Automatic Generation of Certifiable Space Communication Software

Johann Schumann and Ewen Denney
RIACS / NASA Ames, Moffett Field, CA 94035
{schumann|edenney}@email.arc.nasa.gov

Abstract— Reliable, secure and effective communication between a spacecraft and the ground station, or between multiple spacecraft is central to all space missions. Smooth control of spacecraft and the downlink of mission and science data rely heavily on reliable means of communication. Moreover, heightened needs for operations security in recent years add complexity to communication system requirements. The communication system, therefore, is a highly safety and mission critical component. A single successful malicious attack or a flaw in the code can have serious consequences that put the mission or even human life at risk.

We are integrating and adapting a set of existing tools in order to provide a unified end-to-end approach to the design, analysis, implementation, and certification of space communication software. Our tools are based upon rigorous logical and mathematical foundations, and are capable of automatically generating high-quality communication software from a high-level model. Automatic, tamper-proof formal certification techniques are used to provide explicit guarantees about important reliability and security properties and the absence of implementation errors.

Table of Contents

1. Introduction

Reliable, secure and effective communication between a spacecraft and ground station is central to all space missions. An optimal design and implementation of the communication subsystems is an important prerequisite for a successful mission, since control of the spacecraft and the effective downlink of mission or science data rely on reliable communication. This is especially the case for deep-space missions, where bandwidth is at a premium, and elaborate and special purpose communication protocols are used. Heightened needs for operations security also add complexity to the communication system requirements. A malicious attack or a simple flaw in the code can put the mission or even human life at risk.

Although secure communication protocols are in wide use, history has shown that many errors and vulnerabilities do exist and have been actively exploited. Such security flaws can be introduced (or fail to be detected) during all stages of the software development cycle. Space-specific requirements, such as low bandwidth, high latency, or constrained computational capabilities, pose additional severe challenges for developing communication software.

It is our contention that reliable and secure communication software can best be developed with a unified approach throughout the entire software life cycle. We have developed a set of tools that facilitate a unified end-to-end approach to the design, analysis, implementation, and certification of communication software. Our tools are based upon rigorous logical and mathematical foundations, and are capable of automatically generating high quality communication (protocol execution) software from a high-level model using certifiable program synthesis. Moreover, automatic, tamper-proof certification provides explicit guarantees about important reliability and security properties and the absence of implementation and design errors. These properties include absence of buffer-overflow errors, guarantees for variable initialization and correct usage (i.e., all required data are packed/unpacked and transmitted in the right way), and the correct use of encryption algorithms[1]. Security authentication properties are expressed using the well-known BAN logic [3]. Although this logic is relatively weak, it is amenable to automatic processing and, as our tools can produce readable proofs, allows protocol designers to quickly find flaws in protocols.

The remainder of this paper is structured as follows. In Section 2, we discuss important requirements and issues with secure space communications software and introduce a simple demonstration example. Section 3 focuses

1-4244-0525-4/07/$20.00/

[1] We are not, however, analyzing the strength or correctness of encryption algorithms themselves.

on the development process for safety critical software and its augmentations to address security issues. In Section 4, we describe our tools for modeling, analysis, automated code generation, and certification. Section 5 describes related work and Section 6 discusses future work and concludes.

2. Space Communication Software

Despite the fact that many security protocols exist and are used in everyday life, new or customized protocols are continually being designed and implemented to suit specific needs. In space applications, for example, a low bandwidth and high latency time (e.g., 20 minutes to Mars) as well as strongly limited on-board computing power, poses specific constraints on communication software. Other applications, e.g., for Next Generation Air Traffic Control or sensor networks [13], pose different, but still severe constraints on protocol design (or customization) and implementation.

History has shown that errors and vulnerabilities can and actually do occur in communication software, even if it is based upon simple protocols. More elaborate protocols or specialized protocols not only require substantial effort in design, implementation, and testing (a major cost driver), but can also introduce errors and vulnerabilities during the development. The following examples list a few reasons for security flaws and protocol failures:

- *misunderstanding of protocol requirements*: the wrong protocol may be used for a specific application, or specific requirements might be violated (e.g., the existence of a trusted key server).
- *weak cryptography*: often, cryptographic algorithms are used that are much weaker than originally intended. Thus, attackers can hack or reverse engineer the code to open up vulnerabilities. Sometimes, proprietary encoding schemas are much weaker than published and proven protocols and algorithms.
- *coding errors* are a major source of vulnerabilities. Most security warnings regarding software like the Windows OS or Internet browsers have been caused by implementation errors like buffer overflow, uninitialized variables, deadlocks, etc.
- *errors in protocol optimization*: optimizing a complex, layered protocol toward maximal performance can lead to hard-to-detect errors and security vulnerabilities.
- *errors during testing and deployment*: a bad or incomplete selection of test cases would not exhibit flaws in the protocol. Incorrect testing and deployment procedures can also lead to serious problems.

All these issues must be addressed with respect to the application specific requirements (e.g., for bandwidth, quality of service, computing requirements) to yield safe, reliable, and re-usable communications software that can be developed at an affordable price.

For this paper, let us consider a highly simplified example: a secure download of telemetry data from a satellite to a ground station (Figure 1). Upon request from the ground station to download telemetry data, a secure channel (with a session key K_{sg}) must be established, using a (trusted) security server on the ground. For this purpose, the well-known Yahalom protocol will be used. In the following sections, we will use this example to illustrate our modeling, analysis, and synthesis tools. Although our tools are capable of addressing data and control-flow issues (e.g., packing of data into buffers), in this paper we mainly focus on the aspect of establishing a secure authenticated channel with session key K_{sg}.

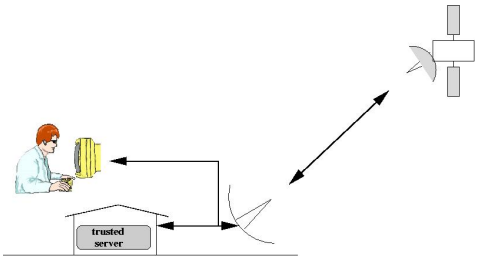

Figure 1. Simplified scenario for a data downlink between satellite, ground station and trusted key server.

3. Software Development Process

Communication software is developed using a similar process to that of traditional software, with phases for requirements, specification, coding, testing, and deployment. High-quality software requires the software developers to accommodate the issues of verification and validation (V&V), which must be carried out in each individual step of the software life cycle (see Figure 2). *Verification* tasks demonstrate the correctness between subsequent stages of the life cycle (e.g., proving that the implementation correctly conforms to the specification), whereas *validation* aims to ensure consistency (horizontal arrows) of implemented artifacts (on the right leg of the V) with the requirements and design. Because a manual software development process can introduce errors at each stage, V&V is a very important, but difficult and time-consuming task.

The main difference for secure communication software is that, in addition to traditional functional and safety requirements, *security* properties (e.g., about encryption, secrecy, authentication) need to be defined and checked during V&V.

In our work, we provide a tool-supported approach that will substantially reduce V&V effort by automating the transition between the individual software design and development stages. Because our framework is based on rigorous formal methods, guarantees can be provided that important and specific classes of errors (e.g., buffer

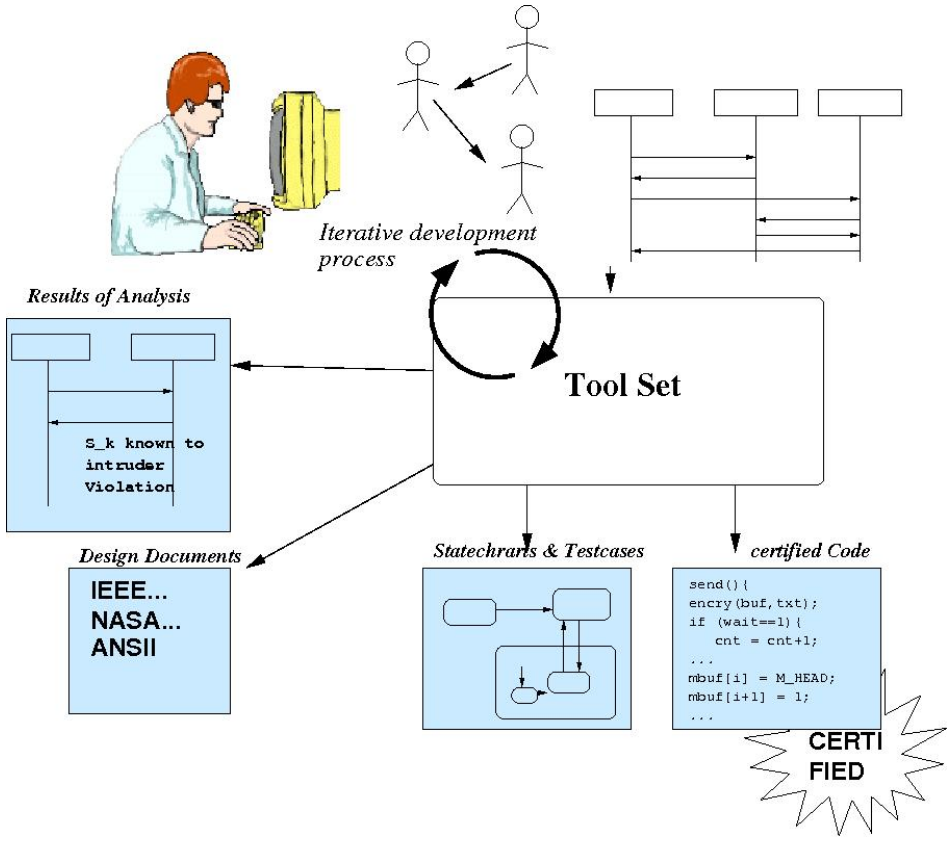

Figure 3. Tool workflow (see Figure 7 for tool architecture)

overrun, deadlocks) are not introduced. The approach is illustrated in Figure 3. Starting from a specification of the protocol along with its requirements and constraints, tools are used to analyze the protocol, generate test cases, produce reliable code with certificates and generate the design documentation. Results of the analyses can be fed back into the original design and specification, supporting an *iterative* approach to software generation.

4. The Individual Stages

Modeling

During the development process, a model of the communication software is developed that captures all important details and architectural considerations. This high-level model needs to reflect the requirements.

We are using UML sequence diagrams (or scenarios) to model the behavior of the protocol. Given a set of participants (in our example, the satellite or the ground station), a sequence diagram defines the temporal sequence of communications between the participants. In order to formalize a deeper semantic content, we augment the sequence diagrams with formal logical annotations.

Figures 4 and 5 show a (simplified) example of protocol model that could be used to securely communicate between a satellite and the ground station (Figure 1). In the scenario in Figure 4, the ground station requests some telemetry data from the satellite by sending a request `send_tel(`N`)`, where N is a nonce (e.g., a time stamp). The receiver on the satellite requests the acquisition of the sensor data, detects that this is a new communication request, and thus initiates run of the Yahalom authentication protocol with the key server and the ground station. The final result is that the satellite now has a new ("fresh") session key K_{sg} for communication between this satellite and the ground station. Upon receipt of this key, the receiver requests the data handler to obtain the data and encrypt it with the session key K_{sg}. Finally, the data are packed as `p(e_data)` and down-linked to the ground station.

For our example, we use the following notation: K_{sk} is the shared key between satellite and the key server, K_{sg} is the shared key between satellite and ground station, and K_{gk} is the shared key between ground station and key server. Messages 2, 3, and 4 of the Yahalom protocol are relatively complex and contain various information about nonces, the shared keys, and identification information of satellite and ground station. Table 2 lists all protocol messages.

A second scenario (Figure 5) shows a situation, where

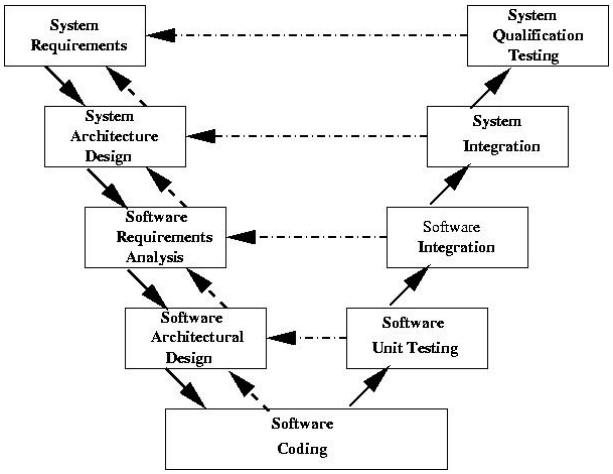

Figure 2. V-shaped software development and V&V process. Dependencies between the development phases are indicated by solid arrows. Dashed arrows concern verification activities; dotted lines validation activities.

the receiver on the satellite detects that the current session key is still valid (the nonce, sent from the ground station, is still sufficiently "fresh". In this case, no new authentication needs to take place and the data can be directly encrypted with K and sent to the ground station. A full model of the communication protocol can consist of many scenarios. In particular, failure scenarios (e.g., wrong messages, time-outs) are necessary to define all details of the communication mechanism. In this paper, we only present two nominal scenarios.

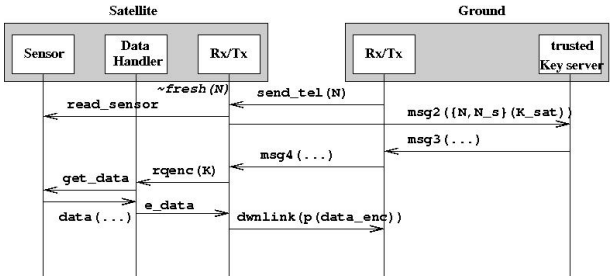

Figure 4. Simplified example of protocol specification.

Logic expressions in OCL (UML's Object Constraint Language) or other formalisms can be used to augment the sequence diagrams. With those mechanisms, information about system state (e.g., if a secure channel has been established), constraints, and assumptions can be expressed. For our tools, annotations are formulated either OCL or, in the case of the security properties, the BAN logic, which will be described in the next section.

Figure 6 shows simplified OCL annotations for the transmitter/receiver object of the satellite. Here, we have a boolean variable `key_recd`, which reflects a part of the state of the satellite communications subsystem. This

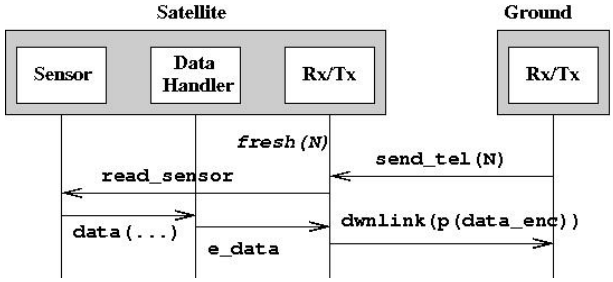

Figure 5. Communication scenario using an existing shared key K.

```
<features>
<type> SAT_RXTX
      <attributes>
            key_is_valid : Boolean;
            key_recd     : Boolean;
      </attributes>
      <operations>
            read_sensor(...) : Void;
            msg2(...)        : Void;
            ...
      </operations>
      ...
<invariants>
context SAT_RXTX:: msg4(...) : Void
      post: key_recd = true;

context SAT_RXTX:: rqenc(...) : Void
      pre: key_recd = true;

context SAT_RXTX:: dwnlink_tel(...)...
      pre: key_recd = true;
      post: key_recd = false;
...
```

Figure 6. OCL annotations (excerpt).

variable is set to true, if and only if a valid session key has been obtained. The other boolean state variable `key_is_valid` keeps information about the validity of the session key. These state variables can be used as guards in the software and also guide our code generation tools (see below).

Protocol Analysis

Analyzing the security properties of a protocol (specified as a sequence of messages between the participants) can be a complicated and time-consuming effort. During analysis, properties about authentication, secrecy, confidentiality, availability, etc. are proven in a formal way, or attack scenarios are generated if a security property can be violated. For this task, a large body of approaches and research tools exist (see Section 5) that are based upon simulation, model checking, or theorem proving.

In an iterative software process, it is important that major requirements and design errors are captured as early as possible. Our protocol analysis tool is designed for this purpose. For formalization of the security proper-

ties, it uses the well-known BAN (Burrows, Abadi, Needham) [3] logic. Although this logic is relatively weak (for example, secrecy cannot expressed within this logic), the formalism is very intuitive to the user and protocol analysis with the BAN logic can be fully automated.

The BAN logic is a multi-sorted modal logic of belief. Its statements describe what the communication partners believe ($\models$), what they receive (see, $\triangleleft$), which information they send (said, $\sim$), and which information they control ($|\Rightarrow$). With additional operators, expressions about the freshness of nonces (typically timestamps, #), communication methods, and ways of encryption can be formulated and reasoning can be performed. For a detailed description of the BAN logic, see [3].

Within our framework of protocol synthesis, the protocol (or, actually, an extracted abstracted version thereof) as well as additional assumptions form the basis of the analysis. Table 3 lists the security assumptions. So, for example, satellite and ground believe that they can communicate with their shared keys (e.g., K_sg), that the produced timestamps are recent, and that the key server is producing reliable keys.

As a run of the protocol is executed, various security properties are valid at the various stages of the protocol. If a successful communication has been established, the satellite and the ground station believe that they communicate securely with the session key K. Table 1 lists all 12 security properties which have to be proven in order to show BAN-security of this protocol. The formalization of the assumptions and security properties have been adapted from [3], but they are similar for each protocol. So our tool can provide a small default set of security properties.

For the actual security analysis, we are using the tool PIL-SETHEO [16]. It takes the BAN representation of the protocol as extracted from the model and the assumptions and tries to automatically prove the given security properties. If the system succeeds, it will produce a human readable proof in BAN logic to document the validity of the security property.

For the example protocol, the set of 12 security properties can be proven by PIL-SETHEO within a few seconds (Table 1). If, however, the protocol designer mistakenly sends the message msg3 from the key server directly to the satellite instead to the ground station[2], the modified protocol will not work properly. Our PIL analysis tool can detect this problem immediately as it now fails to prove those security properties that are associated with the message msg3 (Table 1).

[2] Such a communication pattern seems to be reasonable, since the request to the key server originally came from the satellite.

#	Step	BAN	P_{OK}	P_{bad}
1	3	$G\models K_{gs}$	•	×
2	3	$G\models K\models \#K_{gs}$	•	×
3	3	$G\models S\models N_g$	•	×
4	4	$S\models K\sim K_{gs}$	•	•
5	4	$S\models G\sim K\models \#K_{gs}$	•	•
6	4	$S\models G\models K\models \#K_{gs}$	•	•
7	4	$S\models K\models freshK_{gs}$	•	•
8	4	$S\models \#K_{gs}$	•	×
9	4	$S\models K_{gs}$	•	×
10	4	$G\models K_{gs}$	•	×
11	4	$G\models S\models N_g$	•	×
12	4	$S\models G\models K_{gs}$	•	•

Table 1. PIL-SETHEO analysis results for the correct protocol P_{OK} and the "broken" protocol P_{bad}. A • means that the property could be proven, × indicates a failure. The first column is the property number, the second column defines after which protocol step (i.e., after which protocol message) the property must hold.

Protocol Software Synthesis

Up to this stage, all analysis steps have been performed on a high-level specification of the security protocol. This specification now needs to be implemented as real code. As discussed earlier, this coding phase is very error-prone. We therefore use automatic code generation tools to produce reliable target code from our specifications. Figure 7 shows how the various tools for code generation work together. From our set of annotated sequence diagrams, we first generate a set of (hierarchical) statecharts. This tool, which is described in [22], [17], has been used to generate control code for communication modules within NASA's CTAS advanced air traffic control system (advisory system) [21] and for various agent-based systems [17].

The tool merges all sequence diagrams, automatically recognizes loops, and detects and reports various inconsistencies. The OCL annotations of messages in the sequence diagrams enable the tool to automatically produce compact, hierarchical statecharts. Figure 8 shows the statechart for the component "Satellite RxTx" as it is generated from the sequence diagrams (Figures 4, 5).

The next step is to translate the statechart into executable code. For the actual generation of the executable code, we use a tool based upon the GReAT [11] framework, which is being developed in collaboration with the Vanderbilt university. This tool produces efficient C code from Stateflow diagrams. Stateflow is a part of the MathWorks toolset[3]. The generated statecharts (Figure 8) can be directly translated into Stateflow diagrams,

[3] http://www.mathworks.com

albeit manually, at this point. Future work will enable automatic translation of the generated statecharts.

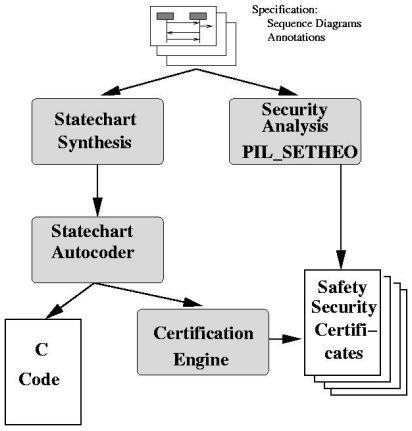

Figure 7. Automatic code generation: tools (shaded) and artifacts.

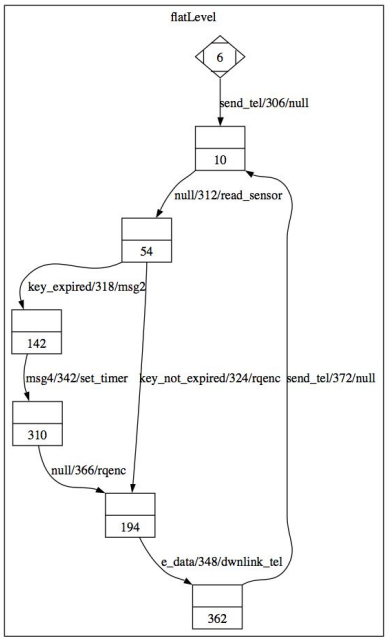

Figure 8. Automatically generated statechart.

Automatic Certification of Protocol Software

Once the security software has been synthesized, it still needs to be verified before it can be deployed. This could be avoided if the synthesis tool was itself verified ("qualified"), but a protocol synthesis tool is an extremely large and complex piece of software itself, so its formal verification is not feasible. In order to overcome this problem, we use the *product certification* approach, in which checks are performed on each and every generated program by a certification engine (Figure 7) rather than on the generator itself. We will adapt existing technology of the AutoBayes/AutoFilter system [6], [5], and focus on *safety properties*, which are generally accepted as important for quality assurance and are used in code reviews of high-assurance software.

The safety policies checked by our system describe either language-specific or domain-specific properties which a safe program must satisfy. A typical example of a language-specific property (C/C++) is array-bounds safety; violations can lead to serious flaws, as many buffer-overrun attacks have shown. Checks for consistency of physical units or symmetry of matrices are specifically tailored to the application domain and provide additional assurance.

The system uses program verification techniques based on Hoare logic and processes logical pre- and post-conditions statement by statement to produce proof obligations. These are then discharged by an automatic theorem prover. However, such techniques require additional program annotations (usually loop invariants) which makes their application very hard in practice. We overcome this obstacle by extending our code generator to synthesize *simultaneously* the code *and* all required annotations. This enables a fully automatic certification which is transparent to the user and produces machine-readable certificates showing that the generated code does not violate the given safety policies.

5. Related Work

Modeling and formal analysis of security protocols is a wide field with a long tradition. One of the most well-known formalisms is the BAN logic [3] and its many successors. UML has been used to specify security protocols; a security extension to UML, UMLsec [7] has been developed. In UMLsec, security properties can be modeled. Specific UML patterns can be used to construct and refine communication models that exhibit specific security properties. After model development, code generators for UML tools can be used to produce executable code.

Tools and methods for the analysis of protocols have been developed using a number of different paradigms. Security and vulnerability of protocols can be analyzed by simulation or model checking (e.g., FDR (Failures-Divergences Refinement) [8] or by modeling as finite state machines [9]). Other approaches use automated ([15], [18], [19]) or interactive ([12]) theorem provers. A number of vulnerabilities can be detected with these tools, but they usually require a specific input format and the results they produce are specific for each tool.

In contrast to approaches based on post hoc analysis and verification on an existing protocol and its implementation, there is a number of approaches toward the automatic synthesis of security protocols from specifications.

For example, the Degas tool[4], extracts UML protocol specifications and generates an ML program from them. The main application of this tool is consistency analysis using type-checking. The authors of [1] also extract protocol specifications and convert them into logic for subsequent analysis, while [20] has developed a tool to convert protocol specifications (in CAPSL [10], [2]) into executable Java.

The main goal of the approaches described in [14], [23], and [4] is to automatically generate secure and efficient protocols from property-based specifications. This task is quite challenging because the tools have to "invent" the appropriate steps of the protocol. In our approach, on the other hand, the user will give, as input, a detailed specification of the protocol, so that well known (and fully analyzed) protocols (or protocol variants) can be used, and a simpler and correct synthesis approach can be used.

6. Conclusions

In this paper we have outlined ongoing work on the development of a set of tools for the design, analysis, and automatic verification of security protocols. The aim of our approach is to provide an integrated toolset that will provide a unified framework for all stages of protocol design and implementation.

There are a number of additional functionalities which could naturally be combined with this toolset, such as the generation of marshalling code (packing/unpacking, etc.), and protocol optimization. Further techniques for achieving assurance could also be incorporated, such as the automated generation of test cases, which would complement the guarantees given by the proofs.

Appendix

1. Protocol Specification

The Yahalom protocol consists of four messages between two participants (ground G and satellite S) and a trusted key server K. Table 2 gives the (idealized) specification of the 4 messages (similar to [3]). An expression $\{X\}(k)$ means that data X are encrypted using the key k.

This protocol requires a total of 14 assumptions shown in Table 3. These assumptions are straight-forward and can be taken directly from the protocol specification or the literature [3]. These assumptions concern the freshness of the keys and nonces which the participants issue (1-3), the use of the keys as shared keys (4-8), the trust in the key server K that it can produce ($|\Rightarrow$) the appropriate keys and timestamps (9-13), and the non-distribution of the satellite's timestamps (14).

[4] `http://www.imm.dtu.dk/cs_LySa/`

#	From	To	Message
1	G	S	$\texttt{send\_tel}(N_g)$
2	S	K	$\{N_g, N_s\}(K_{sk})$
3	K	G	$\{K_{sg}, \#K_{sg}, N_g, N_s, S\|\sim N_g\}(K_g k)$, $\{K_{sg}\}(K_{sk})$
4	G	S	$\{K_{sg}\}(K_{sk}), \{N_s, K_{sg}, S\|\equiv \#K_{sg}\}(K_s k)$

Table 2. Abstracted version of the Yahalom protocol, specified in BAN logic

#	Assumption
1	$A\|\equiv \#N_g$
2	$S\|\equiv \#N_s$
3	$S\|\equiv \#K_{sk}$
4	$A\|\equiv K_{gk}$
5	$K\|\equiv K_{gk}$
6	$S\|\equiv K_{sk}$
7	$K\|\equiv K_{sk}$
8	$K\|\equiv K_{sg}$
9	$G\|\equiv K\|\Rightarrow K_{sg}$
10	$S\|\equiv K\|\Rightarrow K_{sg}$
11	$S\|\equiv K\|\Rightarrow \#K_{sg}$
12	$S\|\equiv G\|\equiv K\|\Rightarrow \#K_{sg}$
13	$G\|\equiv S\|\Rightarrow S\|\sim N_b$
14	$S\|\equiv secret N_s$

Table 3. Assumptions for the Yahalom protocol (formulated in BAN logic)

References

[1] B. Beckert, U. Keller, and P. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002.

[2] S. Brackin, C. Meadows, and J. Millen. A CAPSL interface for the NRL protocol analyzer. In *In Proceedings of ASSET99*, IEEE, 1999.

[3] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. In *ACM Operating Systems Review 23(5) / Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 1989.

[4] H. Chen, J. Clark, and J. Jacob. Automated design of security protocols. In *Proceedings of the 2003 Congress on Evolutionary Computation*, volume 3, pages 2181–2188. IEEE Press, 2003.

[5] E. Denney and B. Fischer. Formal safety certification of aerospace software. In *Proceedings of Infotech@Aerospace*. AIAA, September 2005.

[6] E. Denney, B. Fischer, J. Schumann, and J. Richardson. Automatic certification of Kalman filters for reliable code generation. In *Proceedings of the IEEE Aerospace*

Conference, Big Sky, Montana, March 2005. IEEE.

[7] J. Jürjens. *Secure Systems Development with UML*. Springer, 2004.

[8] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer, 1996.

[9] W. Marrero, E. Clarke, and S. Jha. Model checking for Security Protocols. Technical Report CMU-CS-97-139, Carnegie Mellon University, 1997.

[10] J. Millen and F. Muller. Cryptographic protocol generation from CAPSL. SRI Technical report SRI-CSL-01-07, 2001.

[11] S. Neema, Z. Kalmar, F. Shi, A. Vizhanyo, and G. Karsai. A visually-specified code generator for simulink/stateflow. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 275–277, 2005. IEEE.

[12] L. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.

[13] A. Price, K. Kosaka, and S. Chatterjee. A secure key management scheme for sensor networks. In *Proc. 10th Americas Conference on Information Systems*, 2004.

[14] H. Saidi. Towards automatic synthesis of security protocols. In *Logic-Based Program Synthesis Workshop, AAAI 2002 Spring Symposium*, Stanford University, California, 2002.

[15] J. Schumann. Automatic Verification of Cryptographic Protocols with SETHEO. In *Conference on Automated Deduction (CADE) 14*, LNAI, pages 87–100. Springer, 1997.

[16] J. Schumann. PIL/SETHEO: A Tool for the Automatic analysis of Authentication Protocols. In *Proc. Computer Aided Verification (CAV) '99*, LNAI. Springer, 1999.

[17] J. Schumann and J. Whittle. Automatic synthesis of agent designs in UML. 1871:148–162, 2001.

[18] G. Steel, A. Bundy, and E. Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. *AISB Journal*, 1(2), 2002.

[19] G. Steel, A. Bundy, and E. Denney. Using the CORAL system to discover attacks on security protocols. In *Computer Systems: Theory, Technology and Applications*. Springer-Verlag, 2003. Festschrift for Roger Needham.

[20] B. Tobler and A. Hutchison. Generating network security protocol implementations from formal specifications. In *Proceedings IFIP World Computer Congress - CSES 2004 2nd International Workshop on Certification and Security in Inter-Organizational E-Services*, 2004.

[21] J. Whittle, R. Kwan, and J. Saboo. From scenarios to code: An air traffic control case study. *Journal of Software and Systems Modeling*, 2004.

[22] J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *Proceedings of International Conference on Software Engineering (ICSE 2000)*, pages 314–323, Limerick, Ireland, 2000.

[23] H. Zhou and S. Foley. Fast automatic synthesis of security protocols using backward search. In *Proc. FMSE (Formal Methods in Security Engineering), 2002*, 2002.

Dr. Johann Schumann *(PhD 1991, habilitation degree 2000) is a Senior Scientist with RIACS and working in the Robust Software Engineering Group at NASA Ames. He is engaged in research on verification and validation of autonomy software, adaptive controllers, and learning software. He is also working on automatic generation of navigation and state estimation code. Dr. Schumann is author of a book on theorem proving in software engineering and has published more than 60 articles on automated deduction and its applications, automatic program generation, and neural network oriented topics.*

Dr. Ewen Denney *(PhD University of Edinburgh, 1999) has published 40 papers in the areas of automated code generation, software modeling, software certification, and the foundations of computer science. His research interests at NASA are mainly on the theory and application of automated code generation and automated code certification.*