Visual Model-Driven Design, Verification and Implementation of Security Protocols

Visual Model-Driven Design, Verification and Implementation of Security Protocols

P. Bettassa Copet[*], *A. Pironti*[**], *D. Pozza*[†], *R. Sisto*[*], *and P. Vivoli*[‡]

∗ Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy

∗∗ Prosecco, INRIA, France

† Primeur, Italy

‡ Spike Reply, Italy

Keywords Security protocols; Model-driven development; Formal methods; Visual models

Abstract *A novel visual model-driven approach to security protocol design, verification, and implementation is presented in this paper. User-friendly graphical models are combined with rigorous formal methods to enable protocol verification and sound automatic code generation. Domain-specific abstractions keep the graphical models simple, yet powerful enough to represent complex, realistic protocols such as SSH. The main contribution is to bring together aspects that were only partially available or not available at all in previous proposals.*

1 Introduction

Security protocols are communication protocols that aim at establishing secure communication between two or more parties (called *principals*), even in the presence of an attacker, by orchestrating the use of cryptography and message exchanges. Secure communication means, for example, that exchanged data remains confidential to third parties, or that tampered data can be reliably identified by the honest principals.

Despite the apparent simplicity of security protocols, the task of designing and implementing them correctly is inherently difficult, as witnessed by some widely deployed protocols which were found to be flawed several times, many years after their release (e.g. [1]). On one hand, this difficulty can be alleviated by adopting a model-driven development (MDD) approach [2], which may reduce development time and improve code correctness through automatic generation of substantial parts of implementation from visual (or textual) models. On the other hand, formal verification can systematically rule out subtle logical flaws, provided the security protocol models are expressed formally.

The proposed approach integrates intuitive visual modeling with formal analysis and sound generation of interoperable code for the whole class of security protocols. MDD is leveraged to hide the complexity of a full implementation during the design phase, so that the developer only needs focus on a simplified abstract model. During this phase, formal verification is used in order to get assurance about logical correctness. Later on the details needed to get an implementation can be added and code generation can take place. This approach is implemented as an Eclipse plugin, named Spi2JavaGUI[1].

The remainder of the paper is organized as follows. Section 2 introduces some background about spi calculus and Spi2Java, the formal methods and tools that Spi2JavaGUI exploits. Section 3 describes the Spi2JavaGUI framework by means of a running example. Section 4 discusses related work and Section 5 concludes.

2 Background: Formalism and Tools

The formal language underlying the proposed approach is the domain-specific spi calculus [3] language. A model of a security protocol in spi calculus is composed of a system of parallel processes. Each protocol principal is modeled by a process, exchanging messages with other processes via insecure communication channels. A Dolev-Yao [4] attacker has access to the insecure channels and can eavesdrop, forge, alter or drop messages. Typically, a principal is described as a sequential program that performs a sequence of input/output operations, checks on received data, and cryptographic operations.

In this work, the ProVerif tool [5] is leveraged to formally prove the security properties of a protocol model. ProVerif is a fully automatic verification engine which accepts (an extension of) spi calculus as its input modeling language.

Semi-automatic code generation from an abstract model expressed in spi calculus can be done using the Spi2Java [6], [7] framework in two steps. As a first step the abstract model is refined, by adding low level details (e.g. which cryptographic algorithm must be used for a hash operation, or how to transform a message into its network binary representation). These details are written in a separate document, called the refinement document, so as to keep the spi calculus model as simple (and readable) as possible. As a second step the consistency between the refinement document and the spi calculus model is checked and the automated code generator of Spi2Java is invoked to generate the implementation code from the spi calculus model and the refinement document.

3 Spi2JavaGUI

The Spi2JavaGUI approach is an enhancement of the Spi2Java approach, that reuses the core of Spi2Java, while allowing the user to avoid the complexity of writing models directly in the domain-specific spi calculus. Also, Spi2JavaGUI aims at making refinement and the whole development process simpler and less error prone.

Figure 1 shows the workflow of the framework. Both the spi calculus model and the refinement document are edited jointly in visual form, while code generation is accomplished by re-using parts of the Spi2Java framework, which offers a sound code generation technique [8]. The integrated development environment (IDE) guides the user through the different steps of model verification, refinement and implementation generation.

[1] Available at http://spi2java.polito.it/gui/updates

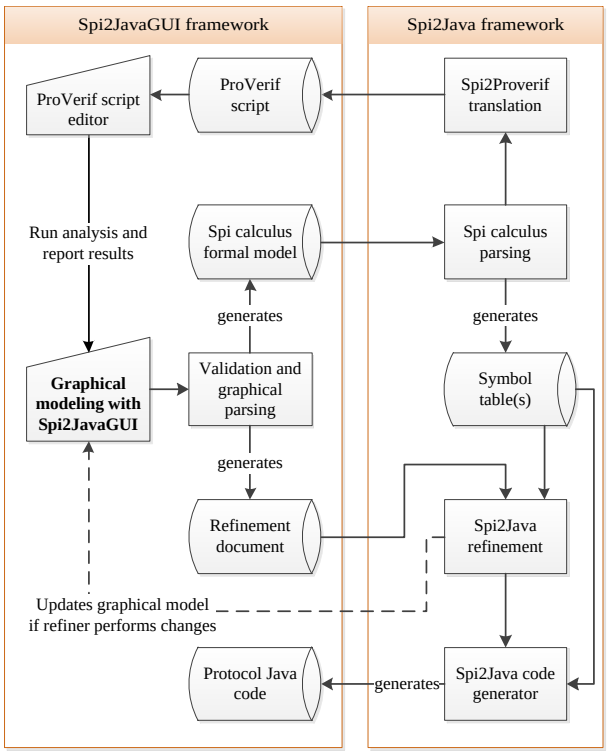

Figure 1: Workflow of Spi2JavaGUI and Spi2Java.

3.1 The Model

According to the Model-View-Controller paradigm, Spi2JavaGUI holds both an underlying model of the protocol being developed and a graphical view of this model.

The Spi2JavaGUI protocol model has the same structure and semantics as a corresponding spi calculus model, but is presented with a different syntax and has some extensions. Some of the extensions provide the refinement information necessary for code generation, while other extensions provide visual information, necessary to visualize model elements.

Graphic and refinement information is coupled strongly with model elements, by incorporating it inside each model element.

Given the strict correspondence between a Spi2JavaGUI model and its associated spi calculus specification, the former can be translated into the latter automatically, and it can be assumed that the spi calculus specification resulting from the translation of a given Spi2JavaGUI model actually provides the formal semantics of the Spi2JavaGUI model.

3.2 Visual Syntax

The visual syntax is based on a single *block-oriented* and *data-flow-oriented* view of the protocol, where principals and message flows are represented according to the common and intuitive representation of message sequence charts (MSC). Figure 2 shows the diagram of a simple authenticated Remote Procedure Call (RPC) protocol.

The client and server roles are represented by the left hand side and right hand side main blocks of the picture, respectively. The client (A) sends a request message to the server (B) which responds with a response message. The structure of the two messages can be represented as follows.

1. $A \rightarrow B : S, Na, H(K_{AB}, REQ, Na, S)$
2. $B \rightarrow A : f(S), H(K_{AB}, RES, Na, S, f(S))$

The request message is composed of a string S, which identifies the remote procedure to call and its parameters, a cryptographic nonce (i.e. a fresh random number) N_a, and a keyed hash (or HMAC), calculated as the cryptographic hash of the concatenation of a shared key K_{AB}, a tag (constant string) REQ, the nonce N_a and the string S. The response message is composed of $f(S)$, which represents the output of the remote procedure call, and a HMAC calculated with the shared key K_{AB}, the RES tag, the nonce N_a, and the request and

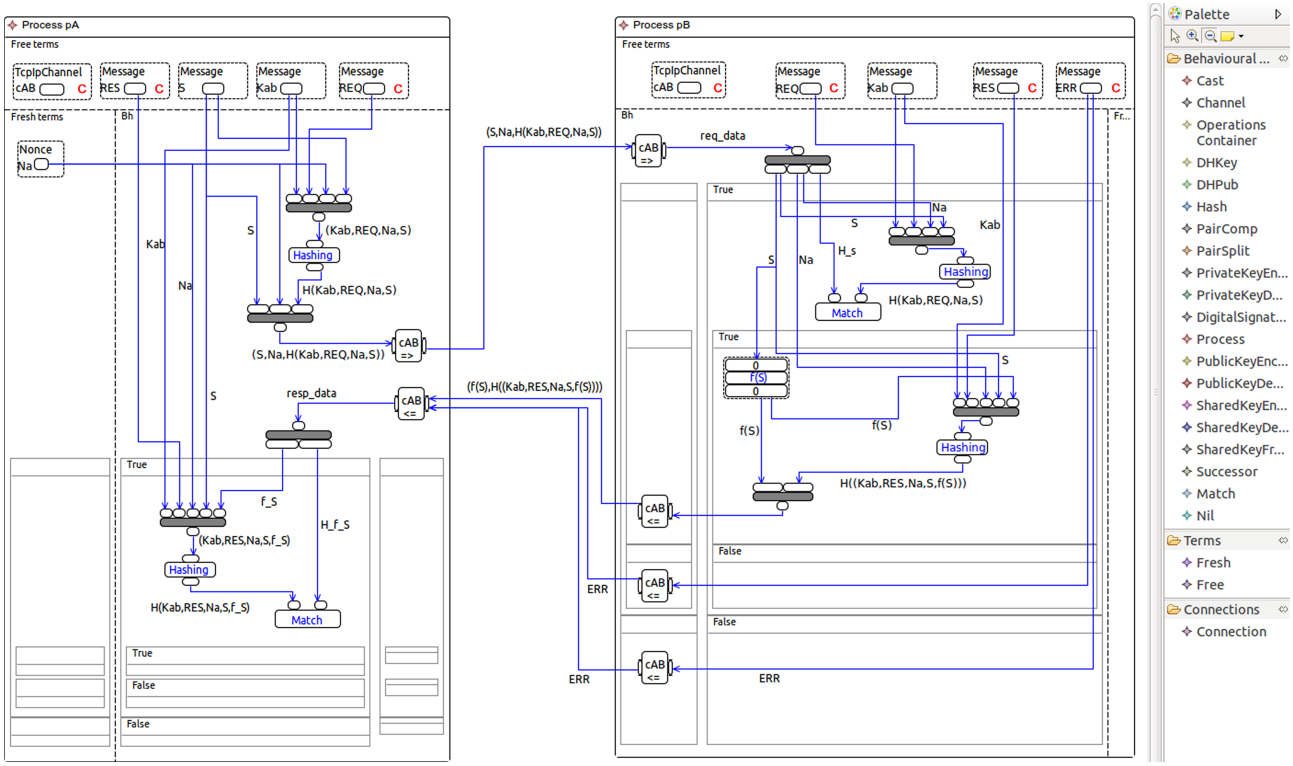

Figure 2: The Spi2JavaGUI main view with a simple RPC protocol model.

response strings. For simplicity, in this example $f(S)$ is defined as the cryptographic hash of S, but in general this can be an arbitrary function.

The way each message is built from elementary components before being sent and the way it is processed after having been received are specified using operation blocks inside each protocol role. Special blocks in the upper area of each role block represent input parameters and protocol constants.

Refinement information of each block can be set in a *property sheet*, available on block selection.

Abstraction is supported through the possibility to have nested blocks and to hide non-relevant aspects when one is focusing on a specific aspect of the protocol. For example, blocks that imply a binary decision are rendered by having separate areas for the True and False branches, as shown in the example of Figure 2. However, each branch can be collapsed by the user when attention has to be focused on the other branch.

Connections between blocks of the model establish dependency relations, because the output value(s) of an operation depend on the block(s) that are connected to its input port(s). The order of operations implied by these dependencies can be automatically computed so as to convert the data-flow representation that is displayed, into the control-flow one of the underlying model. In this way, the model always includes, for each process, a fully ordered set of operations, which can be directly translated into spi calculus syntax.

Visual modeling and abstraction techniques are implemented in a prototype editor which is part of the Spi2JavaGUI framework.

3.3 Formal Analysis and Code Generation

Validation of the modeled protocol aimed at detecting modeling errors is accomplished both by performing in-editing checks and by evaluating OCL-like rules that enforce the internal consistency of the model. For example, a block missing an input connection is reported as a modeling error to the user.

Once the protocol is modeled, and the validation phase passed, the user can run the ProVerif [5] tool to perform formal analysis. The interaction with ProVerif is managed internally by Spi2JavaGUI, which automatically translates the model into a formal spi calculus specification in the ProVerif input syntax. Note that formal verification does not require that implementation details have been specified with their final values, because formal verification just uses the abstract model and disregards implementation details.

The last development phase is the generation of concrete implementation code in the Java language. To that end, Spi2JavaGUI abstracts the interaction with the Spi2Java libraries and, thanks to the refinement information that is now embedded in the graphical model (via the property sheets), the code is generated by a

single automatic step.

Using the Spi2Java code generation engine to obtain the final code brings an added value, i.e. the formally proved soundness of the code generation process. More precisely, as shown in [8], the generated code is guaranteed to fulfill security properties such as secrecy and authentication under a Dolev-Yao [4] attacker, provided that the same properties hold on the abstract formal model from which the code has been generated. Since security properties are formally verified on the abstract model before generating code, high confidence on the generated code is finally obtained.

Spi2JavaGUI has been successfully applied to real security protocols. As an example, it has been used to develop an interoperable implementation of SSH.

4 Related Work

GSPML [9] is a custom visual modeling formalism to represent high-level protocol specifications in a concise manner, aiding in both understanding and verification. The event-based and trace-oriented style in which protocols are modeled makes GSPML amenable to formal verification; however GPSML abstracts internal computation performed within protocol agents, which is essential for code generation. Hence, the framework provides no mechanism to generate a protocol implementation.

SPEAR II [10] allows visual modeling of security protocols integrated with the GNY formal analysis tool, which is based on a variant of the BAN modal logic [11]. The model is based on MSC-like diagrams, and only the main sequence of messages exchanged in a normal run of the protocol is represented. Thus, while this is enough for a BAN-logic analysis, this is less suitable for a rigorous transition from model to protocol implementation.

UMLsec [12] extends UML enabling visual modeling (based on annotated sequence charts) and formal verification of security protocols. The main differences, with respect to the approach proposed here, is that messages are represented only textually, and implementation details cannot be added in the visual model. This makes UMLsec amenable to formal verification, but not to code generation. Extending UMLsec to support automatic code generation would technically be possible. However, this would require the addition of new views, with the inherent problem of keeping them synchronized.

Smith et al. [13] exploit the standard *ports* and *protocols* features of UML 2.0 to define executable security protocol models, from which executable code can be generated. However, no formal semantics is provided for the visual model, so formal verification cannot be supported.

5 Conclusion

A novel domain-specific approach for visual MDD of security protocols has been presented. This approach combines visual editing, which is more intuitive than existing textual formal languages, with a rigorous formal approach to model verification and code generation.

The soundness of the Spi2Java framework is leveraged by Spi2JavaGUI to generate code with high confidence of correctness.

Compared to existing textual models, the proposed visual model provides an effective way of hiding the model complexity. Low-level aspects that are not relevant during protocol design can be collapsed away, and drilled-down later when specifying protocol implementation details. Similarly, specific execution scenarios can be highlighted (e.g. the typical protocol run), despite the model encompassing all scenarios at once (including handling of all error conditions).

Spi2JavaGUI visual models substantially differ from previous visual proposals. While UML-based approaches use many views and diagrams, Spi2JavaGUI provides a single comprehensive view, tailored to the specific needs of security protocol modeling. Furthermore, with respect to other approaches providing linkage with formal models that are oriented to experts in formal methods and security, Spi2JavaGUI uses data-flow-oriented models that are similar to other models well-known by software developers.

The proposed Spi2JavaGUI formalism fulfills the main key characteristics for a model-driven approach, as proposed by Selic [2]. *Abstraction* is enforced by view collapsing and hierarchical models. *Understandability* is achieved by visualizing the protocol data path of each protocol session in MSC-like style. In addition, the Spi2JavaGUI approach clearly fulfills *accuracy*, i.e. true-to-life representation and *predictiveness*, i.e. possibility to predict interesting but non-obvious properties, because of its ability to discover attacks on the modeled protocol. Even if not verified experimentally yet, it is believable that the Spi2JavaGUI approach also fulfills *inexpensiveness*, i.e. the fact that models are significantly cheaper to construct and analyze than the modeled system, because Spi2JavaGUI models are relatively simple and let the user focus on the protocol logic alone, before being involved in other implementation details.

References

[1] M. Ray, "Authentication gap in TLS renegotiation," http://extendedsubset.com/Renegotiating_TLS.pdf, 2009.

[2] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Softw.*, vol. 20, pp. 19–25, September 2003.

[3] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: the spi calculus," in *4th ACM conference on Computer and communications security*, 1997, pp. 36–47.

[4] D. Dolev and A. C.-C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–207, 1983.

[5] B. Blanchet, "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules," in *14th IEEE workshop on Computer Security Foundations*, 2001, pp. 82–.

[6] D. Pozza, R. Sisto, and L. Durante, "Spi2Java: automatic cryptographic protocol Java code generation from spi calculus," in *Advanced Information Networking and Applications, 2004. 18th International Conference on*, vol. 1, 2004, pp. 400 – 405 Vol.1.

[7] A. Pironti, D. Pozza, and R. Sisto, "Formally-based semi-automatic implementation of an open security protocol," *Journal of Systems and Software*, vol. 85, p. 835–849, 2012.

[8] A. Pironti and R. Sisto, "Provably correct Java implementations of Spi Calculus security protocols specifications," *Computers & Security*, vol. 29, p. 302–314, 2010.

[9] J. McDermott, "Visual security protocol modeling," in *Workshop on New security paradigms*, 2005, pp. 97–109.

[10] E. Saul and A. Hutchison, "Using GYPSIE, GYNGER and Visual GNY to Analyze Cryptographic Protocols in Spear II," in *Advances in Information Security Management & Small Systems Security*, 2001, vol. 72, pp. 73–85.

[11] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication," *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 18–36, Feb. 1990.

[12] J. Jürjens, "Developing high-assurance secure systems with UML: a smartcard-based purchase protocol," in *8th IEEE international conference on High assurance systems engineering*, 2004, pp. 231–240.

[13] S. Smith, A. Beaulieu, and W. Phillips, "Modeling and verifying security protocols using UML 2," in *Systems Conference (SysCon), 2011 IEEE International*, April 2011, pp. 72 –79.