The server verifies if both components belong to the same system. In the second scenario both, the server and a random system component, commonly verify C_{dis}.

Example 1

protocol messages:

1. C_{dis}: $\longrightarrow$ S: $goodbye, ID_{dis}$
2. C_i: $\longleftarrow$ S: $x, r_{S_1}, disassembly, ID_{dis}$
3. C_{dis}: $\longleftarrow$ S: $S_m(r_{S_2})$

   C_{dis}: $\longrightarrow$ S: $S_{dis}(f(r'_{S_2}, K'), ID_{server})$
4. C_{sys}: $\longleftarrow$ S: $S_m(r_{S_2})$

   C_{sys}: $\longrightarrow$ S: $S_{sys}(f(r''_{S_2}, K''), r_1, ID_{server})$
5. if both authentications were successful, the server notifies C_{sys}

   C_{sys}: $\longleftarrow$ S: $S_m(f(r'_1), ID_{dis})$

   update *used list* and ID-list of the current system

Protocol steps:

1. C_{dis} sends a *goodbye* message together with its ID to the server.
2. The server chooses two random numbers, one x to select a random system component and another one r_{S_1} as challenge. It then sends both numbers and the command for disassembly and ID_{dis} to all components in the system.

   Each system component does its computation to determine the system component C_{sys} which act as the verifier in the further protocol steps.
3. The server challenges C_{dis} with a signed random number to guarantee that the challenge is send by the server.

   C_{dis} responses with the signed challenge and the system key K as input, together with the server's ID.
4. C_{sys} also computes a value by using the same function f with the r_{S_2} and the system key K as input, together with a new challenge for the possible notification of the authentication, and the server's ID.
5. The server decrypts both responses and compares if $(f(r'_{S_2}, K') \equiv f(r''_{S_2}, K'')?$.

   If both results matches, the server performs a table look-up on the *used list* to check if ID_{dis} matches a valid record.

If all checks are positive the server notifies C_{dis} about the positive result by sending a signed and fresh message containing ID_{dis}.

Finally, the server updates the *used list* by erasing (or altering) C_{dis}'s record and C_{sys} updates the system's ID-list by erasing (or altering) ID_{dis} from the list. C_{sys} then notifies all system components by using challenge-response. For the detailed protocol steps it is referred to the first example of the symmetric solution in 8.1.5.

Example 2

Protocol messages:

1. C_{dis}: $\longrightarrow$ S: $goodbye, ID_{dis}$
2. C_i: $\longleftarrow$ S: $S_m(ID_{sys}, r_{S_1}, disassembly, ID_{dis})$
3. C_{dis}: $\longleftarrow$ S: $P_{dis}(r_{S_2})$

   C_{dis}: $\longrightarrow$ S: $S_{dis}(f(r'_{S_2}))$
4. C_{dis}: $\longleftarrow$ C_{sys}: $E_K(r_1)$

   C_{dis}: $\longrightarrow$ C_{sys}: $E_K(f(r'_1), ID_{sys})$

   if successful, C_{sys} notifies the server

   C_{sys}: $\longrightarrow$ S: $S_{sys}(f(r'_{S_1}), ID_{dis})$
5. update *used list* and ID-list

Protocol steps:

1. C_{dis} sends a *goodbye* message with its ID to the server.
2. The server chooses a system component C_{sys} out of the same system C_{dis} alleges to belong to. The server then signs ID_{sys}, a random number r_{S_1} as challenge, the command for starting the procedure of disassembly, and ID_{dis} to all components of the system.

   Each component decrypts the received message, and all know by then which component acts as verifier C_{sys} in the further protocol steps.
3. The server challenges C_{dis} by sending a random number encrypted with P_{dis}.

C_{dis} encrypts the message with its private key S_{dis} and uses the same key to sign the challenge and to return it.

4. C_{sys} sends a with K encrypted challenge r_1 to C_{dis}.

   C_{dis} responses with the challenge and the verifiers ID encrypted with K.

   C_{sys} proves the challenge and performs a look-up on the current ID-list. If both checks succeed, C_{sys} notifies the server about the successful authentification. For the detailed protocol steps it is referred to the second example of the symmetric solution in 8.1.5.

8.3 Features

We consider two features we can additionally implement in our *proof of origin and system check* protocol. The first presented feature provides the opportunity to distinguish between new and used parts. The second feature deals with the update of the system key K and is presented in Section 8.3.2.

8.3.1 New/Used Detection

The objective of this feature is to be able to distinguish between brand new components and used ones. The implementation of this feature could help to prevent the use of forged papers or certificates which are saying that the component is new.
Parameters:

1. **On-line/off-line connection**

   a) *Permanent on-line connection between the server and the system*

   There are two possible variants to provide the *new/used detection* and both can be implanted in symmetric and asymmetric solutions.

   The first variant requires the use of additional key material. These key material can only be used once. These keys are somewhat as initial keys. If a component hold this key it is proven to be new since all initial keys are erased after the first assembly of the component into a system. In the symmetric solution one symmetric key $K_{i_{new}}$ is necessary. In the asymmetric solution an additional key pair $cert_{S_m}(P_{Init}), S_{Init}$ would be required.

The second variant which is only suited for systems with permanent access to the server. In this realization the *used list* has to be modified in a way that all components ever used are on the list and all dates of assembly and disassembly are recorded. The protocol has not to be changed for this variant.

b) *Temporary on-line connection*

As mentioned in the previous paragraph only the first variant can be implemented in temporary on-line systems. In a asymmetric solution a system component could verify the certified initial key of a component. In a symmetric solution the access to the server would be necessary since only the server can distinguish between the initial symmetric key and the common symmetric keys.

c) *Off-line*

For the same reasons as mentioned in the previous paragraph only the asymmetric implementation of the first variant is possible.

One example of an implementation of the first presented variant using additional key material is presented below.

Example

Protocol messages:

1. C_{new}: $\longleftarrow$ V: r_{S_2}
2. a) C_{new} is brand new component:
   C_{new}: $\longrightarrow$ V: $cert_m(P_{new}, P_{Init}), S_{Init}(f(r_{S_2'}))$
   b) C_{new} is used component
   C_{new}: $\longrightarrow$ V: $cert_m(P_{new}, P_{Init}), S_{new}(f(r_{S_2'}))$
3. a) Verification of the certificate successful:
   i. C_{new} is brand new component
   C_{new}: $\longleftarrow$ V: $erase(S_{init})$
   set status(ID_{new})= new
   ii. C_{new} is used component
   set status(ID_{new})= used
   b) Verification of the certificate not successful:
   Protocol stops, alarm is set

Protocol steps:

1. The Verifier sends a random number r_{S_2} as challenge to the new component
2. a) The new component signs the received value r_{S_2} with its *initial* private key $S_{C_{Init}}$ and sends the result together with its certified public key $cert_m(P_{new}, P_{Init})$ back to the Verifier
   b) The new component signs the received value r_{S_2} with its private key S_{new} and sends the result together with its certified public key $cert_m(P_{new}, P_{Init})$ back to the verifier
3. a) i. The verifier sends a command to erase the initial secret key of the new component, the status of this component is set to "new" then
   ii. The status of the component is set to "used".
   The protocol might continue with further procedures.
   b) The protocol stops and an alarm is set.

8.3.2 Key Update

The objective of this feature is to update the system key K. To increase the system's security the system key K should be updated periodically and/or under certain circumstances. For instance if an unauthorized component is detected in the system, the system key should be updated after removing the fraud component out of the system. Another reason for updating the system key could be if a component is missing during the *system check*. Since the system key K is a secret key used for symmetric encryption, only symmetric encryption is used for the key's update. Since the *system check* is always executed by the system components only (i.e. without any help of a server or other external entities) the presented protocol for the key update does not require the help of any external entities either.

Assumption:

Since the new system key K_{t+1} is distributed by encryption with the old system key K_t, this update is only reasonable if all unauthorized components which (probably) know K are removed out of the system *before* the method is started. Otherwise the unauthorized component would also obtain the new system key K_{t+1} and the update would not gain any progress. Even in the worst case, i.e. that unauthorized components are still part of the system during the update, the unauthorized component would not hold more information than before.

Example

The protocol flow of a possible implementation of this feature is given below.

Protocol messages:

1. choosing a random system component C_j, $withj \in 1, \ldots, n$

   $C_{Comb} = C_j$

2. C_i: $\longrightarrow C_{Comb} : E_{K_t}(r_i, ID_i)$, $\forall i \in 1, \ldots, n \wedge \neq j$

3. C_i: $\longrightarrow C_{Comb} : E_{K_t}(K_{t+1})$

4. C_i: $\longrightarrow C_{all} : E_{K_{t+1}}(ID_i)$, $\forall i \in 1, \ldots, n$

Protocol steps:

1. A random system has to be chosen, called Combiner C_{Comb} in the following protocol

2. a) Each system component C_i sends a random number r_i together with its ID and encrypted with the "old" system key K_t to the Combiner

   b) The Combiner computes: $r = \prod_{i=1,i\neq j)}^{n} r_i$
   and then derives the new system key K_{t+1} from the result $K_{t+1} = f(r)$

3. The combiner encrypts the new system key with the old one and sends it to all system component. Each system component checks if its own challenge is part of the new system key by computing $r = f^{-1}(K_{t+1})$ and checking if they can divide the result by its own random number (challenge) r_i without remainder.

4. a) As acknowledgment and proof that this key-update included all system components each component sends its ID encrypted with the new system key to all system components

   b) Each component checks if it received all current IDs of the system
   If not the respective component sets an alarm and takes further measures.

8.4 Differences between the Both Solutions

There are differences in the particular procedures of the symmetric and hybrid solution, but also some general differences between the data the participants need to hold and thus between the communication among the participants. The differences between the procedures in the symmetric and asymmetric can be best seen by comparison the figures of the protocol flows of the single procedures. In the symmetric solution no procedure of *proof of origin* is necessary because this check is included in the *System Key Check* as can be seen in Figure 8.1 and 8.5. The procedures of *key initialization* differ as well, since no session key is necessary for the transmission of the system key K in the asymmetric solution, as can be seen in Figure 8.3 and 8.8.
The data hold by the entities differ in both solutions, as has been already discussed in Section 7.4 of the *proof of origin*. The main difference is that in the asymmetric solution all participant can communicate securely with each other. This is not possible for "old" system components and new assembled components in the symmetric solution which is solved by generating a session key for both parties. The main advantage of the asymmetric solution is that the server does not necessary hold any component's data, especially no secret one. Requesting components can

always send their certificate containing their public key. Thus the data hold by the server or other verifiers has to be write protected only and not read protected.

9 Enhancements and Improvements of the Protocols

After presenting protocols for providing theft, system and piracy protection respectively, some possible enhancements of the protocols will be discussed. Some of the features presented in the following are not suited for an implementation in present systems yet because of their high computational costs. As an example the use of zero knowledge identification protocols is still not suited for the use in constraint systems but might become attractive in the future due to the improvements of micro processors. The security policy of the *resurrecting duckling* and its similarities and differences to our protocols are briefly discussed in section 9.4. The chapter finishes with applications of key hierarchies and their possible integration in our protocols.

9.1 Zero Knowledge Identification

Due to the high computation cost, *zero knowledge authentication protocols* have not been considered in this thesis at all, although they provide the highest security level because the exchanged protocol messages do not reveal any secret information [30]. With the further development of micro processors and thus increasing computational speed the use of *zero knowledge authentication protocols* might even become suitable in constraint environments such as on smartcards. The protocols then may be replace the challenge-response authentication protocols used in our procedures.

9.2 Threshold Cryptography

Threshold cryptography is used for multiparty signing of messages as described in [29, 22, 17, 16, 21]. These cryptosystems have to be homomorphic such that most schemes are based on *RSA* and *El Gamal*. The advantage of threshold crypto schemes is that all components are only in possession of one piece of the secret. An

attacker would have to compromise k of n components to obtain the secret. In our case the system is represented by k randomly chosen components.
We can divide authentication protocols which use threshold schemes in the following categories:

1. 1 claimant - k verifier
2. k claimants - 1 verifier
3. k claimants - k verifier

Case 1 describes that a component authenticates itself to k components of the system. Thus the single component acts as claimant while the k components are acting as verifier. After the execution the system knows if the requesting component (claimant) belongs to the system or not. For example, during the assembly of a component the new component has to authenticate itself to the system. Instead of authenticating to a server the new component authenticates itself to the system which is represented by k system components.
Case 2 describes k components, representing the system, authenticating themselves to one single component. The single component acts as verifier and the k components as claimants. For example, the single component can check out if it belongs to the system.
Case 3 describes the scenario that all components including the new ones are randomly divided into two groups, whereby one group acts as verifier and the other as claimant. If the process fails at least one of the components responded in a wrong manner and is thus assumed to be malicious. The malicious component can be found by further checks.
Especially the last case seems to be of special interest since it could be used instead of our *system check*. However, there is the necessity of a trustworthy combiner for combining and comparing the results. The result of the authentication is verified and distributed to all other components by a combiner. The combiner needs to be trustworthy, and there has to be a secure channel to each component. Besides that all participant have to assume really talking to the combiner. For ensuring that, the combiner needs to authenticate itself versus all participants which take part in the authentication. Thus at this point it seems difficult to use threshold cryptography.

9.3 One-time Signatures

One-time signatures can be used to sign at most one message; otherwise, signatures can be forged [30]. A new public key is required for each message that is signed. When one-time signatures are combined with techniques for authenticating the public information, multiple signatures are possible. One-time signature schemes are very efficient, thus these schemes are useful in applications such as chip cards. For more information about one-time signatures it is referred to [30, 31, 49, 38], for discussion about the efficiency to [10] and for a suggestion of an on-line/off-line scheme to [20].

The use of one-time signatures could be helpful in the asymmetric solution of the *proof of origin* presented in Section 7.2 and also in the asymmetric combined solution *system check and proof of origin* presented in Section 8.2. In the first protocol the one-time signatures could be used as replacement for the key update. One possible scenario would be that all components hold a list of public keys to be used only once. The list is also hold by the verifier. This would ensure clone detection because the key is altered after each successful verification. Thus this scenario provides the same security feature with the advantage of fast execution due to the simpler signature generation.

In the combined solution the one-time signature scheme could be used instead of the common public and private key scheme. Either the component can only be legally assembled once or a list of public keys is necessary. This would, as in the previous suggestion, accelerate the execution of signing and verifying.

9.4 The Resurrecting Duckling

We could adopt some features of the *Resurrecting Duckling* security model developed in [41] and enhanced in [42] for the implementation of our solutions. There are many similarities but also differences between the our protocols and the one presented in the quoted papers, which we discuss in the following.

In our solutions all components receive a system key K during the first assembly of the system. If the environment this first assembly takes place can be assumed to be secure the key are transmitted in plaintext. In the *Resurrecting Duckling* security model the *secret holder* which initializes all components is referred to as *mother duck* and all components as *ducklings*. After the *imprinting* of the system

all components share a secret. One example could be the first assembly of an automobile. After the assembly of all parts the security engineer has to use his smartcard (*mother duck*) when turning the ignition key to start the initialization (imprinting phase). All components of the car (*ducklings*) receive the symmetric key. In our solutions the *mother duck* usually does not play an important role after the first initialization anymore. To adopt the concept of a *mother duck* could be helpful in some realizations. Especially scenarios in which only one instance should be able to obliterate associations between components (*to kill the duckling*) and to create new associations (*the re-birth of the duckling*) seem to be suited. This could be for instance in an application where the manufacturer of an entire system holds the total power over the system and no other instances are allowed to legally modify the system. For this purpose the manufacturer could hold a smartcard representing the *mother duck*.

Also more complex systems of more hierarchies could be realized by the use of the *Resurrecting Duckling* security model because the *mother duck* is able to delegate its authorizations to other entities. These other entities could be the smartcards hold by other instances such as the mechanics of a repair station or the owner of a car. This concept could be developed in more detail with the use of the *Resurrecting Duckling* security model, e.g., the possibility of the mother duck to delegate its authorizations to other entities and the implementation of more hierarchies and cross relations between the participants as described in [41, 42]. In summary the using of some concepts of the *Resurrecting Duckling* security model could enhance some of our solutions but it has to kept in mind that it would be only suited for applications in which the first assembly takes place in a secure environment.

9.5 Key Hierarchies

In some applications there will be many different groups of different interest and authorizations. For instance, in the implementation of the *Tachosmart* described in Section 2.2 are many different hierarchies of entities, e.g., the truck drivers, the truck itself and the mechanics. The different level of authorizations are realized by smartcards. Each active entity gets its own smartcard which enables it their work. This idea can be adapted to the presented protocols. For instance, for an automobile there are many different groups which need to be considered. There is the owner of the car which should be able to exchange and repair some parts on its own and

also to resell some. Thus he needs to be able to initialize some components and also to reset them for the purpose of resell. There is the group of car mechanics which could be subdivided in mechanics working in independent and authorized garages. The first group should be able to perform most repairs and at least to make the car roadworthy again in case that the car breaks down and no authorized garage is available. The car runs again but the new or exchanged parts are not initialized. The initialization has to be performed in an authorized garage later on. Another instance are the suppliers of the components, e.g. *Bosch*. Finally there is the manufacturer of the entire system, as for instance *Audi*. The manufacturer holds the highest priority and controls the suppliers by authorizing their products for assembly.
The different levels of authorization can be implemented by the use of different keys to enable different levels of rights. A preferable implementation uses smartcards for all entities. Key hierarchies provide methods to implement complex and interwoven relationships in large systems.

9.6 Subgroups of system components

We could divide all components into *subgroups* providing different levels of security. Each subgroup has its own system key. The higher the security level of a group the better the protection of the assigned key, for example ensured by the use of tamper resistant features. That follows if a system key is compromised only the key of one subgroup is compromised which increases the security of our solution. Each subgroup is performing their own *system check* and for communicating among different subgroups additional keys are necessary.

10 Security and Further Aspects

In this chapter we evaluate the security of our protocols. In the first section we present attacks on the three solutions. The attacks are subdivided into three categories. The first one deals with possible attacks during the assembly of a component, the second one with attacks in the operating system, and finally with frauds during the disassembly. Section 10.2 deals with the consequences of compromises. The last section considers some restrictions due to the technical environment of suited applications. Several constraints are listed and their effects on the protocol is discussed, whereby most of the considered effects have already been integrated in our protocols.

10.1 Attacks

Most of the attacks on identification protocols presented in Section 3.3 are prevented by the use of challenges, identifiers, and interleaving of the single protocol steps. How some of these attacks can be prevented has already been described in Section 3.4. For example, the use of challenges precludes the replay of authentication steps and protects ID-lists. First prevents that an attacker can eavesdrop an authentication protocol and just replay the response to authenticate himself to the verifier. Latter follows that "old" ID-lists cannot be used by an attacker to harm the system.

In this section we consider some possible attacks during the different periods of a component in a system, namely the *assembly*, the *running system* and the *disassembly*. Note that all attacks only become possible if one or more of our assumptions are not correct, such as that an attacker can compromise secret data or alter records on write-protected lists.

10.1.1 Attacks during the Assembly

In the solution *system check* presented in Chapter 6 the origin of a component is checked before the protocol starts. Thus the protocol is based on the assumption that all components to be built into the system are of an authorized label. Thus it is assumed for all implementations of this protocol that Malory cannot assembly fraud components into the system.
In the solution *proof of origin* presented in Chapter 7 the authorization of a freshly assembled component is checked during the execution of the first procedure *proof of origin*. All other than original parts will lead to an alarm and the protocol stops. An attacker would either have to manipulate the lists hold by the verifiers or compromise the system key K. Malory could not just use a cloned component because it would be detected during the execution of the protocol. He needs to add a new record on the *ready for assembly list* in the symmetric solution or erase the record on the *CRL* in the asymmetric solution. In the first scenario Malory's component has to hold the same secret key as the list, which is assumed to be impossible since the lists are write-protected. In the second scenario Malory needs to know a formerly valid certificate and the corresponding private key. Then he has to erase that record on the *CRL*. We assume that Malory cannot generate a new certificate. Furthermore we assume that the *CRL* is write-protected and stored on a secured server. Since the system key K is stored read-protected in each component Malory cannot compromise the system.
In the solution *proof of origin and system check* presented in Chapter 8 Malory would need to manipulate lists on the verifier's side or compromise the system key. As before we assume this to be impossible.

10.1.2 Attacks in the Running System

As discussed in the previous section passing the checks during the assembly with a counterfeit is unlikely. Hence an attacker might simply not start the protocol, i.e., the forged component will not send the *hello* message to start the protocol. Instead he builds his component into the system without noticing the system.
If a component is *added* to the system it will not be noticed during the *system check* since this component does not hold a record on the system's ID-list. The component will not obtain the system key K and can neither talk to other system components nor eavesdropping on them. We consider crypto analysis infeasible due to strong

encryption schemes.
Malory could *exchange* a valid system component with a counterfeit. This would be noticed during the next execution of the *system check* since the challenge of the removed component would remain unacknowledged. The verifier then sets an alarm and takes further measures. Illegally removed components will still be challenged because they hold a record on the current ID-list. The unauthorized replacements cannot response in a right manner and the attacks fails. If Malory erases the record of the ID on the write-protected ID-list, still needs to compromise K, as described in the previous paragraph. We can come to the conclusion that the disassembly of a system component will be noticed during the *system check* and the unauthorized replacement either will not be challenged at all or, if holding a valid ID, cannot response in a right manner. Note that obtaining a valid ID is easy since the identifiers are public but does not gain any useful information for an attack.

10.1.3 Attacks During the Disassembly

During the *disassembly* procedure the component gets authorized for a re-assembly into another system. Malory could try to take advantage of this procedure. First he steals a component out of its system, then build it into another system and finally start the procedure of *disassembly* in the new system. The component would then be authorized for a new assembly and the component could legally start the procedure of *assembly*. This attack has been regarded in Section 8.1.5 and 8.2.5. The countermeasures are discussed there as well. A component which starts the procedure of *disassembly* has to prove that it is still in its original system, otherwise the procedure cannot be started. Thus illegally removed parts cannot be legally used in other systems. In addition missing parts will be detected in the original system during the next execution of the *system check*.

10.2 What Happens if the System is Compromised

As discussed in the previous Section 10.1 all considered attacks are only feasible if one or more of the made assumptions are not correct. In this section it will be considered what happens if the system is compromised regardless of the feasibility of an attack. The dimension of any compromise, if and how it will be noticed and which measures have to be taken after noticing the successful attack will be considered for

all secret data separately in the following.

The system key K

If Malory can compromise K he is able to mount all his counterfeits with K and to build them into the corresponding system. He also needs to hold valid IDs of components which are recorded on the current ID-list. These records need to be replaced by the counterfeits' ID. Hence either the IDs hold by the counterfeits have to match the illegally replaced components, or Malory has to alter the ID-list of all system components. Since the ID-lists are stored write-protected by each system component the first attack is more likely to succeed. If this attack cannot be automated only one system is affected. Malory can only use his counterfeits in this particular system. Revealing *one* K affects only the security of *one* system. Only if Malory can find a way to easily obtain K in any system he could use this attack on a larger scale. The disclosure of the key cannot be noticed at all within the system. It might be noticed while manually checking the system, e.g., a mechanic might notice forged parts. When an attack of those kind is noticed, the system key of all components should be updated after removing all bogus parts.

The secret key K_i

If Malory is able to obtain the secret key K_i of a system component C_i he could use this symmetric key as replacement of an original part. Since clones are detected in all presented solutions a particular secret key can only be used by one component. Only if Malory could generate arbitrary secret keys and put them onto the corresponding list hold by the verifier this attack would become useful on a larger scale.

The attack of a single component will not be detected by the system. The altering of data hold by the verifier should be noticed by the verifiers.

There are no other measures than eliminating all security lacks and performing an update of the system key K after removing all malicious parts.

The private key S_i

The private key S_i of a component C_i in an asymmetric solution is the equivalent to the secret key K_i in a symmetric solution. Hence only single components are affected if clone detection is provided by the system. Contrariwise to the symmetric key Malory cannot generate arbitrary key pairs since he is not in possession of the authentic issuer's key.

The fraud with one component cannot be detected during the protocol execution. The countermeasures are the same as in the previous paragraph.

The public key P_m

If Malory could exchange the public key of the manufacturer he could certify his own fraud parts. During the identification process he would send the fraud certificates and the fraud public key. The verifier would successfully verify this components by using the replaced public key.

The private key S_m

If the secret key of the certificate issuer is compromised all systems are compromised. Malory could simply generate key pairs and certificates and thus the counterfeits would not be distinguishable of the originals. A disclosure of this secret key would lead to the total breakdown of all systems.

The disclosure will not be detected in the running system. After noticing the successful attack the entire system has to be totally renewed.

The ID-list

Being able to illegally alter the ID-list might be helpful for running an attack if the component's secret key can be readout as well. Therefore it is referred to read the previous paragraphs. Erasing records of components on the ID-list enables an attacker to remove or manipulate those components without system's notice. Only if devices provide a display showing all system parts the user may notice missing parts. A stolen component which record was erased from the ID-list will not be noticed during the next *system check*, thus the ID-list should be protected adequately.

List provided by the verifier

All lists hold by the verifier need to be highly protected. If they are altered the entire system breaks down. Putting records on the *ready for assembly-list* would enable an attacker to legally assembly faked components into systems. Erasing records on the *CRL* would enable the use of malicious parts which have been already detected as frauds before.

10.3 Key Aspects

First of all it is important to select the adequate key size for each solution. The selection mainly depends on the target security level and the used encryption scheme but it is also affected by the provided memory space and the execution time. Some of the parameters are interdependent and usually a trade-off between the security level and the system requirements is necessary. As stated in [30] a necessary, but usually not sufficient, condition for an encryption scheme to be secure is that the key space be large enough to preclude exhaustive search. Hints for selecting an adequate

key length in asymmetric encryption schemes are given in [28].
Besides choosing the key size an adequate protection of the key material is needed. The public key data and the lists have to be stored write protected, i.e., the data cannot be altered or manipulated by unauthorized entities. The private or secret data usually has to be stored in a write and read protected manner. In some of the patent specifications considered in Chapter 2 the keys and IDs are stored in EEPROMs with anti tamper feature. Properties of tamper resistance and tamper evidence can be found in [5], [40], [6] and [27], whereby latter is dedicated for scenarios using smartcards.

10.4 Constraints Due to the Technical Environment

Most of the suited applications are restricted due to their technical environment in many different ways. For instance, components with embedded chips are likely not capable to perform complex computations. A memory chip with anti-tamper feature probably does not provide a large key memory. The typical constraints of ad-hoc networks can be found in [41, 42]. The used data bus and the operating system in vehicles, the CAN (Controler Area Network) bus and OSEK(german acronym Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug which means open systems and corresponding interfaces for automotive electronics), respectively, are highly effecting the protocols. The CAN bus is a serial bus which supports unique addresses all components. The CAN bus defines the bit length of the packets, restricts the number of connected parts to 128, is collision free and supports real-time applications [19]. Further constraints of other application environments are listed and briefly discussed below:

- No backbone structure to connect the system to the Internet

  Some systems are not connected to a server and thus no up-to-date data can be provided to the system. Thus the protocol including all procedures should be executable without a server. Other entities have to take over the role of the server for acting as verifier. A High Security Modules (HSMs) or system components replacing the server take this role.

- Peanut CPU

  Due to the small computing power computations are rather slow. Our protocols symmetric encryption where possible, and execute only as few protocol

steps as necessary.

- Battery power

  The total energy available to a node might be a scarce resource. By using as few protocol steps as possible and thus performing as less computations as possible the energy consumption is optimized as well. Further measures like sleeping modes of components and preventing denial of service attacks are not regarded in the presented protocol. The underlying layer, for instance the operating system, has to deal with these issues.

- High latency

  In case that components switch to sleeping mode communication with those components would involve waiting until they wake up.

- Small memory

  Memory space is limited in many applications which should be kept in mind when using several keys and lists. In these cases a solution with a HSM or external server is preferable because they hold the extensive data, as for instance the required lists. The required memory space is dependent of the number of components within the system, the key lengths and the amount of further stored data.

- Protection of the memory

  As mentioned before some memory space has to be read and/or write protected. For providing these properties components with anti-tamper features are necessary. It should be noted that the protection of the entity's memory or the entity in general is depending on its accessibility. For instance, an external server can be kept in a locked room which is only accessible by authorized people. On the other side the system has to be open and thus it needs to be especially protected. For example, the owner of a car has permanent access to the parts and is able to execute arbitrary attacks. The attacker could even take a component apart to analyze the chip.

- Data bus

  The protocols execution time depends on the data bus which is used for the communication among the participants. The *bandwidth* of the bus limits the

size of the exchanged packets. The messages used in the presented protocols are as short as possible and thus reduce the number of actually exchanged packets to a minimum. The *transfer rate* of the bus affects the time period of a protocol execution. This is especially important in real-time environments. Another property which directly affects the protocol flow of some procedures is the capability of *multicasting*. In systems supporting multicasting one component can send a message to all components. This would highly accelerate the execution of procedures. Some systems may provide the possibility of sending several messages at once. This would also accelerate the execution. For instance, when executing the *system check* as described in Section 6.4 with the parameter 3a, the verifier could send all messages to all system components at once. Note that the transmission of parallel messages depends on the bandwidth of the bus.

- Special environment

  In many applications the components are exposed to some extraordinary circumstances. For instance in an automobile the parts are exposed to high variations in temperature, extreme vibrations, humidity etc.. The used parts including their embedded chips need to be solid and robust. This has to be ensured by the proper selection of the parts and chip technology.

11 Summary and Suggestions for Future Work

In this thesis solutions for providing component identification were introduced. The thesis can serve as a reference to component identification in systems consisting of several connected and removable components. Once the component's identity can be ensured many applications become feasible. We derived piracy protection, theft and system protection, or a combination of both as suited applications. The combination of these security features have been never considered before. The use of cryptography in a detailed protocol flow are presented in general and in particular for many different circumstances. The selection of parameters and assumptions for the particular application helps to finetune the protocol and customize it for the individual needs. Furthermore all presented protocol can easily be expanded by additional parameters or features. This makes our solutions future-proved and flexible. The general trend is that more and more devices contain embedded chips and more and more components are connected to each other. Thus our solutions will become applicable for more and more systems. In the future many components will contain intelligent chips and are accessible via the Internet. This would lead to a server solution of our protocols which is usually the easiest and most convenient realizations of the protocols. The links between the components presently consisting out of copper cable in most applications will change to fiber, as already implemented in some automobiles. The next step are radio links which leads to wireless systems. The use of new transmission media improves the transmission rate. Wire-less transmission of data would not pose a threat in our protocol since all confidential messages are always exchanged encrypted. Thus the presented solutions can easily be adapted to wire-less scenarios. Furthermore, the computation power of chips is increasing as well. Faster transmission rates and faster microprocessors could enable the use of time consuming encryption schemes even in constrained environments as for instance on smartcards. The use of asymmetric encryption scheme will also become

possible in most systems which would lead to the asymmetric solutions of the presented protocols. The lack of slow execution would not be an issue anymore and then these protocols would become preferable since the potential communication partners do not need to know each other a priori.

Several ideas for future work naturally came up. Most of them were briefly presented in Chapter 9. It is always important to adapt the security system to its environment. Since the environment is changing the solutions have to be changed as well. Some of the suggestions made for enhancements and improvements of the protocols might become possible in the future.

Particularly suited for the implementation of the protocol *proof of origin* are airplanes. By preventing the installation of bogus parts the safety of all passengers would be highly increased. Since all airplanes are checked before taking off, the protocol could run during this period. Instead of using tags as in nowadays systems, data which is electronically verifiable could be used to authenticate all parts. This is not only more convenient but also more secure since nobody else than the manufacturer of licensed components would have to be assumed to be trustworthy.

Especially suited for the implementation of the protocol *proof of origin and system check* are automobiles. From the view of the manufacturer and suppliers this realization could be used to gain profit. For the owner of the vehicle the property of theft protection is of special interest.

Bibliography

[1] all4engineers. BMW stellt On-line-Dienste auf der CeBit vor. T. Jungmann. newsletter 12/3/02.

[2] all4engineers. Infineon zeigt integrierte Plattform für drahtlose Kommunikation im Fahrzeug. T. Jungmann. newsletter 22/3/02.

[3] all4engineers. Studie: Biometrie macht Autos sicherer. T. Jungmann. newsletter 7/6/02.

[4] R. J. Anderson. On the Security of Digital Tachographs. Cambridge University Laboratory.

[5] R. J. Anderson. Security Engineering- A Guide to Building Dependable Distributed Systems. Wiley, 2001.

[6] R. J. Anderson and M. Kuhn. Tamper Resistance- A Cautionary Note. *Proceedings of the 2^{nd} USENIX Workshop on Electronic Commerce, page 1-11, 1996.*

[7] Automobil Industrie. Kfz-Produktpiraterie erreicht industrielle Ausmaße. 24/7/2001.

[8] BBC News. Global alert over faulty plane parts. January 29, 2002.

[9] BBC News. Italian police probe ‘ plane parts scam’. January 26, 2002.

[10] D. Bleichenbacher and U. Maurer. On the Efficiency of One-time Digital Signatures. Bell Laboratories and Department of Computer Science, Swiss Federal Institute of Technology.

[11] Bundesdruckerei. `http://www.bundesdruckerei.de/en/produkte/index.html`.

[12] Business Week. Warning! Bogus parts have turned up commercial jets. Where's the FAA? W. Stern. June 10, 1996.

[13] c't magazin. Erbson-Zähler. T. Gerber. 12/01, page 66.

[14] c't magazin. Intelligenztest- Chipverdongelung von Epson-Tintenpatronen aushebeln. U. Hilgefort, T. Gerber. 13/01, page 213.

[15] der tagesspiegel. Autohersteller jagen Produktpiraten. Branchen und Verbände. 23/02/1999.

[16] Y. Desmedt. Some Recent Research Aspects of Threshold Cryptography. University of Wisconsin-Milwaukee.

[17] Y. Desmedt and Y. Frankel. Threshold Cryptosystem. University of Wisconsin-Milwaukee, 1998.

[18] die tageszeitung. Schrotthändler der Lüfte. M. Braun. 28/1/2002.

[19] H. Engels. CAN-Bus. Franzis Verlag GmbH, 2002.

[20] S. Even, O. Goldreich and S. Micali. On-Line/Off-Line Digital Signatures. *Crypto'89*. Non-final version from 1994.

[21] Y. Frankel and Y. Desmedt. Parallel reliable threshold multisignature. University of Wisconsin-Milwaukee.

[22] H. Ghodesi, J. Pieprzyk and R. Safavi-Naini. A flexible Threshold Cryptosystem. Center for Computer Security Research.

[23] L. Gong. Variations on the Themes of Message Freshness and Replay, or the Difficulty of Devising Formal Methods to Analyze Cryptographic Protocols. *In Proceedings of the Computer Security Foundations Workshop VI, pages 131-136*. IEEE Computer Society Press, 1993.

[24] Heise online news. Canon verklagt Pelikan wegen Tintenpatronen. Verlag Heinz Heise, 16/4/2002.

[25] Heise online news. Wie man SIM-Karten fälscht. Verlag Heinz Heise, 7/5/2002.

[26] A. Huang. Keeping Secrets in Hardware: the Microsoft X-BOX Case Study. *CHES Conference, 2002.*

[27] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. *In USENIX Workshop on Smartcard Technology, 1999.*

[28] A. K. Lenstra and E. R. Verheul. Selecting Cryptographic Keysizes. Citibank, N.A., Universiteit Eindhoven, Pricewaterhouse Coopers.

[29] M. Lischka. Verteilte kompetenzbasierte Authentifizierung von Knoten im Dragon Slayer III System. Diplomarbeit, Fachbereich Informatik, Universität Dortmund, Juli 1998.

[30] A. J. Menezes, P. C. von Orschot and S. A. Vanstone. Handbook of Applied Cryptography. 1997 by CRC press LLC.

[31] R. C. Merkle. A digital Signature based on a conventional encryption function. Elxi, San Jose, CA.

[32] Offenlegungsschrift. Ausbausicherung für elektronische Komponenten bei einem Fahrzeug. DE 100 21 811 A1. 15/11/01. Daimler Chrysler.

[33] Offenlegungsschrift. Kraftfahrzeug mit einer Vielzahl von Bauteilen. DE 100 01 986 A1.26/7/01. Volkswagen AG.

[34] Offenlegungsschrift. Sicherheitssystem für einen mobilen Ausrüstungsgegenstand. DE 41 23 666 A1. 9/7/92. The Intelex Corp., Paramus, N.J., US.

[35] Offenlegungsschrift. Sicherheitssystem zu Gültigkeitsprüfung von Bauteilen. DE368816 T2. 28/10/93. Int. Business Machines Corp.

[36] Offenlegungsschrift. Verfahren und Mittel zum Bereitstellen einer seriellen Identifikation und Erkennung von geschätzten Zeichen in Komponenten wie beispilesweise Silicium- und Keramiksubstraten. DE 100 61 741 A1. 30/8/2001. AccuCorp Technical Services, Inc., Fremont, Calif., US.

[37] Offenlegungsschrift. Verfahren zur kryptografisch prüfbaren Identifikation einer physikalischen Einheit in einem offenen drahtlosen Telekommunikationsnetz. DE 100 26 356 A1. 29/11/01. Bosch.

[38] L. Reyzin and N. Reyzin. Better than BiBa: Short One-time Signatures with Fast Signing and Verifying. January 30, 2002.

[39] Spiegel Online. Airbus-Absturz auf Queens: Ersatzteilfälscher unter Verdacht. 2002.

[40] F. Stajano. Security for ubiquitous computing. Wiley, 2002.

[41] F. Stajano and R. J. Anderson. The Resurrecting Duckling: Security for Ad-hoc Wireless Networks. *In Proceedings of the* 7^{th} *International Workshop on Security Protocols*. Lecture Notes in Computer Science, 1999.

[42] F. Stajano. The Resurrecting Duckling - what next? *In Proceedings of the* 8^{th} *International Workshop on Security Protocols*. Lecture Notes in Computer Science. Springer Verlag, 2000.

[43] tecchannel. Bluetooth-Produkte für jeden Einsatz- Bluetooth im Auto. 2/8/00.

[44] Time. Flying into trouble. M. Schiavo. March 31, 1997.

[45] United States Patent. Method of Legitimate Product Identification and Seals and Identification Apparatus. 5,426,520. June 20, 1995. Shooei Printing Co., Ltd.; AMC Co., Lttd., both Osaka, Japan.

[46] United States Patent. Vehicle anti-theft system including vehicle identification numbers programmed into on-board computers. 5,991,673. 23/11/99. Lear Automotive Dearborn.

[47] United States Patent. Vehicle Occupant Protection Device and System having anti-theft, anti-tamper feature. 6,249,228. 19/6/01. Assignees: TRW Inc.

[48] United States Patent. Vehicle part identification system and method. 6,317,026. 13/11/01. Brodine.

[49] S. Vaudeney. One-time identification with low memory. Laboratoire d'Informatique de l'Ecole Normale Supérieure, Paris, France.

[50] Verpackungsrundschau. miscellaneous editions. `http://verpackunsrundschau.de`.

[51] Verpackungsrundschau. Produktschutz durch OVDs. `http://verpackunsrundschau.de`. October 2001.

[52] ZDNet Deutschland News. Fälschungen von Telefonkarten nicht unsere Schuld. S. Rieger. 10/7/2001.

[53] ZDNet Deutschland News. Gefälschte Handys auf den Vormarsch. M. Fiutak. 11/7/2001.