SECURITY ARCHITECTURE AND FORMAL ANALYSIS OF AN AIRPLANE SOFTWARE DISTRIBUTION SYSTEM

David von Oheimb*, Monika Maidl*, Richard Robinson**
*** Siemens Corporate Technology, Otto-Hahn Ring 6, 80200 München, Germany,**
**** Boeing Phantom Works, P. O. Box 3707, MC 7L-70, Seattle, WA 98127-2207, USA**

Keywords: *Loadable Software, Information Assurance, Safety, System Architecture, Formal Methods*

Abstract

In order to make airplane maintenance procedures more efficient and thus save time and costs, the distribution of airplane embedded software will increasingly be performed electronically over networks. Since such software often implements vital functionality while on the other hand communication networks are inherently unreliable and insecure, protecting the software distribution process is critical, both from the safety and the security perspective. Therefore an airplane software distribution system must provide protection guarantees which need to be verified at a high assurance level.

In this paper, we introduce a software distribution system architecture with a generic core component, such that the overall software transport from the supplier to the airplane 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. Further, we outline the security evaluation of the proposed system according to the Common Criteria (CC) methodology and describe in detail our formal model and verification of its main functionality using the AVISPA tool. This mix of formal and systematic informal methods is a key factor in achieving high confidence in the security of the software distribution system at moderate cost.

1. Introduction

The use of electronic data distribution systems to support airplane maintenance processes introduces a need for information protection where physical protection was formerly employed. The FAA and other authorities mandate regulations and policies for the design and development of safety-critical airplane software [1] to ensure airworthiness. Information security so far has not been regarded as an issue in the context of onboard software, so airplane embedded systems typically check only for accidental modifications of software to be loaded, using Cyclic Redundancy Checks (CRC) or similar. The general trend towards ubiquitous networking motivates some reconsideration of this circumstance. Future airplane models may be expected to communicate with ground-based systems for electronic distribution of software (EDS) and on-line retrieval of airplane status data. While EDS may reduce overhead and improve efficiency and reliability of airplane manufacturing, operation and maintenance processes, these benefits come at the cost of exposing the airplane systems to potential attacks, in particular via data networks, aimed at corrupting or inhibiting the transmission of airplane assets. Flight safety may become dependent to some degree on the security of data transported via the data communication infrastructure. Acknowledging this issue, the FAA has started addressing security threats against ground-based support systems, insofar as they may affect flight safety [2]. In the future, critical aviation systems may require certification not only according to safety regulations, but also according to security regulations referring to the well-established Common Criteria (CC) for IT security evaluation [3]. Documents like the Information Assurance Technical Framework [4] by the U.S. NSA provide guidelines as to which Evaluation

Assurance Level (EAL) is deemed necessary and sufficient for addressing particular levels of threat and criticality. According to the CC, the highest EALs require formal modeling and analysis of the security policy and implementation mechanisms.

Electronic distribution of airplane software and data may be implemented by a system that we generically refer to as an Airplane Assets Distribution System (AADS) [10], [11]. The responsibility of the AADS for an asset begins when it takes over the asset from its producer, e.g. supplier or airplane, and ends when it delivers the asset at its destination, i.e., embedded systems such as a Line Replaceable Unit (LRU) in an airplane, or at the consumer of airplane-generated data. In [9], [11] we present our threat analysis with related security objectives and delineate functional and assurance requirements to achieve the objectives, in the spirit of the CC. The AADS must fulfil both safety- and business-oriented security objectives, in particular authenticity, integrity and availability of the assets including their identification and destination information. Furthermore, authorization and traceability for all critical activity including approval, storage and transmission are required. AADS security objectives are achieved with the help of digital signatures placed on assets and implemented via public-key cryptography.

1.1 Assuring Embedded Software Updates to Airplanes

In the past, distribution of embedded software packages to mobile platforms such as automotive vehicles, commercial passenger airplanes and military aircraft was accomplished by largely manual means. In a typical scenario, the owner or maintenance provider for an airplane would receive software updates on physical media, delivered via, e.g., a courier. A mechanic would be given a work order that entails installing software updates directly from physical media onto the airplane systems by means of a floppy disk drive or similar. In the near future, however, software updates to airplanes may be expected to make use of the Internet for delivering software to airplane owners, and, standard network infrastructure may be the medium used for the exchange of software and data with the airplanes themselves.

Because avionics software frequently implements vital functionality, for instance in the case of flight control software, the process responsible for distributing such software must implement guarantees that the integrity and authenticity of the software cannot be compromised.

1.2 Security Analysis and Certification

Various methods can be used to ensure security properties of networked systems. However, methods typically used in software development, including development assurance and exhaustive testing, do not work well for security properties due to the severe consequences of subtle errors or small oversights. After all, security properties must hold in the presence of attackers who actively try to exploit any weaknesses. A better approach to assess the security of systems is to work with a well-designed catalog of requirements and evaluation methods that is based on a broad range of experience. Certification according to Common Criteria, as discussed below, falls into this category. An even more thorough approach is to perform mathematical proofs or automatic exhaustive search as offered by formal methods.

For assessing the security of a system, i.e., assuring that the system implements countermeasures for all relevant security threats, the Common Criteria (CC) [3] 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 countermeasures are sufficient and correctly implemented. The first step of a CC evaluation 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 Protection Profile (PP)

which generically specifies the evaluation of a class of systems. We have defined such PPs for Airplane Asset Distribution Systems (AADS) and their core component [9].

The CC predefined Evaluation Assurance Levels (EALs) range from 1 to 7 and determine the rigor and depth of the analysis process. In general, the distribution of software controlling safety-critical devices calls for a high assurance level. 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 semi-formally verified design based on a formal security model, and EAL7 requires full formal verification.

In [11] we have justified the selection of 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 [4], EAL6 is recommended for safety-relevant threats, whereas EAL4 is shown sufficient for threats on airline business.

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, potentially by different partners.

On the other hand, component-wise certification faces the composition problem: Threats and vulnerabilities at system level may be different from the ones at component level, and whether the security objectives of the overall system are met as a consequence of the security properties of the individually certified components is a non-trivial 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.

As mentioned above, high assurance calls for formal analysis, to afford confidence grounded in mathematical precision and rigor. Ideally, formal methods are supported by tools ranging from automatic yet limited model checkers to powerful yet at most semi-automatic theorem provers. In recent years, several automatic 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 [6], [7] offers a tailored high-level specification language 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 security protocols, mainly of the Internet Engineering Task Force (IETF).

1.3 Our Contributions

Based on our experience with software distribution for avionics, automotive, and healthcare equipment, we define a generic system architecture for an Airplane Asset Distribution System (AADS) transporting software from supplier to airplane and data from airplane to its owner, possibly over insecure networks. We simplify the system design and its certification by defining a generic core component, the Asset Signer Verifier (ASV), instances of which are used at every node of the system. The overall AADS linking software suppliers with airplanes is essentially an interaction of several ASV instances.

We detail the security requirements necessary to ensure that threats against the software assets manipulated by the system are mitigated and countered. For a cost-efficient and still rigorous assessment of the distributed AADS, 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 determine the security requirements for the ASV and for the overall AADS and specify them in Protection Profiles like [9]. Assuming that the involved ASV components are certified, we use the AVISPA tool to formally specify and model check that the overall AADS protocol fulfils the security objectives at system level.

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

2. Architecture of the Software Distribution System

2.1 Threats and Security Objectives

In [11] we have presented a threat analysis and security objectives for an Airplane Asset Distribution System (AADS). The list of threats can be summarized as follows:

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

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

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

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

Disclosure. The attacker could try to capture restricted software contents to violate license agreements or to reengineer functionality, which may support other attacks.

The last mentioned threat, disclosure, was not included in [11] because it is not necessarily relevant in the AADS context. Yet in general, confidentiality might be important, in particular to protect intellectual property.

From the threats described above we derive a set of security objectives that must be met by the AADS. The threats are formulated for the protection of Loadable Software Airplane Parts (LSAP), but must hold analogously for airplane generated data.

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

Integrity. For every software item accepted at an airplane, 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 AADS (at the supplier) until reaching the airplane.

Correct Destination. An airplane must accept and receive only software items for which it is the true destination intended by the airplane's operator (i.e. the airline).

Correct Version. An airplane must accept software items only in the latest version approved by its airline.

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 transmission step the integrity of an asset is preserved.

2.2 AADS Architecture

On the way from the software *supplier* to the airplane, software items may be handled at intermediate entities: airplane *manufacturers* or OEMs may receive the software items from the supplier, and send it to the *airplane operator* (usually an airline), who bears responsibility for the safe operation of the *airplane*, and has the authorization to send software there. So the software distribution process consists of several hops, and the AADS 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 airplane operator coincides with the manufacturer or even with the supplier.

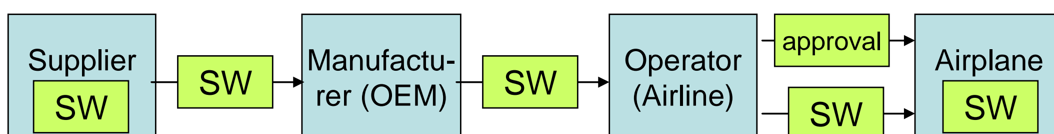

Figure 1: The overall Airplane Assets Distribution System.

For every transportation step, the software item must be protected against the threats listed above. Digital signatures and encryption using public key technology are the fundamental security mechanisms used to implement the protection within the AADS. Signatures are generated by applying the private key of the sender to the contents, or rather to the hash value (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.

Intermediaries might simply store and forward the software, or they may perform some local processing, such as acceptance tests, adding owner specific license keys, and setting airplane specific software parameters. In any case, the intermediary must check the signature of the previous entity and may add a new signature.

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

The airplane verifies the integrity and authenticity of the software item using the signature of the airline and checks, using the approval statement of the airline, whether itself 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 issues installation orders in the form of a work order on paper, to be executed by a mechanic. For some AADS, administration of the airplane may be automatic under remote control of the airline.

We structure the AADS into several instances of a signature application component called Asset Signer Verifier (ASV), 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 ASV instances can be developed and certified independently, or one and the same ASV product can be used at all nodes.

2.3 ASV: the AADS Core Component

Each node in the above distribution chain runs an instance of the ASV, i.e. the AADS core component. Figure 2 shows the ASV in its environment including the flow of software. The ASV instances are used for:

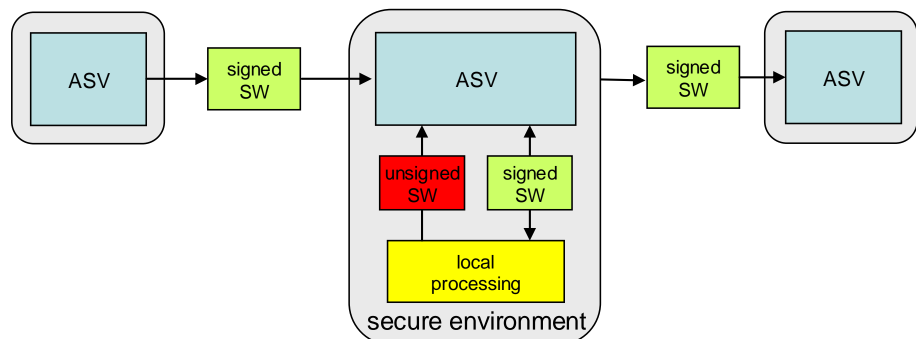

Figure 2: The generic Asset Signer Verifier and its environment.

Introducing unsigned software into the AADS by digitally signing and optionally encrypting it

and making it available for other ASV instances.
Verifying the signature on software received from other ASV 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 ASV instances.
Delivering software out of the AADS after successfully verifying it.

Introduction of software into the AADS typically takes place at the supplier, yet may take place also at intermediate entities, while software delivery happens at the airplane only. All ASV instances except at a supplier verify incoming software. Adding a new signature will be done usually at ASV instances located at manufacturers and airlines after some local processing of the software, such as adding metadata or by performing a quality inspection. Such processing is performed within the local environment of the respective ASV.

3. Security Assessment of the AADS

3.1 Assumptions on the Operational Environment

Not all assurance issues related to software distribution can be covered by the security assessment of the AADS at reasonable costs. For example the reliability of the Public Key Infrastructure (PKI) [5], 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 explicit assumptions on the operational environment of the assessed system. The assumptions on the ASV environment are the following:
ASV protection. The ASV instances are protected against direct manipulation and misuse. The ASVs run on a hardened operating system (OS), where 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 ASV 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 ASV, 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 ASVs is trustworthy and properly managed. Revocation information is issued regularly and immediately after revocation of a signing key.
Airplane configuration enforcement. The local environment of the airplane ASV checks whether the installation of received software items is authorized by an approval statement of the airplane operator. Depending on the system design, this assumption can be relaxed, e.g. the ASV itself may perform such checks.

3.2 Certification of the ASV

The ASV is a component for which a Security Target may be produced according to the CC methodology. The document [9] is a Protection Profile (PP) for the ASV in the special case of airplane software distribution; however this PP could easily be adapted to handle very general software distribution scenarios. The PP specifies the security objectives of integrity and authenticity and – if required – of confidentiality at the component level, and hence after successful CC evaluation of the ASV instances, there will be sufficient evidence that the security mechanisms of the ASVs achieve these security objectives under the assumptions on the ASV 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 below.

3.3 The Protocol for End-to-End Software Distribution

In order to assess the correctness of the AADS at system level, we consider the interaction

between the ASV 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'. In Figure 3, the different nodes are abbreviated as follows:

`SUP`	software supplier,
`MFG`	airplane manufacturer,
`OP`	airplane operator (airline),
`AP`	airplane, and
`CA`	certificate authority.

For each node N, the associated public key is denoted by `KN`, and the private key is denoted by `inv(KN)`.

```
1. SUP - {Asset.{h(Asset).MFG}_inv(KSUP).CertSUP}_KMFG -> MFG

2. MFG - {Asset.{h(Asset).MFG}_inv(KSUP).CertSUP
                .{h(Asset).OP }_inv(KMFG).CertMFG}_KOP   -> OP

3. OP  - {Asset.{h(Asset).MFG}_inv(KSUP).CertSUP
                .{h(Asset).OP }_inv(KMFG).CertMFG
                .{h(Asset).AP }_inv(KOP ).CertOP}_KAP    -> AP
```

Figure 3: End-to-end software distribution protocol

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

The meaning of the constructs used in Alice-Bob notation is as follows:

`A - M -> B`	message `M` sent from `A` to `B`
`Asset`	a software item including its identity
`M.N`	the concatenated contents of M and N
`h(M)`	the hash value of content `M`
`{M}_inv(K)`	content `M` signed with private key `K`
`{M}_K`	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 AADS protocol, signatures are applied in parallel: every ASV keeps the old signatures and adds its own. However, each ASV 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 ASV has a trust relationship with all previous nodes. For example, the airplane 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 airplane 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 airplane in the asset signature applied by the operator.

3.4 Security Properties

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

(1) assets accepted by the airplane have indeed been sent by the supplier,
(2) assets accepted by the airplane have not been modified during transport,
(3) asset authenticity and integrity are maintained hop-by-hop, i.e. from any ASV instance to the next, in particular between the operator and airplane, and

(4) assets remain secret among the ASVs.

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

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

4. Formal Analysis of the AADS Protocol

The Alice-Bob notation, showing only message exchanges, is not detailed and precise enough for 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 [6], 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.

4.1 Modeling an ASV

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

Figure 4 shows the header declaration for the ASV 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 airplane 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`.

```
role softwareSignerVerifier(
        SND,RCV: channel(dy),
        SessN: nat,     % session number, needed just for technical reasons
        SUP,AP: agent,  % supplier and airplane, 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.
        ASV, NextASV: agent,
KASV,KNextASV: public_key,   % public key of this ASV and
                              % the one to which it sends messages
        CertASV: {agent.public_key
                  }_inv(public_key),  % certificate for the private key inv(KASV)
        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,KprevASV: public_key,
       Cert: {agent.public_key}_inv(public_key),
       PrevASV: agent

init
       State := 0
```

Figure 4: Header and local variables of the ASV role

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

```
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'}_KASV)
      /\ PrevSigs' =
             X'.({h(Asset').ASV.SessN}_inv(KprevASV').Cert')
      /\ Cert' = {PrevASV'.KprevASV'}_inv(KCA')
      /\ in(KCA',KCASet) % check if CA is in the accepted CA set
 =|> State':= 1
      /\ wrequest(ASV,PrevASV',asset_hop_by_hop,Asset')

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

send.
      State  = 1 /\ Export = true /\ RCV(start)
 =|> State':= 2
      /\ SND({Asset.PrevSigs.({h(Asset).NextASV.SessN}_inv(KASV)
              .CertASV)}_KNextASV)
      /\ witness(ASV,NextASV,asset_hop_by_hop,Asset)
      /\ witness(SUP,AP     ,asset_end_to_end,Asset)

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

Figure 5: Transitions of the ASV role

There are five transition rules, presented in Figure 5. 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 ASV does with the received asset: either forward it in signed form to the next one, or consume it.

We explain the second transition in more detail. A transition is divided by the `=|>` symbol into a condition part in which a message may be received and checked, and an action part in which a message may be sent. Subcomponents of a transition are conjoined by the `/\` symbol. 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'}_KASV` means that a message that must be encrypted with the key `KASV` 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').ASV.SessN}_inv(KprevASV').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 `ASV`. The public key with which the signature can be validated is stored in `KprevASV`. Next the certificate is validated: it has to contain the identity of `KprevASV`, and has to be signed by a CA whose public key is contained in `KCASet`.

```
role session(SND,RCV: channel(dy),SessN: nat,
     SUP,MFG,OP,AP: agent,
     KSUP,KMFG,KOP,KAP,KCA: public_key,
     SUPCert,MFGCert,OPCert,APCert:
{agent.public_key}_inv(public_key),
     SUPKeySet,MFGKeySet,OPKeySet,APKeySet: public_key set)
def=

composition
   softwareSignerVerifier(SND,RCV,SessN,SUP,AP,false,true,
                    SUP,MFG,KSUP,KMFG,SUPCert,{KCA},SUPKeys)
/\ softwareSignerVerifier(SND,RCV,SessN,SUP,AP,true,true,
                    MFG,OP  ,KMFG,KOP  ,MFGCert,{KCA},MFGKeys)
/\ softwareSignerVerifier(SND,RCV,SessN,SUP,AP,true,true,
                    OP  ,AP  ,KOP  ,KAP  ,OPCert  ,{KCA},OPKeys)
/\ softwareSignerVerifier(SND,RCV,SessN,SUP,AP,true,false,
                    AP,none,KAP,knone,APCert  ,{}    ,APKeys)
end role
```

Figure 6: Specification of the session role

As a by-product of these checks, the ASV learns the identity of the sender, stored in the variable `PrevASV`. As explained above, the ASV 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'`.

4.2 Modeling the AADS

Figure 6 shows a composed role called `session`, which ties together the instantiations of the ASV needed for the end-to-end transport of one asset. Each instantiated ASV is configured with its own parameters. For instance, the eighth parameter is the name of the agent, i.e. SUP in the first instantiation, MFG in the second, and so on.

The last part of the model, given in Figure 7, 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`, manufacturer `MFG`, operator op, and airplane `AP`.

```
role environment() def=

local
      SND,RCV: channel(dy),
      SUP1Cert,SUP2Cert,MFGCert,OPCert,APCert:
                              {agent.public_key}_inv(public_key),
      SUPKeys,MFGKeys,OPKeys,APKeys: public_key set

const
       sessN1,sessN2,sessN3: nat,
        sup1, sup2, sup3, MFG, op, AP     : agent,
       ksup1,ksup2,ksup3,kMFG,kop,kAP,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
        MFGCert := {MFG  .kMFG  }_inv(kca   ) /\
         OPCert := {op   .kop   }_inv(kop   ) /\ % self-signed
         APCert := {AP   .kAP   }_inv(kAP   ) /\ % self-signed, unused
       SUPKeys   := {}              /\          % unused
       MFGKeys   := {ksup2, ksup3} /\          % ksup3 is unused
       OPKeys    := {}              /\
       APKeys    := {kop}

intruder_knowledge = { sup1, sup2, sup3, MFG, op, AP,
                       ksup1,ksup2,ksup3,kMFG,kop,kAP,kca}

composition
       session(SND,RCV,sessN1, sup1, MFG, op, AP,
                               ksup1,kMFG,kop,kAP,kca,
                               SUP1Cert,MFGCert,OPCert,APCert,
                               SUPKeys ,MFGKeys,OPKeys,APKeys)
    /\ session(SND,RCV,sessN2, sup2, MFG, op, AP,
                               ksup2,kMFG,kop,kAP,kca,
                               SUP2Cert,MFGCert,OPCert,APCert,
                               SUPKeys ,MFGKeys,OPKeys,APKeys)
    /\ session(SND,RCV,sessN3, sup2, MFG, op, AP,
                               ksup2,kMFG,kop,kAP,kca,
                               SUP2Cert,MFGCert,OPCert,APCert,
                               SUPKeys ,MFGKeys,OPKeys,APKeys)
end role
```

Figure 7: Specification of the environment and session role instances

4.3 Modelchecking the AADS

In order to validate or falsify the security goals, the model checker enumerates (essentially) all message exchanges possible for the system model applying the usual Dolev-Yao attacker model [8], 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 to be checked for the AADS protocol are authenticity, integrity and confidentiality. These properties are specified in HLPSL by adding annotations, as shown in Figure 5. For instance, the annotation `witness(ASV,NextASV,asset_hop_by_hop,Asset)` asserts that agent `ASV` has sent to agent `NextASV` the value `Asset`, while the corresponding annotation `wrequest(ASV,PrevASV',asset_hop_by_hop,Asset')` expresses that the agent ASV expects that the agent `PrevASV'` 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 shown in Figure 8.

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

Figure 8: Overall system goals

The AVISPA tool offers several model checkers as back-ends, which we have used to validate the AADS 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 AADS. For assessing the security of our design, we have combined 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 and abstract system models, like high-level security protocols, due to the exponential size of the state space of the models, and dealing with implementation details requires the use of abstractions. Hence by assessing the implementation of the core component, the ASV, with the CC methodology and by formally analyzing the overall AADS protocol at high level, we combine the two methodologies according to their strengths, and gain substantial confidence in the overall security of the AADS. 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 AADS protocol is valuable in itself, as it provides 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 [5] underlying our software distribution system, in particular certificate initialization, and we plan to continue this work.

References

[1] DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (RTCA), 1992.

[2] Federal Aviation Administration, 14 CFR Part 25, Special Conditions: Boeing Model 787–8 Airplane; Systems and Data Networks Security—Protection of Airplane Systems and Data Networks From Unauthorized External Access, [Docket No. NM365 Special Conditions No. 25–07–02–SC], Federal Register, Vol. 72, No. 72, 2007.

[3] Common Criteria for IT Security Evaluation. http://www.commoncriteriaportal.org/.

[4] Information Assurance Technical Framework, Release 3.1. US National Security Agency, 2002. http://www.iatf.net/framework_docs/version-3_1/

[5] Adams, C., Lloyd, S.: Understanding PKI: Concepts, Standards, and Deployment Considerations. Addison-Wesley, 2nd edition, 2003.

[6] 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.

[7] AVISPA project homepage 2005. http://www.avispa-project.org/

[8] Dolev, D., Yao, A.: On the Security of Public-Key Protocols. IEEE Transactions on Information Theory, 29(2), pp 198–208, 1983.

[9] Hartmann, P., Tappe, J., von Oheimb, D.: Asset Signer Verifier Protection Profile, 2008. Available upon request.

[10] Robinson, R., Li, M., Lintelman, S., Sampigethaya, K., Poovendran, R., von Oheimb, D., Bußer, J.: Impact of Public Key Enabled Applications on the Operation and Maintenance of Commercial Airplanes. AIAA Aviation Technology, Integration and Operations (ATIO) conference, 2007.

[11] 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, pp 28-39, 2007.

Copyright Statement

The authors confirm that they, and/or their company or institution, hold copyright on all of the original material included in their paper. They also confirm they have obtained permission, from the copyright holder of any third party material included in their paper, to publish it as part of their paper. Siemens and the Boeing Company, as copyright holders of this paper, grant full permission for the publication and distribution of their paper as part of the ICAS2008 proceedings or as individual off-prints from the proceedings.