# Toward Synthesis, Analysis, and Certification of Security Protocols — Position Paper —

Johann Schumann
RIACS / NASA Ames, Moffett Field, CA 94035
schumann@email.arc.nasa.gov

## 1 Introduction

Implemented security protocols are basically pieces of software which are used to (a) authenticate the other communication partners, (b) establish a secure communication channel between them (using insecure communication media), and (c) transfer data between the communication partners in such a way that these data only available to the desired receiver, but not to anyone else. Such an implementation usually consists of the following components: the protocol-engine, which controls in which sequence the messages of the protocol are sent over the network, and which controls the assembly/disassembly and processing (e.g., decryption) of the data, the cryptographic routines to actually encrypt or decrypt the data (using given keys), and the interface to the operating system and to the application.

For a correct working of such a security protocol, *all* of these components must work flawlessly. Many formal-methods based techniques for the analysis of a security protocols have been developed. They range from using specific logics (e.g., BAN-logic [4]) or higher order logics [12] to model checking [2] approaches. In each approach, the analysis tries to prove that no (or at least not a modeled intruder) can get access to secret data. Otherwise, a scenario illustrating the attack may be produced. Despite the seeming simplicity of security protocols ("only" a few messages are sent between the protocol partners in order to ensure a secure communication), many flaws have been detected.

Unfortunately, even a perfect protocol engine does not guarantee flawless working of a security protocol, as incidents show. Many break-ins and security vulnerabilities are caused by exploiting errors in the *implementation* of the protocol engine or the underlying operating system. Attacks using buffer-overflows are a very common class of such attacks. Errors in the implementation of exception or error handling can open up additional vulnerabilities. For example, on a website with a log-in screen, multiple tries with invalid passwords caused the expected error message (too many retries), but let the user nevertheless pass.

Finally, security can be compromised by silly implementation bugs or design decisions. In a commercial VPN software, all calls to the encryption routines were incidentally replaced by stubs, probably during factory testing. The product worked nicely, and the error (an "open" VPN) would have gone undetected, if a team member had not inspected the low-level traffic "out of curiosity".

Also, the use "secret" proprietary encryption routines can backfire, because such algorithms often exhibit weaknesses which can be exploited easily (see e.g., DVD encoding).

Summarizing, there is large number of possibilities to make errors which can compromise the security of a protocol. In today's world with short time-to-market and the use of security protocols in open and hostile networks for safety-critical applications (e.g., power or air-traffic control), such slips could lead to catastrophic situations.

Thus, formal methods and automatic reasoning techniques should not be used just for the formal proof of absence of an attack, but they ought to be used to provide an "end-to-end" tool-supported framework for security software. With such an approach all required artifacts (code, documentation, test cases), formal analyses, and reliable certification will be generated automatically, given a single, high level specification. By a combination of program synthesis, formal protocol analysis, certification, and proof-carrying code, this goal is within practical reach, since all the important technologies for such an approach actually exist and only need to be assembled in the right way.

## 2 The Ingredients

In such an envisioned end-to-end solution for security protocols, a number of tasks must be performed upon the input specification. In the following sections, we will briefly discuss the required steps and present existing approaches which can provide a basis.

The input to such a tool will be a detailed specification of the protocol. Typically, a security protocol is specified as a sequence of messages which are sent between the different protocol participants. Thus, scenarios, sequence diagrams, or STDs are suitable for specifying security protocols (cf. [13, 9]). Due to its widespread use and the existence of (commercial) tools, UML might be a good choice. Its various ways of modeling systems allows the protocol designer a flexible way of specifying all aspects of the security protocol. Besides the "raw" protocols, there will be annotations (e.g., in BAN logic), scenarios for failures and exception handling ("don't-do use-cases"), as well as definitions and interfaces regarding the actual data to be communicated and the connection with the underlying operating system. Also attack scenarios could be specified within UML. Such a UML specification provides the basis for all analyses and synthesis tasks.

### 2.1 Protocol Analysis

A first and important step during protocol design is the automatic analysis of the protocol. Here, properties about authentication, secrecy, confidentiality, etc. are proven in a formal way, or attack scenarios are generated. For this task, a large body of approaches and tools exist. Counter examples which represent a successful attack on the protocol should be converted into UML sequence diagrams for easy human readability and simulation. Positive proofs should be, where possible, represented in a formalism understandable for the protocol designer/analyzer. For example, the tool BAN-SETHEO [15] uses a first-order ATP to find the proofs, and then converts and typesets the proofs as proofs in the BAN-logic. Although, in many cases, an "OK" might be sufficient, such proofs should be kept in such a way that they can be checked (manually or automatically) by an independent certification authority.

## 2.2 Analysis of Cryptographic Routines

Traditional protocol analysis (as discussed above) assumes that implementation and the encryption routines are correct. Due to their mathematical complexity, encryption routines need to be checked with regard to their encryption strength and other properties. This task is probably the most difficult part in the entire scenario, and, within the foreseeable future, off-line manual development of such proofs will be the only option. For practical purposes, libraries of standardized and certified cryptographic routines can be used.

## 2.3 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 are therefore proposing the automatic generation of the protocol software using techniques of automatic program synthesis. Based on the protocol specification and guided by required properties, an implementation of the protocol software can be synthesized automatically. By using a formal-methods based approach, it can be guaranteed (or certified, see below) that the implementation does not violate any of the security properties established and proven during the protocol analysis.

There is a number of interesting approaches toward automatic synthesis of security protocols, e.g., [14], [20], or [5]. A main goal of these approaches is to automatically generate a secure and efficient protocol, given a property-based specification. In our framework, however, we can be much less ambitious. We already have a detailed specification of the protocol (as set of sequence diagrams). Therefore, we can use an approach like [19] which takes a set of annotated UML sequence diagrams and generates highly structured statecharts from them. From statecharts, traditional code-generation techniques can be used to yield the final executable code.

In this application, correctness of the generated artifacts is paramount. Therefore, traceability between specification and synthesized code, as well as the generation of the appropriate documentation is important. Here, work from program synthesis, e.g., [18, 17] can be adapted.

## 2.4 Automatic optimization of protocols

In application with small bandwidth (e.g., aircraft-ground communications), it is important that the execution of the security protocol does not unnecessarily burden the communication channels. Thus, a security protocol with a minimal number of protocol steps should be preferred. Also important is the ratio $r$ of the size of actual (secure) user data vs. the transmitted data. In the traditional layered protocol architecture, each layer is adding its own wrapper and control information around the data, leading to small values of $r$. Some protocols, for example, will send a 1024 byte package, even if only a few bytes need to be transmitted. Optimizing such a behavior manually, however, is extremely error-prone. Furthermore, optimization goals for security protocols can contradict those for traditional protocols (e.g., replacing $f^{-1}(f(D))$ by $D$ is not legal if $f$ is a cryptgraphic routine). Techniques for formal protocol optimization (e.g., [10]) could be of high value here.

### 2.5 Automatic Certification of Protocol Software

Once the protocol software has been synthesized, it should be ready to be deployed. However, a protocol synthesis tool is an extremely large and complex piece of software itself, so its formal verification is not practical. In order to overcome this problem, the approach of product certification can be used: instead of verifying of the synthesizer, each synthesized program is certified individually. Certificates are automatically generated proofs that the program fulfills certain properties. These certificates can be checked independently (e.g., by a certification authority).

We have developed a program synthesis system (AUTOBAYES/AUTOFILTER) with such a certification extension [16, 7]. During synthesis, the program is automatically adorned by annotations. Then, a Hoare-style verification generator produces a number of first-order logic proof obligations. After simplification, these proof obligations are processed by an automated theorem prover. With this approach, traditional language-specific properties (e.g., array-bounds, variable initialization) can be proven; also domain-specific properties can be handled by this system. Such a system can be used to certify all important properties of synthesized security software. For additional security, this approach can be combined with the proof-carrying code techniques [6, 11, 1] to provide tamper-proof certificates when the code is used in mobile applications.

In addition to the tasks discussed above, the specific characteristics of security protocols must be threaded through the entire software lifecyle. For example, the software process (in particular the parts dealing with V&V) needs to be augmented specifically to handle security protocols in a proper way (cf. [3]). Also testing of a security protocol is somewhat different from testing ordinary software. Finally, the successful operation of a security protocol substantially relies on its correct use: the best security protocol is of little use if users write the passwords on sticky notes and place them next to the computer, or if passwords are "reused" [8].

## 3 Conclusions

The design, development, and deployment of a reliable and secure security protocol has to address many issues which go beyond the analysis of the core security protocol. Tremendous progress in this field has been accomplished by using automated reasoning techniques (e.g., theorem proving, model checking) for the analysis task. The potential for automated reasoning techniques in the area of security protocols is by far not exhausted: during practically every life-cycle step of a protocol development, formal-methods based techniques can—and must—be applied. By combining existing approaches to protocol analysis, program synthesis, logic-based optimization, automated certification, and proof carrying code, it will be possible to develop a powerful and practically useful framework and tool for safe and secure "end-to-end" protocol design.

## References

[1] A. W. Appel and A. Felty. A semantic model of types and machine instuctions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 243–253. ACM Press, 2000.

[2] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mädersheim, M. Rusinowitch, M. Turuani, L. Vigan, and L. Vigneron. The AVISS Security Protocol Analysis Tool, systems. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *LNCS*, pages 349–353. Springer, 2002.

[3] R. Breu, K. Burger, M. Hafner, J. Jürjens, G. Popp, G. Wimmel, and V. Lotz. Key issues of a formally based process model for security engineering. In *ICSSEA 2003 - Sixteenth International Conference "Software & Systems Engineering & their Applications"*, 2003.

[4] 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.

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

[6] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, 2000.

[7] E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software, 2004. Accepted for IJCAR'04.

[8] B. Ives, K. Walsh, and H. Schneider. The domino effect of password reuse. *Communications of the ACM*, 47(4):75–78, 2004.

[9] J. Jürjens and G. Wimmel. Formally testing fail-safety of electronic purse protocols. In *16th International Conference on Automated Software Engineering (ASE 2001)*, pages 408–411, 2001.

[10] C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artifical Intelligence*, pages 316–332. Springer Verlag, 1998.

[11] G. C. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104. IEEE Computer Society Press, 1998.

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

[13] G. Popp, J. Jürjens, G. Wimmel, and R. Breu. Security-critical system development with extended use cases. In *10th Asia-Pacific Software Engineering Conference (APSEC 2003)*, 2003.

[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, B. Fischer, M. Whalen, and J. Whittle. Certification support for automatically generated programs. In *In Proceedings of the Thirty-Sixth Annual Hawaii International Conference on System Sciences (HICSS-36)*. IEEE, 2003.

[17] J. Schumann and P. Robinson. [] or success is not enough: Current technology and future directions in proof presentation. In *Future Trends in Automated Deduction (during IJCAR 2001)*, 2001.

[18] J. van Baalen, P. Robinson, M. Lowry, and Th. Pressburger. Explaining synthesized software. In D. F. Redmiles and B. Nuseibeh, editors, *Proc 13th IEEE Conference on Automated Software Engineering*, pages 240–248, 1998.

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

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