Formal Security Analysis of Electronic Software Distribution Systems

Monika Maidl[1], David von Oheimb[1]
Peter Hartmann[2], Richard Robinson[3]

[1]Siemens Corporate Technology, Otto-Hahn Ring 6, 80200 München, Germany
{monika.maidl, david.von.oheimb}@siemens.com
[2]Landshut University of Appl. Sciences, Am Lurzenhof 1, 84036 Landshut, Germany
peter.hartmann@fh-landshut.de
[3]Boeing Phantom Works, P. O. Box 3707, MC 7L-70, Seattle, WA 98127-2207, USA
richard.v.robinson@boeing.com

Abstract. Software distribution to target devices like factory controllers, medical instruments, vehicles or airplanes is increasingly performed electronically over insecure networks. Such software often implements vital functionality, and so the software distribution process can be highly critical, both from the safety and the security perspective. In this paper, we introduce a novel software distribution system architecture with a generic core component, such that the overall software transport from the supplier to the target device is an interaction of several instances of this core component communicating over insecure networks. The main advantage of this architecture is reduction of development and certification costs. The second contribution of this paper describes the validation and verification of the proposed system. We use a mix of formal methods, more precisely the AVISPA tool, and the Common Criteria (CC) methodology, to achieve high confidence in the security of the software distribution system at moderate costs.

1 Introduction

1.1 Network enabled Software Distribution

In recent years, computer systems that support industrial applications, energy management and distribution, transportation systems, medical and many other applications started to use network interconnections for a range of communication needs. One such need is the distribution of software to devices in the field, in particular to allow for software updates. If such software is used to implement critical functionality that can affect the safety of people or valuable property, the software distribution process itself becomes highly critical. In other words, networked software distribution makes the safety and/or security of a system dependent upon securing

communication over potentially insecure channels, facing threats like corruption, injection, diversion, replay, and disclosure of the software payload.

Various methods can be used to ensure security properties of networked systems. However, methods typically used in software development, such as testing, do not work well for security properties due to the severe consequences of subtle errors or small oversights. After all, security properties have to hold in the presence of attackers who actively try to exploit any weaknesses. A better approach to assess security of systems is to work with a well-designed catalog of requirements that is based on a broad range of experience. Certification according to Common Criteria, as discussed in the next section, falls into this category. Another proven approach is to use exhaustive search as offered by formal methods, in our case by model checking.

1.2 Security Certification

For assessing the security of a system, i.e., assuring that the system implements countermeasures for all relevant security threats, the Common Criteria (CC) [5] is one of the most advanced and widely accepted methodologies. The aim of an evaluation according to the CC is to systematically and objectively demonstrate that the countermeasures are sufficient and correctly implemented. The first step is to produce a specification called Security Target (ST). It defines the Target of Evaluation (TOE) which is the software, firmware and/or hardware component(s) to be evaluated, identifies threats the TOE is exposed to, derives objectives to cover the threats, states functional requirements to implement the objectives, and demands assurance requirements. The Security Target can be an instance of a generic Protection Profile (PP) which specifies the evaluation of a class of systems. We have defined such PPs for an Airplane Asset Distribution System (AADS) and its core component [7].

The CC predefined Evaluation Assurance Levels (EALs) range from 1 to 7 and determine the rigor and depth of the analysis process. Evaluation at high assurance levels, i.e., EAL5-EAL7, requires high effort for the design and implementation and also for the CC evaluation. For example, EAL6 requires a semiformally verified design based on a formal security model, and EAL7 requires full formal verification.

In [8] we have determined the assurance levels that must be met by a distribution system for airplane software. Given the high criticality of some airplane software, according to the NSA, EAL6 is recommended for safety-relevant threats, whereas EAL4 is shown sufficient for threats on airline business. In general, the distribution of software controlling safety-critical processes will require a high assurance level.

Usually CC certifications are applied to single strongly confined IT components, not to whole distributed systems consisting of several interacting entities. This is done mainly in order to limit the evaluation effort. The component-wise certification of complex systems also gives flexibility for the assembly of the overall system: components may be developed and certified individually, even by different partners.

On the other hand, we face the composition problem: the threats and vulnerabilities at system level may be different from the ones at component level. Therefore, whether the security objectives of the overall system are met as a consequence of the security properties of the individually certified components is a question to be addressed separately. The latest version 3.1 of the CC provides a first step to address this

problem by providing composed assurance package (CAP) evaluations. However, CAP evaluations cannot achieve a high evaluation assurance level.

1.3 Model checking

As mentioned above, high assurance calls for formal analysis. Tool-supported formal methods range from automatic model checkers to powerful theorem provers. In the last years, several tools targeted for the verification of security protocols, i.e. protocols that are based on the use of cryptographic measures, have been developed and proven very successful. Among those, the AVISPA tool [1,2] offers a front-end and several model checkers. In its design special care has been given to offer easy use even in an industrial setting. It has been applied to many protocols, mainly of the IETF. Other tools for verifying correctness of security protocols are ProVerif [3], based on resolution theorem proving, and LySa [4], which is based on static analysis.

1.4 Our Contributions

Based on our experience with software distribution for avionics, automotive, and healthcare equipment, we define a generic system architecture for a Software Distribution System (SDS). We simplify the system design and its certification by defining a generic core component, the Software Signer Verifier (SSV), instances of which are used at every node of the system. The overall SDS from the software supplier to the target device is essentially an interaction of several SSV instances.

For a cost-efficient and still rigorous assessment of the distributed SDS, we propose a *hybrid approach*, based on the Common Criteria and on formal methods, that takes advantage of the architecture outlined above and addresses the composition problem for CC-high assurance as mentioned in Section 1.2. We analyze and specify the security requirements for the SSV and for the overall SDS with Protection Profiles like [7]. Assuming that the involved SSV components are certified, we use the AVISPA tool to formally specify and model check that the overall SDS protocol fulfils the security objectives at system level.

The main contributions of this paper are the system architecture for a SDS, its formal model as an abstract security protocol, and the validation of its system-level security properties.

2 System Architecture of the Software Distribution System

2.1 Threats and Security Objectives for a SDS

In [8] we have presented a threat analysis and security objectives for an Airplane Asset Distribution System (AADS). We can generalize those threats to more general software distribution systems as follows:

Corruption. The contents of software items could be altered or replaced.

Injection. The target device's configuration could be affected by invalid software items created by the attacker and installed on the target device.

Diversion. Software items could be diverted to an unsuitable destination, e.g. by disturbing the execution of other software at that destination.

Wrong version. A mismatch between the target's intended and actual configuration could be caused by replaying outdated versions or by forging version numbers.

Disclosure. The attacker can get hold of the software item contents without having a license, or reengineer functionality in order to help manipulating software.

The last threat was not included in [8] because it is not needed in the AADS context. Yet in general, confidentiality might be necessary, e.g. to protect intellectual property

Based on the threats described above we derive a set of security objectives that must be met by the SDS:

Authenticity. Every software item accepted must originate from a genuine supplier.

Integrity. For every software item accepted at a target, its identity and contents must not have been altered on the way—it must be exactly the same as at the supplier.

Confidentiality. If required, software items must be kept secret from the entry point of the SDS (at the supplier) until reaching the target device.

Correct Destination. A target device must accept and receive only software items for which it is the true destination intended by the target operator.

Correct Version. A target device must accept software items only in the latest version approved by the target operator.

Note that the first three requirements are stated *end-to-end*, i.e. they are properties stretching from the initial source of software assets to their final destination. In contrast, *hop-by-hop* properties refer to the transport of assets between adjacent entities, for instance that in each step the integrity of an asset is preserved.

2.2 SDS Architecture

On the way from the software *supplier* to the target device, software items may be handled at intermediate entities: software *distributors* or OEMs might receive the software items from the supplier, and send it to the *target operator*, who bears responsibility for the safe operation of the *target device*, and has the authorization to send software there. So the software distribution process consists of several hops, and the SDS stretches over the IT systems related to the process at each of these entities.

Figure 1 shows the overall flow of software items. Simpler scenarios are possible, e.g. where the operator coincides with the distributor or even with the supplier.

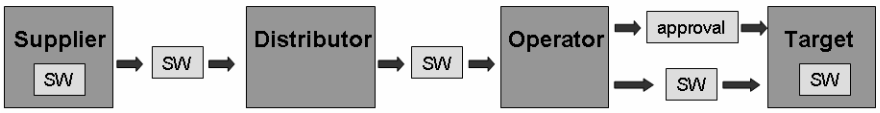

Figure 1: A typical Software Distribution System.

For every transportation step, the software item has to be protected against the threats listed above. Digital signatures and encryption using public key technology are the fundamental security mechanisms used to implement protection for the SDS. Signatures are generated by applying the private key of the sender to the contents, or rather to the hash (which is a cryptographic checksum) of the contents. The recipient applies the corresponding public key, compares the result with the contents which have been received in the clear, and if there are no differences, the receiver can be sure that the contents have not been modified during transport and that only the owner of the private key could have produced this signature. If in addition confidentiality is required, the sender encrypts the signed message with the public key of the receiver. Only the owner of the corresponding private key can decrypt and hence read the contents, not an attacker intercepting it.

The intermediaries might just store and forward the software, or perform some local processing, such as including owner specific license keys and setting target specific software parameters. In any case, the intermediary has to check the signature of the previous entity and might add a new signature.

As the target operator is responsible for its target devices, he has the special task of managing the software configurations on the devices, i.e. deciding which software versions may be installed on which targets. This may take the form of an explicit *installation approval statement* that is sent by the operator to a target, and authorizes the installation of the software item with a suitable version at the specific target instance. We do not specify how installation approval statements are transported securely from the operator to the target. This can be done for example in an out of band communication, or in a protected separate message, or it can be included in the distributed software package.

The target device verifies the integrity and authenticity of the software item using the signature of the operator and checks, using the approval statement of the operator, whether it is an approved recipient of the software item with the given version. Airplane software distribution typically uses an out of band process for the installation approval: the airplane operator (i.e., the airline) issues installation orders in the form of a work order on paper, to be executed by a mechanic. Similar processes might apply in software distribution systems if target devices are located in the vicinity of the operator. For other SDS, administration of the target device should be automatic under remote control of the operator.

We structure the SDS into several instances of a signature application component called Software Signer Verifier (SSV), which is responsible for applying digital signatures on software items before transmitting them, and for verifying signatures on software items received from other entities in the distribution process. For different nodes involved in the software distribution, the SSV can be developed and certified independently or one and the same SSV product can be used at all nodes.

2.3 SSV: the SDS Core Component

Each node in the above distribution chain runs an instance of the SSV, i.e. the SDS core component. The SSV instances are used for:

Introducing unsigned software into the SDS by digitally signing and optionally encrypting it and making it available for other SSV instances.

Verifying the signature on software received from other SSV instances (after decrypting it if needed) and checking the authenticity and authorization of the sender.

Approving the software by adding a signature and optionally re-encrypting the software and making it available to further SSV instances.

Delivering software out of the SDS after successfully verifying it.

Introduction of software into the SDS typically takes place at the supplier, yet may take place also at intermediate entities, while software delivery happens at the software target. All SSV instances except at a supplier verify incoming software. Adding a new signature will be done usually at SSV instances located at distributors and operators after some local processing of the software, such as adding license information or by performing a quality inspection. Such processing is performed within the local environment of the respective. Figure 2 shows the SSV in its environment including the flow of software.

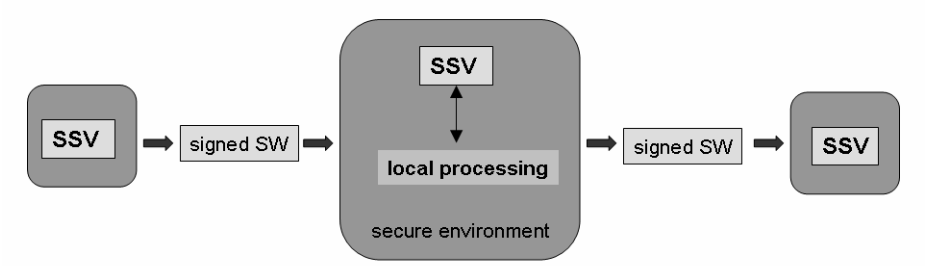

Figure 2: The generic Software Signer Verifier and its environment.

3 Security Assessment of the SDS

3.1 Assumptions on the Operational Environment

Not all assurance issues related to software distribution can be covered by the security assessment of the SDS at reasonable costs. For example the reliability of the Public Key Infrastructure (PKI), which is used to provide keys and certificates for asset protection, is considered out of scope. According to the CC methodology, such aspects are collected as assumptions on the operational environment of the assessed system. The assumptions on the SSV environment are the following:

SSV protection. The SSV instances are protected against direct manipulation and misuse. The SSVs run on a hardened operating system (OS), user access is possible only locally and restricted by effective access control mechanisms. Keys, certificates and other critical data are protected against manipulation and disclosure. Authorized personnel are assumed to be trustworthy.

Secure local environment. The SSV instances and their underlying OS run in a secure local environment, which may contain processing facilities for performing local operations on software items. An adequately configured firewall ensures that the

SSV, its underlying OS and the local environment are not compromised through network access.

Reliable PKI. It is assumed that the PKI used to certify keys used by the SSVs is trustworthy and properly managed. Revocation information is issued regularly and immediately after revocation of a signing key.

Target configuration enforcement. The local environment of the target SSV checks whether the installation of received software items is authorized by an approval statement of the target operator. Depending on the system design, this assumption can be relaxed, e.g. the SSV itself might perform such checks.

3.2 Certification of the SSV

The SSV is a component for which a security target may be produced according to the CC methodology. The document [7] is a Protection Profile (PP) for the SSV in the special case of airplane software distribution; however this PP can easily be adapted to handle the very generic case discussed in this paper. The PP specifies the security objectives of integrity and authenticity and – if required – of confidentiality at the component level, and hence after successful CC certification of the SSV instances, there will be sufficient evidence that the security mechanisms of SSVs achieve these security objectives under the assumptions on the SSV environment stated above. The remaining security objectives of correct destination and correct version and end-to-end integrity and authenticity will be covered at system level by the formal analysis described in the next sections.

3.3 The Protocol for End-to-End Software Distribution

In order to assess the correctness of the SDS at system level, we consider the interaction between the SSV instances located at the different nodes. As the interaction consists of exchanging cryptographically secured messages, we have chosen the form of a cryptographic protocol analysis.

First we present the protocol in the common Alice-Bob notation. The different nodes are abbreviated as follows: SUP software supplier, `DIS` software distributor, `OP` target operator, `TD` target device and `CA` certificate authority. For each node N, the associated private key is denoted by `inv(KN).`

In the first step, the supplier SSV imports assets from its local environment. In every further step, the SSV at the respective node receives a signed asset and checks the signature. Except in the last step, the SSV adds its approval signature, encrypts the whole message if needed, and sends the new message to the next SSV instance.

```
1. SUP - {Asset.{h(Asset).DIS}_inv(KSUP).CertSUP}_KDIS -> DIS
2. DIS - {Asset.{h(Asset).DIS}_inv(KSUP).CertSUP
               .{h(Asset).OP }_inv(KDIS).CertDIS}_KOP  -> OP
3. OP  - {Asset.{h(Asset).DIS}_inv(KSUP).CertSUP
               .{h(Asset).OP }_inv(KDIS).CertDIS
               .{h(Asset).TD }_inv(KOP ).CertOP}_KTD   -> TD
```

We shortly explain the constructs used in Alice-Bob notation:

`A - M -> B`	means message `M` sent from `A` to `B`
`Asset`	means a software item including its identity
`M.N`	means the concatenated contents of M and N
`h(M)`	means the hash value of content `M`
`{M}_inv(K)`	means content `M` signed with private key `K`
`{M}_K`	means content `M` encrypted with public key `K`

As usual when producing a signature, not the asset itself is signed but only its hash value. Note that the signature also includes the identity of the intended receiver. The sender's certificate, which ties the sender's identity together with its public key, is also included into the message. The certificates are self-signed or signed by a certificate authority (CA) that confirms the identity of the certificate holder.

In the SDS protocol, signatures are applied in parallel: every SSV keeps the old signatures and adds its own. However, each SSV only checks the signature applied by its immediate predecessor, but not the signatures applied in the steps before, as it is not assumed that an SSV has a trust relationship with all previous nodes. For example, the target device trusts its operator, but is not configured to know all potential suppliers. For signatures with self-signed certificates, the check consists in looking up the public key in a locally stored set of authorized senders. For instance, the target device typically knows the public key of its operator. For CA-signed certificates, the CA key has to be contained in a locally stored set of public keys of trusted CAs. We assume that the two locally stored sets of trusted public keys are managed by a trustworthy administrator.

We do not model installation approvals explicitly. Instead, we model part of the approval information by including the identity of the intended target device in the asset signature applied by the operator.

3.4 Security Properties

For the SDS protocol, we formally validate the authenticity of the asset origin and the integrity and confidentiality during asset transport. More precisely, we show that

(1) assets accepted by the target device have indeed been sent by the supplier,
(2) assets accepted by the target have not been modified during transport,
(3) asset authenticity and integrity also hop-by-hop, i.e. from any SSV instance to the next, in particular between the operator and target device, and
(4) assets remain secret among the SSVs.

Clearly the security objectives of *authenticity*, *integrity* and *confidentiality*, stated in Section 2.1, are covered by (1), (2), and (4). Further, when sending a message, every sender includes the name of the intended receiver in the signature, and the receiving SSV checks whether it is the intended destination, so together with (3), the objective *correct destination* is also satisfied. In other words, the signature of the operator containing the name of the target device models part of the installation approval statement for the asset. The remaining part of the installation approval statement, namely the version information, is not contained in our model. The corresponding security objective of *correct version* is covered by the *target*

configuration enforcement assumption, i.e. that version checking is done by the SSV local environment.

Hence the formal analysis, presented in the next section, implies that our formal model of the SDS architecture satisfies the security objectives at the system level. As the implementation details of the SSVs at the different nodes are covered by CC certification, we gain substantial confidence in the overall security of the SDS.

4 Formal Analysis of the SDS Protocol

The Alice-Bob notation, showing only message exchanges, is not detailed and precise enough for any thorough analysis. It leaves important processing steps implicit, in particular the checks an agent performs to accept a message and the parts of received messages and other state information the agent uses to construct further messages. The specification language of the AVISPA tool, HLPSL, offers constructs to express all steps involved in the message exchange in a precise, declarative way. Agents are defined generically as a role, of which multiple instances may exist in a given system or scenario. The behavior of a role is specified as a set of state transitions. During such a transition, an agent receives and checks messages before sending new messages, which then can be received in a transition by another agent.

Instead of individually modeling all roles, i.e. supplier, distributor, operator and target device, we use the fact that all run an instance of the SSV component. Hence we can specify a parameterized role, called SSV, which is then instantiated multiple times to represent the overall SDS protocol.

Figure 3 shows the header declaration for the SSV role. The parameters are used to configure the different instances, e.g. Import is true if signed assets may be received. The parameter KeySet holds a set of public keys that acts as authorization information: software items signed with a key in this set are accepted. For instance, the target device only accepts software items signed by its operator. Alternatively, signed software items can be sent together with a CA-signed certificate, and are accepted if the public key of the CA is contained in `KCASet`.

The local variables of the SSV include the variable `State`, which acts as a program counter, and others that are mainly used to hold values received in messages.

```
role softwareSignerVerifier(
        SND,RCV: channel(dy),
        SessN: nat,     % session number, needed just for technical reasons
        SUP,TD: agent, % supplier and target, just for expressing asset_end_to_end
        Import,Export: bool,% Import is true if a signed asset is expected,
                            % Export is true if a signature has to be added.
        SSV, NextSSV: agent,
        KSSV,KNextSSV: public_key,   % public key of this SSV and the one to
                                     % which it sends messages
        CertSSV: {agent.public_key
                 }_inv(public_key), % certificate for the private key inv(KSSV)
        KCASet: public_key set,      % set of accepted CA certificates
        KeySet: public_key set       % set of public keys of authorized senders
)
local
```

```
      State: nat,
      Asset: text,
      Msg,X,PrevSigs: message,
      KCA,KprevSSV: public_key,
      Cert: {agent.public_key}_inv(public_key),
      PrevSSV: agent

init
      State := 0
```

Figure 3: Header and local variables of the SSV role

There are five transition rules, presented in Figure 4. The first covers the case that an asset is imported from the local environment (in unsigned form). The second and third rules cover the reception of a signed part, authorized either by a CA-signed certificate or by a public key contained in the internal key set. The remaining two rules describe what the SSV does with the received asset: either forward it in signed form to the next one, or consume it.

```
transition

introduceNew.
     State  = 0 /\ Import = false /\ RCV(start)
 =|> State':= 1 /\ Asset' := new() /\ PrevSigs' := nil
     /\ secret(Asset',asset,{})

importCASignedCert.
     State  = 0 /\ Import = true
     /\ RCV({Asset'.PrevSigs'}_KSSV)
     /\ PrevSigs' =
           X'.({h(Asset').SSV.SessN}_inv(KprevSSV').Cert')
     /\ Cert' = {PrevSSV'.KprevSSV'}_inv(KCA')
     /\ in(KCA',KCASet) % check if CA is in the accepted CA set
 =|> State':= 1
     /\ wrequest(SSV,PrevSSV',asset_hop_by_hop,Asset')

importSelfSignedCert.
     State  = 0 /\ Import = true
     /\ RCV({Asset'.PrevSigs'}_KSSV)
     /\ PrevSigs' =
            X'.({h(Asset').SSV.SessN}_inv(KprevSSV').Cert')
     /\ Cert' = {PrevSSV'.KprevSSV'}_inv(KprevSSV')
     /\ in(KprevSSV',KeySet) % check if signing key acceptable
 =|> State':= 1
     /\ wrequest(SSV,PrevSSV',asset_hop_by_hop,Asset')
```

```
send.
     State  = 1 /\ Export = true /\ RCV(start)
 =|> State':= 2
     /\ SND({Asset.PrevSigs.({h(Asset).NextSSV.SessN}_inv(KSSV)
            .CertSSV)}_KNextSSV)
     /\ witness(SSV,NextSSV,asset_hop_by_hop,Asset)
     /\ witness(SUP,TD     ,asset_end_to_end,Asset)

final.
     State  = 1 /\ Export = false /\ RCV(start)
 =|> State':= 2 /\ wrequest(TD,SUP,asset_end_to_end,Asset)
```

Figure 4: Transitions of the SSV role

We explain the second transition in more detail. A transition is divided into a condition part in which a message may be received and checked, and an action part in which a message may be sent. Variables can occur in a transition in primed or unprimed form, where the unprimed from refers to the value of the variable *before* the transition, whereas the primed form refers to the value of the same variable *after* the transition. Variables can obtain a new value once during a transition, either by assignments, written in the action part, or by pattern matching in the condition part, typically during reception of a message. For example, `State = 0` means the condition that the variable `State` has the value zero, while `State':= 1` means that the variable `State` is assigned a new value: one. The expression `{Asset'.PrevSigs'}_KSSV` means that a message that must be encrypted with the key `KSSV` is received, the first part of which is stored in the variable `Asset`, and the second part is stored in `PrevSigs`. The next line specifies the constraint that the second part of the message has a specific form, namely `X'.({h(Asset').SSV.SessN}_inv(KprevSSV').Cert')`. As `Asset` has already been assigned a value in this transition, in this way it is checked whether the hash value of the asset is correct. Furthermore, the name of the receiving agent must be the identity of the current `SSV` The public key with which the signature can be validated is stored in `KprevSSV`. Next the certificate is validated: It has to contain the identity of `KprevSSV`, and has to be signed by a CA whose public key is contained in `KCASet`. As a by-product of these checks, the SSV learns the identity of the sender, stored in the variable `PrevSSV`. As explained above, the SSV checks only the signature applied by the direct sender. This is modeled by using the variable `X'` for the signatures that are not handled and by not performing verification on `X'`

Figure 5 shows the composed role called `session`, which ties together the instantiations of the SSV needed for the end-to-end transport of one asset. Each instantiated SSV is configured with its own parameters. For instance, the eighth parameter is the name of the agent, i.e. SUP in the first instantiation, DIS in the second and so on.

```
role session(SND,RCV: channel(dy),SessN: nat,
     SUP,DIS,OP,TD: agent,
     KSUP,KDIS,KOP,KTD,KCA: public_key,
     SUPCert,DISCert,OPCert,TDCert:
                          {agent.public_key}_inv(public_key),
     SUPKeySet,DISKeySet,OPKeySet,TDKeySet: public_key set)
def=

composition
   softwareSignerVerifier(SND,RCV,SessN,SUP,TD,false,true,
                        SUP,DIS,KSUP,KDIS,SUPCert,{KCA},SUPKeys)
/\ softwareSignerVerifier(SND,RCV,SessN,SUP,TD,true,true,
                        DIS,OP ,KDIS,KOP ,DISCert,{KCA},DISKeys)
/\ softwareSignerVerifier(SND,RCV,SessN,SUP,TD,true,true,
                        OP ,TD ,KOP ,KTD ,OPCert ,{KCA},OPKeys)
/\ softwareSignerVerifier(SND,RCV,SessN,SUP,TD,true,false,
                        TD,none,KTD,knone,TDCert ,{}   ,TDKeys)
end role
```

Figure 5: Specification of the session role

The last part of the model specifies the environment, including initializing channels and other parameters, defining the initial knowledge of the attacker, and starting three different sessions of the protocol, for instance a session between supplier `sup1` with a CA-signed certificate `Sup1Cert`, distributor `dis`, operator `op`, and target device `td`.

```
role environment() def=

local
      SND,RCV: channel(dy),
      SUP1Cert,SUP2Cert,DISCert,OPCert,TDCert:
                          {agent.public_key}_inv(public_key),
      SUPKeys,DISKeys,OPKeys,TDKeys: public_key set

const
       sessN1,sessN2,sessN3: nat,
        sup1, sup2, sup3, dis, op, td    : agent,
       ksup1,ksup2,ksup3,kdis,kop,ktd,kca: public_key,
       asset_hop_by_hop,asset_end_to_end,asset: protocol_id

init
       SUP1Cert := {sup1.ksup1}_inv(kca  ) /\
       SUP2Cert := {sup2.ksup2}_inv(ksup2) /\ % self-signed
        DISCert := {dis .kdis }_inv(kca  ) /\
          OPCert := {op  .kop  }_inv(kop  ) /\ % self-signed
          TDCert := {td  .ktd  }_inv(ktd  ) /\ % self-signed, unused
       SUPKeys  := {}             /\          % unused
       DISKeys  := {ksup2, ksup3} /\          % ksup3 is unused
       OPKeys   := {}             /\
       TDKeys   := {kop}

intruder_knowledge = { sup1, sup2, sup3, dis, op, td,
                       ksup1,ksup2,ksup3,kdis,kop,ktd,kca}
```

```
composition
       session(SND,RCV,sessN1, sup1, dis, op, td,
                               ksup1,kdis,kop,ktd,kca,
                                SUP1Cert,DISCert,OPCert,TDCert,
                                SUPKeys ,DISKeys,OPKeys,TDKeys)
    /\ session(SND,RCV,sessN2, sup2, dis, op, td,
                               ksup2,kdis,kop,ktd,kca,
                                SUP2Cert,DISCert,OPCert,TDCert,
                                SUPKeys ,DISKeys,OPKeys,TDKeys)
    /\ session(SND,RCV,sessN3, sup2, dis, op, td,
                               ksup2,kdis,kop,ktd,kca,
                                SUP2Cert,DISCert,OPCert,TDCert,
                                SUPKeys ,DISKeys,OPKeys,TDKeys)
end role
```

Figure 6: Specification of the environment and session role instances

In order to validate or falsify the security goals specified for the system, the model checker enumerates (essentially) all message exchanges possible for the given model applying the usual Dolev-Yao attacker model [6], which assumes an intruder capable of controlling the whole network traffic. He can intercept and take apart messages (as far as he knows the secret keys required to decrypt them) and learn their contents, construct new messages out of the material known to him, and send them to any party.

As stated in the previous section, the security properties checked for the SDS protocol are authenticity, integrity and confidentiality. These properties are specified in HLPSL by adding annotations, as shown in Figure 4. For instance, the annotation *witness(SSV,NextSSV,asset_hop_by_hop,Asset)* asserts that agent SSV has sent to agent NextSSV the value Asset, while the corresponding annotation *wrequest(SSV,PrevSSV',asset_hop_by_hop,Asset')* expresses that the agent SSV expects that the agent PrevSSV' has sent the value Asset'. If during the model checker run, a *wrequest* event is not matched by a previous *witness* event with the same identifier (in this case, *asset_hop_by_hop*) such that the values of sender, receiver and asset correspond, an attack has been found. The confidentiality goal is expressed by another annotation: secret(Asset',asset,{}). An attack against the confidentiality of the value Asset' is found if during the model checker run this value becomes part of the evolving intruder knowledge, which the model checker keeps track of. The overall system goals and system run are activated as follows:

```
goal
       weak_authentication_on asset_hop_by_hop
       weak_authentication_on asset_end_to_end
       secrecy_of asset
end goal
environment()
```

The AVISPA tool offers several model checkers as back-ends, which we have used to validate the SDS protocol, i.e. to check the specified security properties. We have performed the analysis on the protocol with and without encryption of messages, and in both cases, no attack has been found.

5 Conclusions and Future Work

We have proposed an architecture for a security-critical software distribution system, in particular the use of a generic component that is instantiated at different points of the SDS. For assessing the security of our design, we have composed two approaches, namely CC certification and formal analysis in the form of model checking. While the CC methodology is strong in systematically covering the secure implementation of a confined IT-product, it does not offer cost-efficient support for the assessment of a system composed of several instances of a generic component with a high assurance level. On the other hand, the automatic state exploration done by model checking is restricted to relatively small systems, like high-level security protocols, due to the exponential size of the state space of formal models, and dealing with implementation details requires the use of abstractions. Hence by assessing the implementation of the core component, the SSV, with the CC methodology and by formally analyzing the overall SDS protocol at high level, we combine the two methodologies according to their strengths, and gain substantial confidence in the overall security of the SDS. Apart from its role in the security assessment, the process of writing a formal model helps removing the inconsistencies and omissions usually present in a design specified in natural language. Moreover, having a formal model of the SDS protocol is valuable in itself, as it provides a highly precise documentation.

As further work, we plan to extend the formal model and include full configuration management with explicit installation instructions and configuration reports. We also have formally modeled aspects of the PKI underlying our software distribution system, in particular certificate initialization, and we plan to continue this work.

References

1. Armando, A., von Oheimb, D. et al. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In K. Etessami and S. K. Rajamani, editors, 17th Intl Conf. on Computer Aided Verification (CAV'05), Lecture Notes in Computer Science, Vol. 3576. Springer-Verlag, (2005)
2. AVISPA project homepage, 2005. http://www.avispa-project.org/
3. Blanchet B., From Secrecy to Authenticity in Security Protocols. In M. Hermenegildo and G. Puebla, editors, 9th Intl Static Analysis Symposium (SAS'02), Lecture Notes in Computer Science, Vol. 2477. Springer-Verlag, (2002) 342-359
4. Bodei C., Buchholtz M., Degano P., Nielson H. R., Nielson F.. Static validation of security protocols. Journal of Computer Security, 13(3) (2005) 347-390
5. Common Criteria. http://www.commoncriteriaportal.org/
6. Dolev, D., Yao, A.: On the Security of Public-Key Protocols. IEEE Transactions on Information Theory, 29(2) (1983) 198–208
7. Hartmann, P., Tappe, J., von Oheimb, D.: Asset Signer Verifier Protection Profile (2008) Available upon request
8. Robinson, R. Li, M., Lintelman, S., Sampigethaya, K., Poovendran, R, von Oheimb, D., Bußer, J., Cuellar, J: Electronic Distribution of Airplane Software and the Impact of Information Security on Airplane. In F. Saglietti, N. Oster, editors, 26th Intl. Conf. on Computer Safety, Reliability and Security (SAFECOMP), Lecture Notes in Computer Science, Vol. 4680. Springer-Verlag (2007) 28-39