Protocol Implementation Generator

Jose Quaresma and Christian W. Probst

Technical University of Denmark
{jncq,probst}@imm.dtu.dk

Abstract. Users expect communication systems to guarantee, amongst others, privacy and integrity of their data. These can be ensured by using well-established protocols; the best protocol, however, is useless if not all parties involved in a communication have a correct implementation of the protocol and all necessary tools. In this paper, we present the *Protocol Implementation Generator* (PiG), a framework that can be used to add protocol generation to protocol negotiation, or to easily share and implement new protocols throughout a network. PiG enables the sharing, verification, and translation of communication protocols. With it, partners can suggest a *new protocol* by sending its specification. After formally verifying the specification, each partner generates an implementation, which can then be used for establishing communication. We also present a practical realisation of the *Protocol Implementation Generator* framework based on the LySatool and a translator from the LySa language into C or Java.

1 Introduction

The Internet, and network technology in general, are increasingly used for providing central functionality for applications and systems, most notably through, *e.g.*, infrastructure services such as the cloud or web services, or any kind of client-server architecture. This seamless integration of network facilities into user applications has enabled development of new application domains, which in turn resulted in increased integration of networks. While in the past data was mostly stored locally, today, due to the wide availability of networks, data is often being communicated or accessed via local or wide-area networks. Most of the time, users do not need to be aware of where their data is located, and how it is communicated. In fact, being able to access data from wherever one wants to is one of the driving forces behind network integration.

Being able to access data via the network is only half the story, of course. What is (often implicitly) expected is that data is secured by the application and system, both when stored and in transit. Users expect communication systems to guarantee, amongst others, privacy and integrity of their data. When storing data, this can be achieved, *e.g.*, by (a combination of) access control and cryptography. In communication, this can be ensured by using well-established protocols.

Protocols are usually specified by means of protocol narrations, which describe in detail how the involved partners communicate with each other, and

T. Aura, K. Järvinen, and K. Nyberg (Eds.): NordSec 2010, LNCS 7127, pp. 256–268, 2012.
© Springer-Verlag Berlin Heidelberg 2012

which messages they exchange. Protocol narrations serve two purposes: they can be used for formally verifying the protocol, and for guiding the implementation of the protocol. Recently, researchers have looked into generating protocol implementations from specifications [1–3] and extracting protocol specifications from implementations [4], thereby narrowing the gap between the formal, verified specification, and the usually unverified implementation.

The best protocol, however, is useless if not all parties involved in a communication "speak it", that is, have an implementation of the protocol and all necessary tools. This is why protocols such as "Secure Socket Layer" (SSL) [5, 6] start with a negotiation phase where the partners agree on a suite of algorithms necessary for establishing a connection using the protocol. Whenever no such common algorithms are found, the negotiation phase fails.

In this paper we present the *Protocol Implementation Generator* (PiG) [7], a framework for adding protocol *generation* to protocol *negotiation*. In PiG, when the negotiation phase fails, one of the partners can suggest a *new protocol* by sending its specification. After formally verifying the specification against a set of security properties, each partner generates an implementation, which then can be used for establishing communication. We also present a practical realisation of the *Protocol Implementation Generator* based on the LySatool and a LySa to C and Java translator.

The rest of this paper is structured as follows. We start with a general overview of the *Protocol Implementation Generator* in Section 2, followed by a presentation of a prototype realisation in Section 3. After discussing related work (Section 4), Section 5 concludes the paper and gives an outlook on future work.

2 The Protocol Implementation Generator

In this section we describe the overall layout of the PiG framework, as well as the individual components necessary to realise it. In the next section we will present a concrete implementation of PiG based on the LySatool [8].

The idea behind the PiG framework is to allow communication partners to establish a secure communication channel *without* previously sharing an implementation for the protocol. Instead, one of the partners can suggest a new protocol by sending its formal specification, *e.g.*, as a protocol narration. This formal specification can be verified by the other partner, and if the verification succeeds, the specification can be used to automatically generate a protocol implementation.

Deriving the protocol implementation directly from the specification is an important aspect of our approach, closing the often found gap between the two when protocols are implemented by hand.

Figure 1 shows the process implemented in the PiG framework. When protocol negotiation fails (steps 1,2), Alice sends a specification for a "new" protocol (step 3). Bob checks the specification against the desirable security requirements (step 4) and, if it is found to be safe, he generates the implementation for the protocol (step 6), and Alice and Bob start communication using the new protocol (step 7).

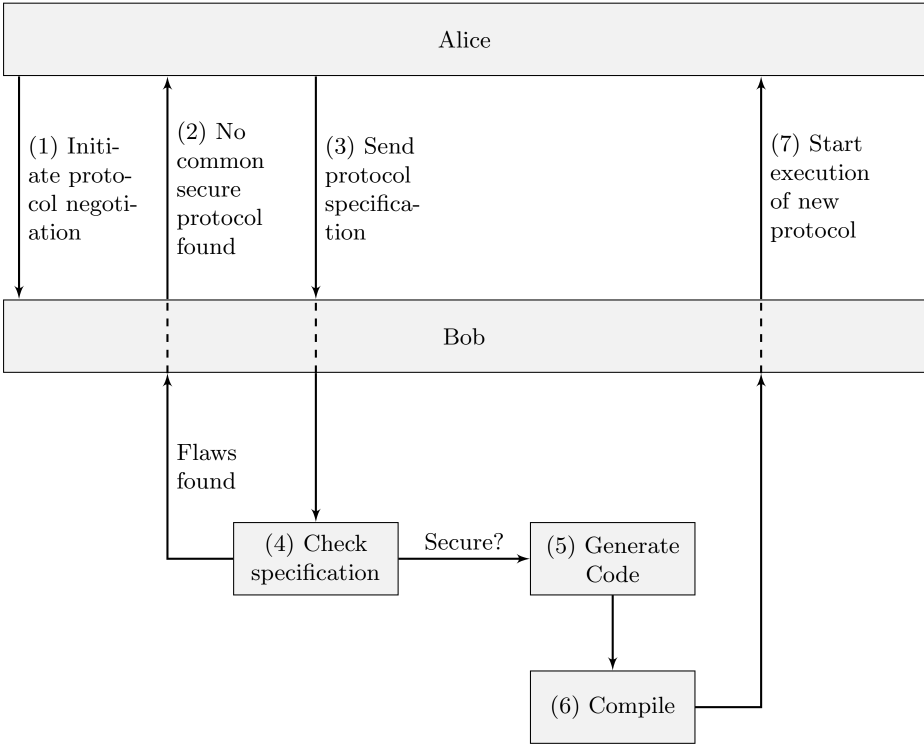

Fig. 1. The process implemented in the *Protocol Implementation Generator*

The *Protocol Implementation Generator* is based on three core components:

- the protocol specification language,
- the protocol verifier, and
- the code generator.

The only requirement for a realisation is that all involved partners have these components. While the components used in the framework can be freely chosen, they must fulfil the requirement that the verifier and the code generator work on the same formalism. Furthermore, the code generator must be complete with respect to the elements of the specification formalism. This is essential, since the *Protocol Implementation Generator* should be transparent to the user, and therefore code generation should be automated, with no need for human interaction.

A possible extension is the use of proof-carrying code [9] techniques to avoid a full re-analysis of the specification before code generation. Instead, the protocol specification would be annotated with the proof for a verification condition that guarantees the protocol to pass verification.

Another extension in the same direction targets a slightly different arrangement of the steps described above. The overall process would be similar, but

in step 3 Alice would send to Bob an implementation of the "new" protocol and Bob, in step 4, would need to extract the protocol specification from the received implementation. Only then he would verify the correctness of the protocol. If found to be correct, Alice and Bob could start the execution of the protocol.

This idea of extracting the protocol specification from its implementation could also be useful in proving the correctness of the translation from the protocol specification to the protocol implementation—after the translation one could derive the specification from the generated implementation and verify it against the original specification.

3 A Practical Realisation of PiG

Having presented the *Protocol Implementation Generator* framework in the previous section, we now discuss a specific instance of the framework. The used configurable components are LySa [10] as protocol specification language, which is analysed using the LySatool [8], and a code generator based on ANTLR [11]. These components are at the core of a prototype realisation of PiG [7].

In order to give an example of the functioning of the prototype, we use as a running example the Otway-Rees protocol [12], which, while not in wide-spread use, is simple enough to be covered in an article. The Otway-Rees protocol is used in order to mutually authenticate two principals, Alice (A) and Bob (B) via a mutually trusted third party (S), and to generate a secret shared key that they can use to securely communicate.

```
1. A->B : M,A,B,{M,A,B,NA}:KA
2. B->S : M,A,B,{M,A,B,NA}:KA,{M,A,B,NB}:KB
3. S->B : M,{NA,KAB}:KA,{NB,KAB}:KB
4. B->A : M,{NA,KAB}:KA
5. B->A : {MSG}:KAB
```

Fig. 2. Pseudo-code specification of the Otway-Rees protocol

As seen in the protocol definition, in Figure 2, this is achieved with the exchange of four messages in total. A starts by sending B a message (line 1) encrypted with a shared key between A and S, which contains a nonce NA generated by A, a running serial M, and both principals' identities.

B, after receiving this message — that he cannot decrypt — sends it to S, together with another message encrypted with a key shared between B and S, with similar content to the one A sent, but in this case with a nonce NB generated by B (line 2). After S has received those two encrypted elements from B, it will verify the identities of the principals and the running serial M, and generate a symmetric key KAB that A and B will use to securely communicate.

S then encrypts that key together with the nonce generated by A with the key shared between S and A and, similarly, also encrypts that same key together with the nonce generated by B with the key shared by S and B.

S then sends those two elements to B (line 3), which decrypts the one encrypted with the key it shares with S. If the nonce matches, it will be ready to use the new key in future communications with A.

In the last step of the protocol (line 4), B sends to A the other element that it received from S. A decrypts it with the key it shares with S and, if the nonce matches, uses the new key for future communications with B.

In this example, an extra message MSG is sent in line 5 to illustrate the use of the new key shared between Alice and Bob.

3.1 LySa and the LySatool

LySa [10] is a process algebra aimed at specifying communication protocols. A LySa protocol specification consists of a standard protocol narration extended with annotations in order to remove analysis ambiguity that could arise from vague protocol narrations. This is especially beneficial in a setting like PIG, where the analysis and code generation should be automatic and transparent.

LySa is based on the Spi Calculus [13], which extends the π Calculus with cryptographic primitives that are used in the description and analysis of cryptographic protocols. In Spi Calculus, protocols are represented as processes and their security properties are stated by protocol equivalence.

The main difference between LySa and these calculi is that LySa assumes a single, global communication medium, to which all processes have access. The LySa approach seems, therefore, more natural when considering communication scenarios on the Internet, where it is not difficult to eavesdrop a conversation.

In LySa, only the legitimate part of the protocol in question is described, while the illegitimate, malicious part is implicitly modelled as a Dolev-Yao attacker [14]. The attacker can be equipped with some initial knowledge, which can be expanded by eavesdropping the network or by decrypting messages using known keys, it may encrypt messages using those keys, and is also able of initiating new sessions.

Protocols specified in the LySa calculus can be analysed for security properties using the LySatool [8]. The checked properties include authenticity, secrecy, and confidentiality and are fixed by the tool. The LySatool is implemented in Standard ML and uses the Succinct Solver [15]. It receives a LySa specification of a communication protocol as input, and performs a static analysis assuming the presence of the strongest possible attacker previously mentioned. The analysis result either indicates that the protocol is found to be secure, or not. In the latter case, further analyses have to be performed to distinguish between a false negative and a real security problem. These false negatives can happen due to the over-approximation analysis performed by LySa. This has to be taken into account when designing or choosing a protocol specification for use in the *Protocol Implementation Generator*.

In the initial phase of PiG, as described in the previous section, one of the principals is waiting for another principal to connect. When that happens, the latter will send a LySa protocol specification to the former. After receiving the protocol specification, the principal checks the protocol's security by verifying the specification with the LySatool. As mentioned above, in the current version, those security properties are implicit in the tool. If the protocol specification is found to be insecure, the tool reports that, and the process is terminated. If the protocol analysis identifies the protocol specification to be secure (which means that the security properties were guaranteed for the new protocol), the principal will execute the next step, starting the translation process.

LySa Specification of the Otway-Rees Protocol. When converting from pseudo-code, or Alice and Bob notation, to LySa notation there are several steps that need to be performed. It is necessary to model the principals running in parallel —in this case there are 3 principals— and each message exchanged in the pseudo-code needs to be modelled as being sent by one principal and being received by another. For example, line 1 of Figure 2 specifies that Alice sends a message to Bob (A->B:M,A,B,{M,A,B,NA}:KA), and that same message in the LySa specification consists of the sending on Alice's specification (<A,B,M,A,B,{M,A,B,NA}:KA>), and the receiving on Bob's specification ((A,B,M,A,B;y1)).

Another detail that needs to be taken into account is the pattern matching and the assignment of variables. This might not be trivial because, sometimes what is supposed to be pattern matched and assigned is not explicitly shown in the pseudo-code description of the protocol, and so it is necessary to interpret the protocol in order to make those details explicit in the LySa specification. Using the same first message as an example, when Bob receives it ((A,B,M,A,B;y1)), it will check if A is Alice's address, if B is its own address and if M is equal to the session running serial. Furthermore, it will assign the last element of the message sent by Alice, which is the encrypted block {M,A,B,NA}:KA, to the variable y1. In fact, Bob cannot decrypt that block since he does not have the key that was used for its encryption.

Another point to consider is that, regardless of what the elements of the messages are, LySa requires that the address of the sender and the address of the receiver are the first two elements of the sent and received messages. This is easily solved by tagging the messages from the pseudo-code description with these two elements in the LySa specification.

The full LySa specification of the Otway-Rees protocol can be seen in Figure 3.

3.2 ANTLR

ANTLR, which stands for "ANother Tool for Language Recognition", is a framework for generating recognisers, interpreters, compilers and translators based on grammatical descriptions. Besides providing support for building lexers and parsers, it also supports tree construction and tree walking. Parsers can generate

```
(new M)(  (new KA)(  (new KB)(

/* Initiator - A */
((new NA)
 <A,B,M,A,B,{M,A,B,NA}:KA>.          //line 1, send with encr.
 (B,A,M;x1).                         //line 4, receive
 decrypt x1 as {NA;xk}:KA in         //line 4, decryption
 (B,A;x2).                           //line 5, receive
 decrypt x2 as {;xmsg}:xk in         //line 5, decryption
 0)                 //termination
|

/* Responder - B */
((new NB)
 (A,B,M,A,B;y1).                     //line 1, receive
 <B,S,M,A,B,y1,{M,A,B,NB}:KB>.       //line 2, send with encr.
 (S,B,M;y2,y3).                      //line 3, receive
 decrypt y3 as {NB;yk}:KB in         //line 3, decryption
 <B,A,M,y2>.                         //line 4, send
 (new MSG)                           //line 5, message creation
 <B,A,{MSG}:yk>.                     //line 5, send with encr.
 0)                 //termination
|

/* Server - S */
((B,S,M,A,B;z1,z2).                  //line 2, receive
 decrypt z1 as {M,A,B;zna}:KA in     //line 2, decryption
 decrypt z2 as {M,A,B;znb}:KB in     //line 2, decryption
 (new K)                             //line 3, new shared key
 <S,B,M,{zna,K}:KA,{znb,K}:KB>.      //line 3, send with encr.
 0)                 //termination
)))
```

Fig. 3. LySa specification of the Otway-Rees protocol. The comments refer to the lines on the specification.

Abstract Syntax Trees, which can be further analysed with Tree Parsers (also called Tree Walkers). The framework has a tight integration with StringTemplate [16], which makes it ideal for translating from one language to another. It is this feature that we use for generating C and Java code from LySa specifications.

After the protocol as been verified, the ANTLR-based code generator translates the specification to a programming language that can then be compiled and executed. The presented prototype implementation translates the specification to C code as well as to Java code. ANTLR to generates the Lexer, the Parser, and the Tree Walker that performs the actual translation.

The first phase of the translator transforms the LySa specification into an abstract syntax tree (AST), which is used as input for the code generation, performed by the Tree Walker. A node in the AST represents an action in the protocol, and one of its children represents the next action taken by the principal.

The Tree Walker traverses the AST and, using string templates, generates the code that corresponds to the AST structure, and consequently to the original LySa protocol specification. Using StringTemplate is not only of advantage when extending the translation to support more target languages, but it is also advantageous when generating code for the different actors in a protocol. Due to supporting inheritance, StringTemplate enables the specification of generic templates together with specific ones. The latter are used to define actor-specific code — which depends on the role of the actor in the protocol — and can be loaded individually when performing the translation.

Figure 4 shows a SEND block that belongs to the Otway-Rees AST generated after parsing the initiator code. This block corresponds to the sending of the first message of the protocol.

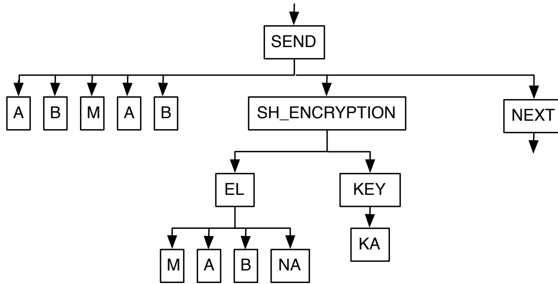

Fig. 4. Part of the AST subtree showing the send block of the initiator of the (first line of the) protocol specified in Figure 3

3.3 Retargeting the PiG

An important part regarding the implementation of the framework is the generation of the String Templates that will be used by the Tree Walker. For this prototype, the LySa specification has to be analysed and its main components identified, which need to have a direct correspondence to the main components of the String Templates.

The full version of the LySa specification language [10] contains artefacts that are not relevant for the actual protocol, but only help to increase the precision of the analysis tool. Taking this into consideration, the following main components can be identified:

– sending and receiving messages on the network,
– encryption and decryption (symmetric and asymmetric), and
– generation of fresh values (nonces, symmetric and asymmetric keys).

When implementing the translation from a specification language, identifying its main components is the first step because the templates are a direct implementation of the main components of the specification language. It is important to note that adding new languages as a target of the framework to an existing realisation of the PiG is very simple. As said before, it is only necessary to specify the StringTemplate for the main components.

An improvement on this part of the framework would be to make this translation provably correct, similar to the method presented by Pironti *et al.* [17]. They define a type-system and a translation function that allow to prove that the generated code simulated the process represented by the protocol specification, thus proving the correctness of the translation.

One of the main challenges when specifying template functions is that they need to fit together in the generated code, *e.g.*, how values are communicated between different functions. This can be tricky, since the same value may be used in different roles by different components, depending on the LySa specification.This resembles building the small parts of a puzzle before ensuring that the whole puzzle can be solved.

Another detail requiring special attention when specifying templates for a new language are variables reused between different blocks that are generated from the same template component. These cannot be be declared globally in each of the individual blocks, and consequently must be declared before any of the individual blocks and cleared before usage.

Last but not least, all the target languages must use the same format for message exchange, to allow interoperability. In our current prototype we apply the following straightforward format, which can be easily changed:

– firstly, the **number of elements** in the message;
– then, the **size of each element** in the message;
– finally, the **payload**: all the elements in the protocol message are concatenated without any separation between them.

As an illustrative example, if one wants to send a message with two elements, the first being "Hello" and the second being "Reader", the sent message would have the following format:

$$\text{“}2, 5, 6, HelloReader\text{”}$$

4 Related Work

Recently, a lot of work has been done in the automatic verification of security protocols as well as in the automatic translation from a protocol specification into a real programming language such as C, Java or F#. The goal of this paper is to present a framework that uses and implements both automatic processes—the verification and the translation—so there is an automatic and secure way from the writing of the specification protocol, over its verification, to its translation and execution. This is done in a way that enables the sharing of the specification (and consequently verification and translation) of protocols.

Possible Similar Tool Combinations. Several existing tools could be combined in order to realise a *Protocol Implementation Generator* implementation, providing the same functionality as described in the previous section.

The same high-level specification used in this paper could be used together with other tools. A protocol can be described in the LySa language, verified with LySatool and, with some extra annotations, can be translated using the YALT [18] tool, which automatically translates a LySa specification into Java code.

Another option would be to use Spi Calculus together with Spi2Java or S3A and Spi2F#.

After using the Spi Calculus to describe a security protocol, one could use Spi2Java [1] to verify and translate the description into a protocol implementation. Another option would be to use S3A [19] to verify the protocol specification and Spi2F# [20] that specification into a protocol implementation.

Using F# together with FS2PV [4] and ProVerif [21, 22] one could achieve a similar tool chain, although with a big difference. While our framework verifies the protocol specification and then translates it to some implementation language, this combination would translate the implementation of the protocol into a verifiable specification and only then would verify it. This setup, as already mentioned in Section 2, could also be seen as an extension/improvement to our framework. In this combination, the functional language F# would be used for protocol specification. Then, FS2PV would derive a formal model from that protocol code and symbolic libraries. FS2PV currently only supports a first-order subset of F#, with simple formal semantics facilitating model extraction, and primitives for communication and concurrency. The tool would translate the protocol implementation into π Calculus, which can be verified by ProVerif, an automatic cryptographic protocol verifier based on a simple representation of the protocol using Prolog rules.

Existing Frameworks. Some frameworks aim at combining protocol specification, verification, and implementation.

In the AGVI framework [23] the designer describes the security requirements and the system specification. The toolkit will attempt to find a protocol according to the demands. If found, it will translate it into Java. The SPEAR II Framework [24, 25] is a GUI-based framework that enables secure and efficient security protocol design and implementation, combining formal specification, security and performance analysis, meta-execution and automatic code generation. ACG-C# [2] automatically generates a C# implementation of a security protocol verified in Casper and FDR. Casper translates from high level to CSP, which can be verified using FDR, and translated by ACG-C#.

All these approaches differ significantly from the work presented in this paper. For example, AGVI does not support a protocol specification, but only receives the security requirements, and SPEAR II receives the protocol specification in a GUI environment, which hinders automating the implementation generation.

Furthermore, none of these frameworks offer support for sharing the protocol specification, making it impossible to rapidly enable two hosts to share the same protocol and to spread new protocols.

Last but not least, Kiyomoto *et al.* [3] present a tool that translates a high-level XML protocol specification into C, without any verification of the protocol specification.

5 Conclusions and Future Work

In this work we present a new approach to securing the communication in scenarios where partners do not initially share a protocol. This is especially important for the kind of networked applications we are relying on today, where the location of data is mostly hidden from users.

The *Protocol Implementation Generator* allows communication partners to exchange protocol specifications that can be verified and implemented on the fly; both the verification and the implementation, or code generation, are based on the same formal specification of the protocol, resulting in a direct link between the two.

We have implemented and presented a prototype realisation of PiG based on the process calculus LySa, its verifier the LySatool, and a standard code generation tool, ANTLR, which was set to generate C and Java code. The same functionality can be achieved with other combinations, as long as they share the protocol specification formalism.

We are currently investigating several extensions of the presented framework; we are investigating how to use proof-carrying code techniques [26] or lightweight verification [27] to avoid a full re-analysis of the specification before code generation. We are also interested in combining our approach with techniques that extract protocol specifications from implementations. This would allow to perform sanity checks by comparing the specification extracted from the generated implementation with the original specification.

Another thread of future work has to do with the security properties that are used by the verification tool of our framework. In the current version of our implementation, the security properties are implicit in the used tool (LySatool). A way of extending this version would be to enable the principals of the framework to negotiate security properties as part of the initial phase. Another possible way of approaching this would be to automate the download of general security properties from a set of trusted servers. With this, the PiG principals would have updated security properties that they would use for protocol verification.

The ideas behind PiG are being extended, and will be used to develop a framework for Service Oriented Systems, composed of different levels of abstraction, that includes verification (with different tools) and translation (into different languages) of abstractly specified Service Oriented Systems.

Finally, a word of warning seems in place. Approaches like PiG allow to add new protocols on the fly, and this might seem like a well-suited technique to updating large parts of a network by feeding newly designed protocols using a

framework like ours. However, the underlying automatism also allows to exploit shortcomings in the used tools to distribute a protocol that is known to pass verification but to result in faulty implementations. How to mitigate this threat remains a topic for future work.

References

1. Pozza, D., Sisto, R., Durante, L.: Spi2java: automatic cryptographic protocol java code generation from spi calculus. In: 18th International Conference on Advanced Information Networking and Applications, AINA 2004, vol. 1, pp. 400–405 (2004)
2. Jeon, C., Kim, I., Choi, J.: Automatic generation of the C# code for security protocols verified with casper/FDR. In: Proc. IEEE Int. Conf. on Advanced Inf. Networking and Applications (AINA), Taipei, Taiwan (2005)
3. Kiyomoto, S., Ota, H., Tanaka, T.: A security protocol compiler generating c source codes. In: 2008 International Conference on Information Security and Assurance (ISA 2008), pp. 20–25 (2008)
4. Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified interoperable implementations of security protocols. ACM Transactions on Programming Languages and Systems (TOPLAS) 31(1), 5 (2008)
5. Hickman, K., Elgamal, T.: The SSL protocol. Netscape Communications Corp. (1995)
6. Frier, A., Karlton, P., Kocher, P.: The SSL 3.0 protocol. Netscape Communications Corp. 18 (1996)
7. Quaresma, J.: A protocol implementation generator. Master's thesis, Kgs. Lyngby, Denmark (2010)
8. Buchholtz, M.: User's Guide for the LySatool version 2.01. DTU (April 2005)
9. Necula, G.C.: Proof-carrying code. In: POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 106–119. ACM, New York (1997)
10. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.: Static validation of security protocols. Journal of Computer Security 13(3), 347–390 (2005)
11. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf (2007)
12. Otway, D., Rees, O.: Efficient and timely mutual authentication. Operating Systems Review 21(1), 8–10 (1987)
13. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Information and Computation 148(1), 1–70 (1999)
14. Dolev, D., Yao, A.C.: On the security of public key protocols. In: Annual IEEE Symposium on Foundations of Computer Science, pp. 350–357 (1981)
15. Nielson, F., Riis Nielson, H., Sun, H., Buchholtz, M., Rydhof Hansen, R., Pilegaard, H., Seidl, H.: The Succinct Solver Suite. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 251–265. Springer, Heidelberg (2004)
16. Parr, T.: Stringtemplate documentation (May 2009), `http://www.antlr.org/wiki/display/ST/StringTemplate+Documentation`
17. Pironti, A., Sisto, R.: Provably correct java implementations of spi calculus security protocols specifications. Computers & Security 29(3), 302–314 (2010); Special issue on software engineering for secure systems
18. Vind, S., Vildhøj, H.W.: Secure protocol implementation with lysa. Bachelor's Thesis, DTU (2009)

19. Durante, L., Sisto, R., Valenzano, A.: Automatic testing equivalence verification of spi calculus specifications. ACM Trans. Softw. Eng. Methodol. 12(2), 222–284 (2003)
20. Tarrach, T.: Spi2f# – a prototype code generator for security protocols. Master's thesis, Saarland University (2008)
21. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proceedings of 14th IEEE Computer Security Foundations Workshop, pp. 82–96 (2001)
22. Abadi, M., Blanchet, B.: Analyzing security protocols with secrecy types and logic programs. Journal of the ACM (JACM) 52(1), 102–146 (2005)
23. Song, D., Perrig, A., Phan, D.: Agvi - Automatic Generation, Verification, and Implementation of Security Protocols. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 241–245. Springer, Heidelberg (2001)
24. Saul, E., Hutchison, A.: SPEAR II-The Security Protocol Engineering and Analysis Resource (1999)
25. Lukell, S., Veldman, C., Hutchison, A.: Automated attack analysis and code generation in a unified, multi-dimensional security protocol engineering framework. Comp. Science Hon (2002)
26. Necula, G.C.: Proof-carrying code. In: Conference Record of the Annual ACM Symposium on Principles of Programming Languages, pp. 106–119 (1997)
27. Rose, E.: Lightweight bytecode verification. Journal of Automated Reasoning 31(3-4), 303–334 (2003)