A Domain-Specific Language for Cryptographic Protocols based on Streams ⋆

Jan Jürjens

Department of Computing, The Open University, GB

Abstract

Developing security-critical systems is difficult and there are many well-known examples of security weaknesses exploited in practice. Thus a sound methodology supporting secure systems development is urgently needed. In particular, an important missing link in the construction of secure systems is finding a practical way to create reliably secure crypto protocol implementations. We present an approach that aims to address this need by making use of a domain-specific language for crypto protocol implementations. One can use this language to construct a compact and precise yet executable representation of a cryptographic protocol. This high-level program can be verified against the security goals using automated theorem provers for first order logic. One can then use it to provide assurance for legacy implementations of crypto protocols by generating test-cases.

1 Introduction

Modern society and modern economies rely on infrastructures for communication, finance, energy distribution, and transportation. These infrastructures depend increasingly on networked information systems. Attacks against these systems can threaten the economical or even physical well-being of people and organizations. Due to the widespread interconnection of information systems, attacks can be waged anonymously and from a safe distance. Many security incidents have been reported, sometimes with potentially quite severe consequences. Any support to aid secure systems development is thus dearly needed.

⋆ This work was partially funded by the Royal Society within the project Model-based Formal Security Analysis of Crypto-protocol Implementations and partly performed while the author was visiting Software & Systems Engineering, TU Munich.

Email address: `http://www.jurjens.de/jan` (Jan Jürjens).

In particular, it would be desirable to consider security aspects already in the design phase, before a system is actually implemented, since removing security flaws in the design phase saves cost and time.

With respect to crypto-based software (such as crypto-protocols or software somehow making use of cryptographic signatures), a lot of very successful work has been done to formally analyze abstract specifications of these protocols for security design weaknesses. What is still largely missing is an approach which analyzes implementations of crypto-based systems for security weaknesses. This is necessitated by the fact that so far, crypto-based software is usually not generated automatically from formal specifications. Thus, even where the corresponding specifications are formally verified, the implementations may still contain vulnerabilities related to the insecure use of cryptographic algorithms. An example for a crypto-protocol whose design had been formally verified for security and whose implementation was later found to contain a weakness with respect to its use of cryptographic algorithms can be found in [34].

Towards this goal, we present an approach that aims to address this need by making use of a domain-specific language for crypto protocol implementations. One can use this language to construct a compact and precise yet executable representation of a cryptographic protocol. This high-level program can be verified against the security goals using automated theorem provers for first order logic. One can then use it to provide assurance for legacy implementations of crypto protocols by generating test-cases. As a running example, we use a variant of the Internet security protocol TLS.

The current work is part of a wider initiative to provide methods for model-based security engineering of software, initially mostly targeted to the model level (see [18,19,22]). The current work tries to bridge the gap to the implementation level by offering to use a domain-specific language for crypto-protocol which is executable (and therefore directly usable e.g. to generate test-sequences for legacy implementations of crypto protocols), but abstract enough for an efficient, automated formal verification. Since the language is based on the paradigm of stream-processing functions, a number of useful techniques are available from that paradigm which we use as well to reason about programs on the language level, such as refinement and rely-guarantee properties of programs in the language (see e.g. [9,8] for a general introduction to these concepts in the context of stream-processing functions). We show that these notions preserve the security properties considered in this paper.

Sect. 2 presents the domain-specific language for cryptographic protocols. Sect. 3 explains how one can formalize the security property of data secrecy in this context. Sect. 4 shows how to make use of the various notions of refinement available from the paradigm of stream-processing functions here. Sect. 5

introduces the running application of this paper, the variant of the security protocol TLS proposed in [3]. Sect. 6 explains how to construct a secure channel making use of the DSL at the hand of the TLS variant. Sect. 7 explains how to translate DSL programs to first-order logic formulas. In Sect. 7.1, we explain the translation at the hand of a variant of the Internet protocol TLS. Sect. 7.2 explains how to perform the security analysis using the automated theorem prover. Sect. 8 explains how we generate concrete test cases. We close with comparisons to related work and a discussion of our work.

2 A Domain-Specific Language for Cryptographic Protocols

In this section, we introduce the domain-specific language for cryptographic protocols, which is based on the stream-processing function paradigm (cf. [9] for a general introduction).

Specifically, we consider concurrently executing processes interacting by transmitting sequences of data values over unidirectional FIFO communication channels. Communication is asynchronous in the sense that transmission of a value cannot be prevented by the receiver (note that one may implement synchronous communication using handshakes [9]).

Processes are collections of programs that communicate through channels, with the constraint that for each of its output channels c a given process P contains exactly one program p_c that outputs on c. This program p_c may take input from any of P's input channels. Intuitively, the program is a description of a value to be output on the channel c in round $n+1$, computed from values found on channels in round n. Local state can be maintained through the use of feedback channels, and used for iteration (for instance, for coding *while* loops).

Note that all programs (one for each output channel) that are used to define a process are executed synchronously while the process is running, in the sense that they progress in lockstep (i.e., at each time interval, each program performs exactly one computation step, all in parallel). In contrast, different processes that are executed concurrently are executed such that they are linked to each other in an asynchronous way, in the sense that if one process communicates with another, this is done in an asynchronous fashion (i.e., the sender of a message proceeds with its own execution independently of whether or when the receiver processes that message).

We assume disjoint sets $\mathcal{D}$ of data values, **Secret** of unguessable values (such as "nonces" – freshly generated values supposed to be used only once –, other random values, session keys, or similar), **Keys** of keys, Channels of channels

$E ::=$	expression
d	data value ($d \in \mathcal{D}$)
N	unguessable value ($N \in \mathbf{Secret}$)
K	key ($K \in \mathbf{Keys}$)
$\mathsf{inp}(c)$	input on channel c ($c \in \mathsf{Channels}$)
x	variable ($x \in \mathbf{Var}$)
$E_1 :: E_2$	concatenation
$\{E\}_e$	encryption ($e \in \mathbf{Enc}$)
$\mathcal{D}ec_e(E)$	decryption ($e \in \mathbf{Enc}$)
$\mathcal{S}ign_e(E)$	signature creation ($e \in \mathbf{Enc}$)
$\mathcal{E}xt_e(E)$	signature extraction ($e \in \mathbf{Enc}$)

Fig. 1. Grammar for simple expressions in the Domain-Specific Language

and **Var** of variables. Write $\mathbf{Enc} \stackrel{\text{def}}{=} \mathbf{Keys} \cup \mathsf{Channels} \cup \mathbf{Var}$ for the set of *encryptors* that may be used for encryption or decryption. The values communicated over channels are formal *expressions* built from variables, values on input channels, and data values using concatenation. Precisely, the set **Exp** of expressions contains the empty expression ε and the non-empty expressions generated by the grammar given in Figure 1.

An occurrence of a channel name c refers to the value found on c at the previous instant. The empty expression ε denotes absence of output on a channel at a given point in time. We write **CExp** for the set of *closed* expressions (those containing no subterms in $\mathbf{Var} \cup \mathsf{Channels}$). We write the decryption key corresponding to an encryption key K as K^{-1}. In the case of asymmetric encryption, the encryption key K is public, and K^{-1} secret. For symmetric encryption, K and K^{-1} may coincide. We assume $\mathcal{D}ec_{K^{-1}}(\{E\}_K) = E$ for all $E \in \mathbf{Exp}, K, K^{-1} \in \mathbf{Keys}$ and $\mathcal{E}xt_K(\mathcal{S}ign_{K^{-1}}(E)) = E$ for all $E \in \mathbf{Exp}, K, K^{-1} \in \mathbf{Keys}$ (and we assume that no other equations except those following from these hold, unless stated otherwise).

Programs in the DSL are then defined by the grammar given in Figure 2. Note that the grammar in particular includes a non-deterministic choice operator. This allows one to use the DSL notation also for specifications which admit underspecification, as will be explained in later sections.

In the DSL grammar, variables are introduced in case constructs, which determine their values. The first case construct tests whether E is a key; if so, p is executed, otherwise p'. The second case construct tests whether E is a list

with head x and tail y; if so, p is evaluated, using the actual values of x, y; if not, p' is evaluated. In the second case construct, x and y are bound variables. A program is *closed* if it contains no unbound variables. *while* loops can be coded using feedback channels.

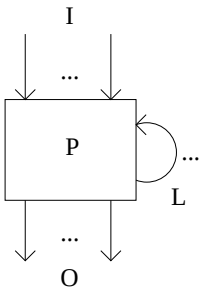
From each assignment of expressions to channel names $c \in \mathsf{Channels}$ appearing in a program p (called its *input channels*), p computes an output expression.

For simplification we assume that in the following all programs are *well-formed* in the sense that each encryption $\{E\}_e$ and decryption $\mathcal{D}ec_e(E)$ appears as part of p in a *case* E' *of key do* p *else* p' construct (unless $e \in$ **Keys**), to ensure that only keys are used to encrypt or decrypt. It is straightforward to enforce this using a type system.

Example The program *case* c *of key do* $\{d\}_c$ *else* ε outputs the value received at channel d encrypted under the value received on channel c if that value is a key, otherwise it outputs ε.

A *process* is of the form $P = (I, O, L, (p_c)_{c \in O \cup L})$ where

- $I \subseteq \mathsf{Channels}$ is called the set of its *input channels* and
- $O \subseteq \mathsf{Channels}$ the set of its *output channels*,



and where for each $c \in \tilde{O} \stackrel{\text{def}}{=} O \cup L$, p_c is a closed program with input channels in $\tilde{I} \stackrel{\text{def}}{=} I \cup L$ (where $L \subseteq \mathsf{Channels}$ is called the set of *local channels*). From inputs on the channels in $\tilde{I}$ at a given point in time, p_c computes the output on the channel c.

We write I_P, O_P and L_P for the sets of input, output and local channels of P, $K_P \subseteq$ **Keys** for the set of private keys and $S_P \subseteq$ **Secret** for the set of

$p ::=$	programs
E	output expression ($E \in$ **Exp**)
either p *or* p'	nondeterministic branching
if $E = E'$ *then* p *else* p'	conditional ($E, E' \in$ **Exp**)
case E *of key do* p *else* p'	determine if E is a key ($E \in$ **Exp**)
case E *of* $x :: y$ *do* p *else* p'	break up list into head::tail ($E \in$ **Exp**)

Fig. 2. Grammar for programs in the Domain-Specific Language

unguessable values (such as nonces) occurring in P. We assume that different processes have disjoint sets of local channels, keys and secrets. Local channels are used to store local state between the execution rounds.

2.1 Stream-processing functions

Since we aim to assign a formal interpretation to programs in our DSL using stream-processing functions, we recall the definitions of streams and stream-processing functions from [7,9] in this subsection.

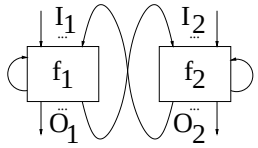
We write $\mathbf{Stream}_C \stackrel{\text{def}}{=} (\mathbf{CExp}^\infty)^C$ (where $C \subseteq \mathsf{Channels}$) for the set of C-indexed tuples of (finite or infinite) sequences of closed expressions. The elements of this set are called *streams*, specifically *input streams* (resp. *output streams*) if C denotes the set of non-local input (resp. output) channels of a process P. Each stream $\vec{s} \in \mathbf{Stream}_C$ consists of components $\vec{s}(c)$ (for each $c \in C$) that denote the sequence of expressions appearing at the channel c. The n^{th} element x_n in such a sequence $\vec{s}(c) = (x_1, x_2, x_3, \ldots, x_n, \ldots)$ consisting of expressions $x_1, x_2, x_3, \ldots, x_n, \ldots$ is the expression appearing at time $t = n$.

A function $f : \mathbf{Stream}_I \to \mathcal{P}(\mathbf{Stream}_O)$ from streams to sets of streams is called a *stream-processing function*.

The composition of two stream-processing functions $f_i : \mathbf{Stream}_{I_i} \to \mathcal{P}(\mathbf{Stream}_{O_i})$ $(i = 1, 2)$ with $O_1 \cap O_2 = \emptyset$ is defined as

$$f_1 \otimes f_2 : \mathbf{Stream}_I \to \mathcal{P}(\mathbf{Stream}_O)$$

(with $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$, $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$).



where $f_1 \otimes f_2(\vec{s}) \stackrel{\text{def}}{=} \{\vec{t}\restriction_O : \vec{t}\restriction_I = \vec{s}\restriction_I \wedge \vec{t}\restriction_{O_i} \in f_i(\vec{s}\restriction_{I_i}) \ (i = 1, 2)\}$ (where $\vec{t}$ ranges over $\mathbf{Stream}_{I \cup O}$). For $\vec{t} \in \mathbf{Stream}_C$ and $C' \subseteq C$, the restriction $\vec{t}\restriction_{C'} \in \mathbf{Stream}_{C'}$ is defined by $\vec{t}\restriction_{C'}(c) = \vec{t}(c)$ for each $c \in C'$. Since the operator $\otimes$ is associative and commutative [9], we can define a generalised composition operator $\bigotimes_{i \in I} f_i$ for a set $\{f_i : i \in I\}$ of stream-processing functions.

Example If $f : \mathbf{Stream}_{\{a\}} \to \mathcal{P}(\mathbf{Stream}_{\{b\}})$, $f(\vec{s}) \stackrel{\text{def}}{=} \{0.\vec{s}, 1.\vec{s}\}$, is the stream-processing function with input channel a and output channel b that outputs the input stream prefixed with either 0 or 1, and $g : \mathbf{Stream}_{\{b\}} \to \mathcal{P}(\mathbf{Stream}_{\{c\}})$, $g(\vec{s}) \stackrel{\text{def}}{=} \{0.\vec{s}, 1.\vec{s}\}$, the function with input (resp. output) channel b (resp. c) that does the same, then the composition $f \otimes g : \mathbf{Stream}_{\{a\}} \to \mathcal{P}(\mathbf{Stream}_{\{c\}})$, $f \otimes g(\vec{s}) = \{0.0.\vec{s}, 0.1.\vec{s}, 1.0.\vec{s}, 1.1.\vec{s}\}$, outputs the input stream prefixed with either of the 2-element streams 0.0, 0.1, 1.0 or 1.1.

$[E](\vec{M}) = \{E(\vec{M})\}$	where $E \in \mathbf{Exp}$
$[\mathit{either}\ p\ \mathit{or}\ p'](\vec{M}) = [p](\vec{M}) \cup [p'](\vec{M})$	
$[\mathit{if}\ E = E'\ \mathit{then}\ p\ \mathit{else}\ p'](\vec{M}) = [p](\vec{M})$	if $[E](\vec{M}) = [E'](\vec{M})$
$[\mathit{if}\ E = E'\ \mathit{then}\ p\ \mathit{else}\ p'](\vec{M}) = [p'](\vec{M})$	if $[E](\vec{M}) \neq [E'](\vec{M})$
$[\mathit{case}\ E\ \mathit{of}\ \mathit{key}\ \mathit{do}\ p\ \mathit{else}\ p'](\vec{M}) = [p](\vec{M})$	if $[E](\vec{M}) \in \mathbf{Keys}$
$[\mathit{case}\ E\ \mathit{of}\ \mathit{key}\ \mathit{do}\ p\ \mathit{else}\ p'](\vec{M}) = [p'](\vec{M})$	if $[E](\vec{M}) \notin \mathbf{Keys}$
$[\mathit{case}\ E\ \mathit{of}\ x :: y\ \mathit{do}\ p\ \mathit{else}\ p'](\vec{M}) = [p[h/x, t/y]](\vec{M})$	
if $[E](\vec{M}) = h :: t$ where $h \neq \varepsilon$ and h is not of the form $h_1 :: h_2$ for $h_1, h_2 \neq \varepsilon$	
$[\mathit{case}\ E\ \mathit{of}\ x :: y\ \mathit{do}\ p\ \mathit{else}\ p'](\vec{M}) = [p'](\vec{M})$	if $[E](\vec{M}) = \varepsilon$

Fig. 3. Definition of $[p](\vec{M})$.

2.2 Associating a stream-processing function to a process

A process $P = (I, O, L, (p_c)_{c \in O})$ is modelled by a stream-processing function $[\![P]\!] : \mathbf{Stream}_I \to \mathcal{P}(\mathbf{Stream}_O)$ from input streams to sets of output streams.

For honest processes P, $[\![P]\!]$ is by construction *strictly causal*, which means that the $(n+1)^{st}$ expression in any output sequence depends only on the first n input expressions. As pointed out in [33], adversaries can not be assumed to behave causally in that strict sense: Given a worst-case scenario, the speed of the adversary machine may exceed the speed of the "honest" machine to an extent that the adversary behavior appears instantaneous from the point of view of the honest machine. Therefore for an adversary A we need a slightly different interpretation $[\![A]\!]_r$ (called *sometimes rushing adversaries* in [33]), which we will define further below at the end of this section.

For any closed program p with input channels in $\tilde{I} \stackrel{\text{def}}{=} I \cup L$ and any $\tilde{I}$-indexed tuple of closed expressions $\vec{M} \in \mathbf{CExp}^{\tilde{I}}$ we define a set of expressions $[p](\vec{M}) \in \mathcal{P}(\mathbf{CExp})$ in Fig. 3, so that $[p](\vec{M})$ is the expression that results from running p once, when the channels have the initial values given in $\vec{M}$.

We write $E(\vec{M})$ for the result of substituting each occurrence of $c \in \tilde{I}$ in E by $\vec{M}(c)$ and $p[E/x]$ for the outcome of replacing each free occurrence of x in process P with the term E, renaming variables to avoid capture.

Then any program p_c (for $c \in \mathsf{Channels}$) defines a strictly causal stream-processing function $[p_c] : \mathbf{Stream}_{\tilde{I}} \to \mathcal{P}(\mathbf{Stream}_{\{c\}})$ as follows. Given $\vec{s} \in \mathbf{Stream}_{\tilde{I}}$, let $[p_c](\vec{s})$ consist of those $\vec{t} \in \mathbf{Stream}_{\{c\}}$ such that

- $\vec{t}_0 \in [p_c](\varepsilon, \ldots, \varepsilon)$
- $\vec{t}_{n+1} \in [p_c](\vec{s}_n)$ for each $n \in \mathbb{N}$.

Finally, a process $P = (I, O, L, (p_c)_{c \in \tilde{O}})$ (where $\tilde{O} \stackrel{\text{def}}{=} O \cup L$) is interpreted as the composition $[\![P]\!] \stackrel{\text{def}}{=} \bigotimes_{c \in \tilde{O}} [p_c]$.

Similarly, any p_c (with $c \in \mathsf{Channels}$) defines a stream-processing function $[p_c]_r : \mathbf{Stream}_{\tilde{I}} \to \mathcal{P}(\mathbf{Stream}_{\{c\}})$ as follows, which is not strictly causal. Given $\vec{s} \in \mathbf{Stream}_{\tilde{I}}$ and $c \notin \tilde{I}$, let $[p_c]_r(\vec{s})$ consist of those $\vec{t} \in \mathbf{Stream}_{\{c\}}$ such that $\vec{t_n} \in [p_c]_r(\vec{s_n})$ for each $n \in \mathbb{N}$.

An adversary $A = (I, O, L, (p_c)_{c \in \tilde{O}})$ is then interpreted as the composition:

$$[\![A]\!]_r \stackrel{\text{def}}{=} \bigotimes_{c \in O} [p_c]_r \otimes \bigotimes_{l \in \tilde{O} \setminus O} [p_l].$$

Thus the programs with outputs on the non-local channels are defined to be *rushing*. Note that at the local channels an adversary still shows strictly causal behaviour (which ensures that the above definition is actually well-defined). This does not mean that it is limited in power, however: The adversary is able to arbitrarily delay the further execution of a protocol (by waiting with forwarding a message to the intended receipient for an arbitrary number of time steps) which allows him to perform further computations on the data he already has. He is also able to store arbitrary amounts of data in any of his local channels (by building up arbitrarily large expressions).

Examples

- $[\mathit{if}\ \mathcal{D}ec_{K'}(\{0\}_K) = 0\ \mathit{then}\ 0\ \mathit{else}\ 1](\vec{s}) = (0, 0, 0, \ldots)$ iff $K = K'$
- Assume that l, o, and i are channel names. For the process P with $I_P = \{i\}$, $O_P = \{o\}$ and $L_P = \{l\}$ and with $p_l \stackrel{\text{def}}{=} \mathsf{inp}(l) :: \mathsf{inp}(i)$ (that is, the concatenation of the input values received at the channels l and i) and $p_o \stackrel{\text{def}}{=} \mathsf{inp}(l) :: \mathsf{inp}(i)$ we have $[\![P]\!](\vec{s}) = \{(\varepsilon, \vec{s}_0, \vec{s}_0 :: \vec{s}_1, \vec{s}_0 :: \vec{s}_1 :: \vec{s}_2, \ldots)\}$ and $[\![P]\!]_r(\vec{s}) = \{(\vec{s}_0, \vec{s}_0 :: \vec{s}_1, \vec{s}_0 :: \vec{s}_1 :: \vec{s}_2, \ldots)\}$.

3 Secrecy

We say that a stream-processing function $f : \mathbf{Stream}_\emptyset \to \mathcal{P}(\mathbf{Stream}_O)$ *may eventually output* an expression $E \in \mathbf{CExp}$ if there exists a stream $\vec{t} \in f(*)$ (where $*$ denotes the sole element in $\mathbf{Stream}_\emptyset$), a channel $c \in O$ and an index $j \in \mathbb{N}$ such that $(\vec{t}(c))_j = E$.

Definition 1 We say that a process P *leaks* a secret $m \in \mathbf{Secret} \cup \mathbf{Keys}$ if there is a process A with $I_A \subseteq O_P$, $I_P \subseteq O_A$ and $m \notin S_A \cup K_A$ such that $[\![P]\!] \otimes [\![A]\!]_r$ may eventually output m. Otherwise we say that P *preserves the secrecy of* m.

The idea of this definition is that P preserves the secrecy of m if no adversary can find out m in interaction with P. Note that for a process A to be able to output the secret m on its output channel, it needs to "find it out" first. Conversely, if there is a process A which is able to "find out" m, there is a process A' which does output m on its output channel. In that sense, "finding out" the secret and sending it out are equivalent (and we choose to formalize secrecy by referring to an output of A because this simplifies the definition). In our formulation m is either an atomic secret value or a key. This is sufficient in practice, since the secrecy of a compound expression can be reduced to the secrecy of a key or an atomic secret value [1]: Depending on the circumstances, one can define the compound expression to be secret if at least one, or alternatively if all its parts remain secret. Because we want to be able to use both alternatives, we only give a definition for atomic values here and leave it up to the user to apply it to compound values as may seem fit.

Note, also, that this definition is intended to be used with "complete" specifications, as opposed to "partial" specifications which might for example only define one role in a protocol. There may also be the possibility of defining secrecy for such "partial" specifications. However, to be useful this would raise the issue of the preservation of the notion of secrecy by composition. Although that would be a very interesting question, it is beyond the scope of the current paper, and therefore here we only consider secrecy for "complete" specifications which model an entire protocol, in the sense that all non-local channels are accessible to the adversary.

Note, however, that the decision to focus on "complete" specifications in this paper only restricts the process of constructing and verifying specifications, in the sense that the verification can only be done when the specification is complete, at least at a high level of abstraction (cf. later sections for how to use refinement techniques to add more details). It does not in itself restrict the systems and the security properties that can be analyzed. For example, it is still possible to consider parallel sessions of a protocol for the purpose of the security analysis, whose number is bounded by an arbitrary number n which needs to be chosen before performing the analysis. To do this, one parameterizes the protocol specification with respect to a value which uniquely identifies a session (such as a session number of session key), and then consider the composition of these parameterized specifications.

It is also possible to analyze a specification against an adversary that may take a role in the given protocol: The behavior of the component A in the above definition is not constrained. In particular, it can execute any role in any protocol, assuming it has the knowledge of the relevant data values (such as keys) that are needed to do so. This knowledge can be provided to the adversary using suitable input channels. This is demonstrated at the hand of the example of the TLS variant given further below.

Examples

- $p \stackrel{\text{def}}{=} \{m\}_K :: K$ does not preserve the secrecy of m or K, but $p \stackrel{\text{def}}{=} \{m\}_K$ does.
- $p_l \stackrel{\text{def}}{=} \mathit{case}\ \mathsf{inp}(c)\ \mathit{of\ key\ do}\ \{m\}_{\mathsf{inp}(c)}\ \mathit{else}\ \varepsilon$ (where $c \in \mathsf{Channels}$) does not preserve the secrecy of m, but $P \stackrel{\text{def}}{=} (\{c\}, \{e\}, \{l\}, (p_l, p_e))$ (where $p_e \stackrel{\text{def}}{=} \{\mathsf{inp}(l)\}_K$) does.

Since our language is based on the paradigm of stream-processing functions, a number of useful techniques are available from that paradigm which we use as well to reason about programs on the language level, such as refinement and rely-guarantee properties of programs in the language (see e.g. [9] for a general introduction to these concepts in the context of stream-processing functions). We start with these by defining a rely-guarantee condition for secrecy.

Given a relation $C \subseteq \mathbf{Stream}_O \times \mathbf{Stream}_I$ and a process A with $O \subseteq I_A$ and $I \subseteq O_A$ we say that A *fulfils* C if for every $\vec{s} \in \mathbf{Stream}_{I_A}$ and every $\vec{t} \in [\![A]\!](\vec{s})$, we have $(\vec{s}\downarrow_O, \vec{t}\downarrow_I) \in C$.

Definition 2 Given a relation $C \subseteq \mathbf{Stream}_{O_P} \times \mathbf{Stream}_{I_P}$ from output streams of a process P to input streams of P, we say that P *leaks* m *assuming* C (for $m \in \mathbf{Secret} \cup \mathbf{Keys}$) if there exists a process A with $m \notin S_A \cup K_A$ that fulfils C and such that $[\![P]\!] \otimes [\![A]\!]_r$ may eventually output m.
Otherwise P *preserves the secrecy of* m *assuming* C.

This definition is useful if P is a component of a larger system S that is assumed to fulfil the rely-condition, or if the adversary is assumed to be unable to violate it.

Example $p \stackrel{\text{def}}{=} \mathit{if}\ \mathsf{inp}(c) = \mathbf{password}\ \mathit{then}\ \mathbf{secret}\ \mathit{else}\ \varepsilon$ preserves the secrecy of *secret* assuming $C = \{(\vec{t}, \vec{s}) : \forall n. \vec{s}_n \neq \mathbf{password}\}$.

4 Refinement

Another family of important techniques available within the paradigm of stream-processing functions, besides the notion of rely-guarantee properties used above, is that of different kinds of refinement defined in [9]. We define these notions of refinement and exhibit conditions under which they preserve our proposed secrecy properties.

4.1 Property refinement

Definition 3 For processes P and P' with $I_P = I_{P'}$ and $O_P = O_{P'}$ we define $P \rightsquigarrow P'$ if for each $\vec{s} \in \mathbf{Stream}_{I_P}$, $[\![P]\!](\vec{s}) \supseteq [\![P']\!](\vec{s})$.

Example $(\mathit{either}\ p\ \mathit{or}\ q) \rightsquigarrow p$ and $(\mathit{either}\ p\ \mathit{or}\ q) \rightsquigarrow q$ for any programs p, q.

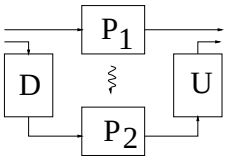
Theorem 1

- *If P preserves the secrecy of m and $P \rightsquigarrow P'$ then P' preserves the secrecy of m.*
- *If P preserves the secrecy of m assuming C (for any $C \subseteq \mathbf{Stream}_{O_P} \times \mathbf{Stream}_{I_P}$) and $P \rightsquigarrow P'$ then P' preserves the secrecy of m assuming C.*

Proof The proof follows immediately from the definitions of secrecy and refinement, since secrecy is defined over the set of communicated values, and this set can only be reduced when using refinement.

4.2 Interface refinement

Definition 4 Let P_1, P_2, D and U be processes with $I_{P_1} = I_D$, $O_D = I_{P_2}$, $O_{P_2} = I_U$ and $O_U = O_{P_1}$.
We define $P_1 \overset{(D,U)}{\rightsquigarrow} P_2$ to hold if $P_1 \rightsquigarrow D \otimes P_2 \otimes U$.



Example Suppose we have

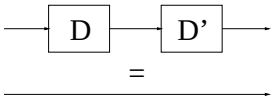
- $P_1 = (\{c\}, \{d\}, p_d \overset{\text{def}}{=} \mathit{if}\ \mathsf{inp}(c) = 1\ \mathit{then}\ 2\ \mathit{else}\ 3)$,
- $P_2 = (\{c'\}, \{d'\}, p_{d'} \overset{\text{def}}{=} \mathit{if}\ \mathsf{inp}(c') = 4\ \mathit{then}\ 5\ \mathit{else}\ 6)$,
- $D = (\{c\}, \{c'\}, p_{c'} \overset{\text{def}}{=} \mathit{if}\ \mathsf{inp}(c) = 1\ \mathit{then}\ 4\ \mathit{else}\ \varepsilon)$ and
- $U = (\{d'\}, \{d\}, p_d \overset{\text{def}}{=} \mathit{if}\ \mathsf{inp}(d') = 5\ \mathit{then}\ 2\ \mathit{else}\ 3$.

Then we have $P_1 \overset{(D,U)}{\rightsquigarrow} P_2$.

For the next preservation result we need the following concepts.

Given a stream $\vec{s} \in \mathbf{Stream}_X$ and a bijection $\iota : Y \to X$ we write $\vec{s}_\iota$ for the stream in $\mathbf{Stream}_Y$ obtained from $\vec{s}$ by renaming the channel names using ι: $\vec{s}_\iota(y) = \vec{s}(\iota(y))$.

Given processes D, D' with $O_D = I_{D'}$ and $O_{D'} \cap I_D = \emptyset$ and a bijection $\iota : O_{D'} \to I_D$ such that $[\![D]\!] \otimes [\![D']\!](\vec{s}) = \{\vec{s}_\iota\}$ for each $\vec{s} \in \mathbf{Stream}_{I_D}$, we say that D is a *left inverse* of D' and D' is a *right inverse* of D.



Example $p_d \stackrel{\text{def}}{=} 0 :: \mathsf{inp}(c)$ is a left inverse of $p_e \stackrel{\text{def}}{=} \mathit{case}\ \mathsf{inp}(c)\ \mathit{of}\ h :: t\ \mathit{do}\ t\ \mathit{else}\ \varepsilon$.

We write $S \circ R \stackrel{\text{def}}{=} \{(x,z) : \exists y.(x,y) \in R \wedge (y,z) \in S\}$ for the usual composition of relations R, S and generalize this to functions $f : X \to \mathcal{P}(Y)$ by viewing them as relations $f \subseteq X \times Y$.

Theorem 2 *Let P_1, P_2, D and U be processes with $I_{P_1} = I_D$, $O_D = I_{P_2}$, $O_{P_2} = I_U$ and $O_U = O_{P_1}$ and such that D has a left inverse D' and U a right inverse U'. Let $m \in (\mathbf{Secret} \cup \mathbf{Keys}) \setminus \bigcup_{Q \in \{D', U'\}} (S_Q \cup K_Q)$.*

- *If P_1 preserves the secrecy of m and $P_1 \overset{(D,U)}{\rightsquigarrow} P_2$ then P_2 preserves the secrecy of m.*
- *If P_1 preserves the secrecy of m assuming $C \subseteq \mathbf{Stream}_{O_{P_1}} \times \mathbf{Stream}_{I_{P_1}}$ and $P_1 \overset{(D,U)}{\rightsquigarrow} P_2$ then P_2 preserves the secrecy of m assuming $[\![U']\!] \circ C \circ [\![D']\!]$.*

Proof This statement follows directly from Theorem 1 and the definition of interface refinement.

4.3 Conditional refinement

Definition 5 Let P_1 and P_2 be processes with $I_{P_1} = I_{P_2}$ and $O_{P_1} = O_{P_2}$. We define $P_1 \rightsquigarrow_C P_2$ for a total relation $C \subseteq \mathbf{Stream}_{O_{P_1}} \times \mathbf{Stream}_{I_{P_1}}$ to hold if for each $\vec{s} \in \mathbf{Stream}_{I_{P_1}}$ and each $\vec{t} \in [\![P_2]\!]$, $(\vec{t}, \vec{s}) \in C$ implies $\vec{t} \in [\![P_1]\!]$.

Example $p \rightsquigarrow_C (\mathit{if}\ \mathsf{inp}(c) = \mathbf{emergency}\ \mathit{then}\ q\ \mathit{else}\ p)$ for $C = \{(\vec{t}, \vec{s}) : \forall n.\vec{s}_n \neq \mathbf{emergency}\}$.

Theorem 3
Given total relations $C, D \subseteq \mathbf{Stream}_{O_P} \times \mathbf{Stream}_{I_P}$ with $C \subseteq D$, if P preserves the secrecy of m assuming C and $P \rightsquigarrow_D P'$ then P' preserves the secrecy of m assuming C.

Proof Again, this statement follows directly from Theorem 1 and the definition of conditional refinement.

5 A Variant of the TLS Protocol

To demonstrate usability of our domain-specific language, we specify a variant of the handshake protocol of TLS [1] as proposed in [3] (note that this is not the variant of TLS in common use). To show applicability of our approach, we exhibit a security vulnerability, suggest a correction, and verify it. The goal of the protocol is to let a client send a secret over an untrusted communication link to a server in a way that provides secrecy and server authentication, by using symmetric session keys.

5.1 The Handshake Protocol

The central part of the specification of this protocol is shown in Fig. 4. The two protocol participants client and server are connected by an Internet connection. The value secret which is exchanged encrypted in the last message of the protocol is required to remain secret.

Depicted in Fig. 4, the protocol proceeds as we explain in the following. Here we assume that the set **Var** contains elements $\mathsf{arg}_{O,l,n}$ for each $O \in \mathsf{Obj}(D)$ and numbers l and n, representing the nth argument of the operation that is supposed to be the lth operation received by O according to the sequence diagram D.

[1] TLS (transport layer security) is the successor of the Internet security protocol SSL (secure sockets layer).

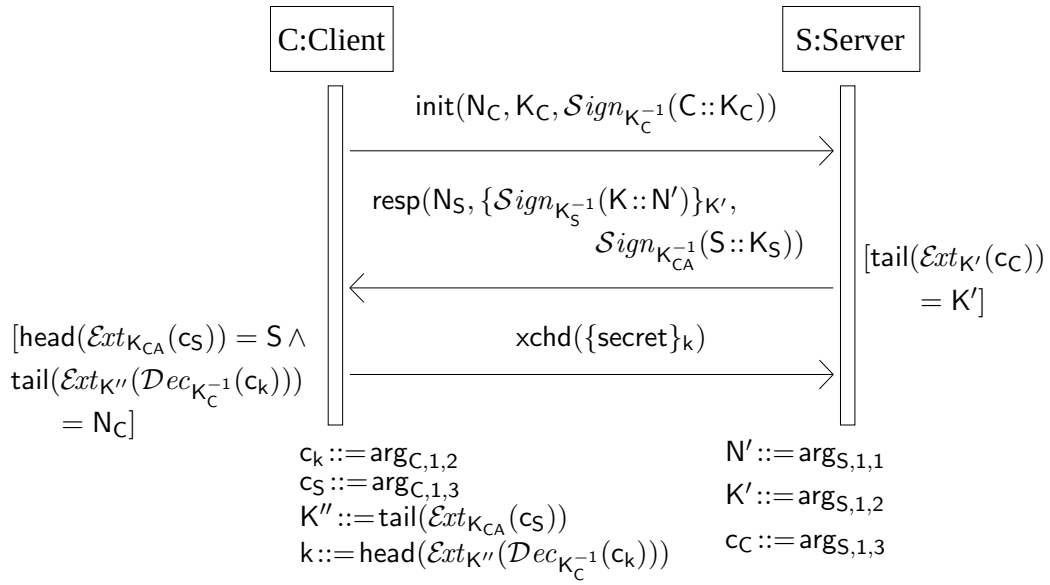


Fig. 4. Variant of the TLS handshake

The client C initiates the protocol by sending the message $\mathsf{init}(\mathsf{N_C}, \mathsf{K_C}, \mathcal{S}ign_{\mathsf{K_C}^{-1}}(\mathsf{C} :: \mathsf{K_C}))$ to the server S. Suppose that the condition $[\mathsf{tail}(\mathcal{E}xt_{\mathsf{K}'}(\mathsf{c_C}))=\mathsf{K}']$ holds, where $K' ::= \mathsf{arg}_{S,1,2}$ and $\mathsf{c_C} ::= \mathsf{arg}_{\mathsf{S},1,3}$. That is, the key K_C contained in the signature matches the one transmitted in the clear. In that case, S sends the message $\mathsf{resp}(\mathsf{N_S}, \{\mathcal{S}ign_{\mathsf{K_S}^{-1}}(\mathsf{K} :: \mathsf{N}')\}_{\mathsf{K}'}, \mathcal{S}ign_{K_{\mathsf{CA}}^{-1}}(S :: K_S))$ back to C (where $\mathsf{N}' ::= \mathsf{arg}_{\mathsf{S},1,1}$). Then if the condition

$$[\mathsf{head}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S}))=\mathsf{S} \wedge \mathsf{tail}(\mathcal{E}xt_{\mathsf{K}''}(\mathcal{D}ec_{\mathsf{K_C}^{-1}}(\mathsf{c_k})))=\mathsf{N_C}]$$

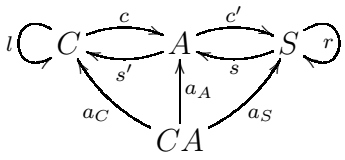
holds, where $\mathsf{c_k} ::= \mathsf{arg}_{\mathsf{C},1,2}$, $\mathsf{c_S} ::= \mathsf{arg}_{\mathsf{C},1,3}$, and $\mathsf{K}'' ::= \mathsf{tail}(\mathcal{E}xt_{K_{\mathsf{CA}}}(c_S))$ (that is, the certificate is actually for S and the correct nonce is returned), C sends $\mathsf{xchd}(\{\mathsf{s_i}\}_\mathsf{k})$ to S, where $\mathsf{k} ::= \mathsf{head}(\mathcal{E}xt_{\mathsf{K}''}(\mathcal{D}ec_{\mathsf{K_C}^{-1}}(\mathsf{c_k})))$. If any of the checks fail, the respective protocol participant stops the execution of the protocol.

The goal is thus to let a client C send a master secret $m \in \mathbf{Secret}$ to a server S in a way that provides confidentiality and server authentication.

The protocol uses both RSA encryption and signing. Thus in this and the following section we assume also the equation $\{\mathcal{D}ec_{K^{-1}}(E)\}_K = E$ to hold (for each $E \in \mathbf{Exp}$ and $K \in \mathbf{Keys}$). We also assume that the set of data values $\mathcal{D}$ includes process names such as $C, S, Y, \ldots$ and a message *abort*.

The protocol assumes that there is a secure (wrt. integrity) way for C to obtain the public key K_{CA} of the certification authority, and for S to obtain a certificate $\mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)$ signed by the certification authority that contains its name and public key. The adversary may also have access to K_{CA}, $\mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)$ and $\mathcal{S}ign_{K_{CA}^{-1}}(Z :: K_Z)$ for an arbitrary process Z.

The channels between the participants are thus as follows.



Now we define the protocol using our domain-specific language (here and in the following we denote a program with output channel c simply as c for readability).

$$\begin{aligned} c \stackrel{\text{def}}{=} \mathit{if}\ \mathsf{inp}(l) = \varepsilon\ \mathit{then}\ N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C) \\ \mathit{else}\ \mathit{case}\ \mathsf{inp}(s')\ \mathit{of}\ s_1 :: s_2 :: s_3 \\ \mathit{do}\ \mathit{case}\ \mathcal{E}xt_{\mathsf{inp}(a_C)}(s_3)\ \mathit{of}\ S :: x \end{aligned}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{do if } \{\mathcal{D}ec_{K_C^{-1}}(s_2)\}_x = y :: N_C \textit{ then } \{m\}_y \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{else } \text{abort} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{else } \text{abort} \\
\qquad\qquad\qquad\qquad\quad \textit{else } \varepsilon \\
l \stackrel{\text{def}}{=} 0 \\
s \stackrel{\text{def}}{=} \textit{case } \mathsf{inp}(c') \textit{ of } c_1 :: c_2 :: c_3 \\
\qquad\qquad\quad \textit{do } \textit{ case } \mathcal{E}xt_{c_2}(c_3) \textit{of } x :: c_2 \textit{ do } N_S :: \{\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: c_1)\}_{c_2} :: \mathsf{inp}(a_S) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{else } \text{abort} \\
\qquad\qquad\quad \textit{else } \varepsilon \\
r \stackrel{\text{def}}{=} \textit{if } \mathcal{D}ec_{K_{CS}}(\mathsf{inp}(c')) \in \mathbf{Data} \cup \mathbf{Secret} \textit{ then } \mathcal{D}ec_{K_{CS}}(\mathsf{inp}(c')) \textit{ else } \varepsilon \\
a_C \stackrel{\text{def}}{=} K_{CA} \\
a_A \stackrel{\text{def}}{=} K_{CA} :: \mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S) :: \mathcal{S}ign_{K_{CA}^{-1}}(Z :: K_Z) \\
a_S \stackrel{\text{def}}{=} \mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)
\end{array}
$$

For readability we leave out a time-stamp, a session id, the choice of cipher suite and compression method and the use of a temporary key by S since these are not relevant for the weakness. We use syntactic sugar by extending the case list construct to lists of finite length and by using pattern matching, and we also leave out some *case of key do else* constructs to avoid cluttering. Similarly, we use the expression $\mathcal{D}ec_{K_{CS}}(\mathsf{inp}(c')) \in \mathbf{Data} \cup \mathbf{Secret}$ as a shorthand for nested *if then else* statements which iteratively check equality of $\mathcal{D}ec_{K_{CS}}(\mathsf{inp}(c'))$ with all values in the finite set $\mathbf{Data} \cup \mathbf{Secret}$. Here the local channel l of C only ensures that C initiates the handshake protocol only once (by sending out an arbitrary message (0) so that only at the start of the program execution the first condition in the definition of the program on channel c will hold). The exchanged key is symmetric, i. e. we have $K_{CS}^{-1} = K_{CS}$. The values sent on a_A signify that we allow A to eavesdrop on a_C and a_S and to obtain the certificate issued by CA of some third party. The local channel r of the server will contain the decrypted secret (which is assumed to be a value in the set $\mathbf{Data} \cup \mathbf{Secret}$) after it has been communicated successfully.

Note that the above specifications aims to perform a security analysis of a specific situation for the protocol, namely that of a single parallel execution of each of the two protocol participants, and against an attacker that does not have control over either of the participants. Although for simplicity, we focus on this scenario in this paper, we now shortly explain that our approach is general enough to also consider multiple parallel sessions, and attackers that take on the role of one or more protocol participants.

Multiple parallel sessions: To specify the situation where there are multiple parallel sessions, instead of just considering one session key K_{CS} and the associated client C and server S, we consider a set of session keys $calK$, a set of clients $\mathcal{C}$, and a set of servers $\mathcal{S}$ (which may each be infinite), together with two functions $k_C : \mathcal{K} \to \mathcal{C}$ and $k_S : \mathcal{K} \to \mathcal{S}$ which determine which client and server are involved in a given session, which is represented by the session key (which is unique to the session, where the actual key generation is left implicit here). Each client C then has a local channel l_C, input channels s'_C and a_C, and an output channel c_C, as in the above figure. Similarly, each server S has channels as in the figure above, with names indexed by S in the above way. The behavior of each channel C resp. each channel S is as defined above, except that the session key K_{CS} is subtituted with the unique key K such that $k_C(K) = C$ and $k_S(K) = S$ (for all clients C resp. servers S which are in the image of k_C resp. k_S, which means that they are part of a session in the scenario that is modelled).

Insider attackers: To specify the situation where the attacker is actually one of the protocol participants (as in the attack by Lowe against the Needham-Schroeder protocol), we leave out the relevant protocol participant from the model (say, the client C), and instead give the data that is contained in its specification to the adversary, for example through an additional input channel. In the case of the client, that would be the client's public key K_C and private key K_C^{-1}, his nonce N_C, the public key K_{CA} of the certification authority, and the session key K_{CS}.

5.2 The flaw

Theorem 4 *$P \stackrel{\text{def}}{=} C \otimes S \otimes CA$ does not preserve the secrecy of m.*

We prove this theorem by exhibiting a successful attacker, the behaviour of which can itself be represented using our DSL notation as follows:

$$
\begin{aligned}
c' \stackrel{\text{def}}{=}\ & \mathit{case}\ \mathsf{inp}(c)\ \mathit{of}\ c_1 :: c_2 :: c_3 \\
& \quad \mathit{do}\ c_1 :: K_A :: \mathcal{S}ign_{K_A^{-1}}(C :: K_A) \\
& \quad \mathit{else}\ \varepsilon \\
s' \stackrel{\text{def}}{=}\ & \mathit{case}\ \mathsf{inp}(s)\ \mathit{of}\ s_1 :: s_2 :: s_3 \\
& \quad \mathit{do}\ s_1 :: \{\mathcal{D}ec_{K_A^{-1}}(s_2)\}_{K_C} :: s_3 \\
& \quad \mathit{else}\ \varepsilon \\
l_A \stackrel{\text{def}}{=}\ & \mathit{if}\ \mathsf{inp}(l_A) = \varepsilon\ \mathit{then}\ \mathit{case}\ \mathsf{inp}(s)\ \mathit{of}\ s_1 :: s_2 :: s_3
\end{aligned}
$$

$$\textit{do case } \{\mathcal{D}ec_{K_A^{-1}}(s_2)\}_{K_S} \textit{ of } x_1 :: x_2 \textit{ do } x_1 \textit{ else } \mathsf{inp}(l_A)$$
$$\textit{else } \mathsf{inp}(l_A)$$
$$c_0 \stackrel{\text{def}}{=} \textit{case } \mathsf{inp}(l_A) \textit{ of key do if } \mathcal{D}ec_{\mathsf{inp}(l_A)}(\mathsf{inp}(c)) = \bot \textit{ then } \varepsilon \textit{ else } \mathcal{D}ec_{\mathsf{inp}(l_A)}(c) \textit{ else } \varepsilon$$

Proposition 1 $[\![P]\!] \otimes [\![A]\!]_r$ *eventually outputs* m.

The validity of this statement is demonstrated by the man-in-the-middle attack scenario shown in the message flow diagram shown in Fig. 5.

5.3 The fix

Let S' be the process derived from S by substituting $K_{CS} :: c_1$ in the second line of the definition of s by $K_{CS} :: c_1 :: c_2$. Change C to C' by substituting $y :: N_C$ in the fourth line of the definition of c by $y :: N_C :: K_C$.

$$C \xrightarrow{N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C)} S$$

$$C \xleftarrow{N_S :: \{\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: N_C :: \underline{K_C})\}_{K_C} :: \mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)} S$$

$$C \xrightarrow{\{m\}_{K_{CS}}} S$$

Theorem 5 $P' \stackrel{\text{def}}{=} C' \otimes S' \otimes CA$ *preserves the secrecy of* m.

Proof Although we will explain later in this section how protocols written in our DSL can be formally verified in an automated way by making use of automated theorem provers for first-order logic, here we additionally give an informal proof to explain the intuition behind this kind of reasoning.

Given an adversary A with $m \notin S_A \cup K_A$, we need to show that $[\![P']\!] \otimes [\![A]\!]_r$ does not eventually output m. We proceed by execution rounds, making use

$$C \xrightarrow{N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C)} A \xrightarrow{N_C :: K_A :: \mathcal{S}ign_{K_A^{-1}}(C :: K_A)} S$$

$$C \xleftarrow{N_S :: \{\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: N_C)\}_{K_C} :: \mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)} A \xleftarrow{N_S :: \{\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: N_C)\}_{K_A} :: \mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)} S$$

$$C \xrightarrow{\{m\}_{K_{CS}}} A \xrightarrow{\{m\}_{K_{CS}}} S$$

Fig. 5. Attack against TLS Variant

of the fact that the adversary may let its output depend on the output from the honest participants at the same time.

In every round, 0 is output on l, K_{CA} on a_C and a_A, and $\mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)$ on a_S. After the first round, the local storage of C remains unchanged whatever happens, and S and CA do not have a local storage. Thus we only need to consider those actions of A that immediately increase its knowledge (i. e. we need not consider outputs of A that prompt C or S to output ε or *abort* in the following round.

In the first round, $N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C)$ is output on c and ε on s. Since A is not in possession of any message containing S's name and signed by CA at this point, any output on s' will prompt C to output ε or *abort* in the next round, so the output on s' is irrelevant. Similarly, the only relevant output on c' is of the form $c_1 :: K_X :: \mathcal{S}ign_{K_X^{-1}}(Y :: K_X)$, where K_X is a public key with corresponding private key K_X^{-1} and Y a name of a process.

In the second round, the output on c is ε or *abort*, and that on s is ε or *abort* or $N_S :: \{\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: c_1 :: K_X)\}_{K_X} :: a_S$. The only possibility to cause C in the following round to produce a relevant output would be for A now to output a message of the form $N_Z :: \{\mathcal{S}ign_{K_Z^{-1}}(K_{CS} :: c_1 :: K_X)\}_{K_X} :: \mathcal{S}ign_{K_{CA}^{-1}}(S :: K_Z)$. Firstly, the only certificate from CA containing S in possession of A is $\mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)$. Secondly, the only message containing a message signed using K_S in possession of A is $\{\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: c_1 :: K_X)\}_{K_X}$. In case $K_X \neq K_C$ the message signed by S is of the form $\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: c_1 :: K_X)$ for $K_X \neq K_C$, so that C outputs *abort* on receipt of this message anyhow. In case $K_X = K_C$, A cannot decrypt or alter the message $\{\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: c_1 :: K_C)\}_{K_C}$ by assumptions on cryptography and since A does not possess K_C^{-1}. A may forward the message on s'. In this case, C outputs $\{m\}_{K_{CS}}$ in the following round, which A cannot decrypt.

Since the internal state of C, S and CA does not change after the first round, further interaction does not bring any change whatsoever (since it makes no difference if A successively tries different keys K_X or names Y).

Thus P' preserves the secrecy of x.

Note that the nonce N_S is in fact not needed for establishing this claim.

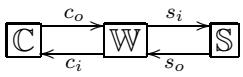
Note also that, just as above we explained how the specification of the TLS variant can be generalized to cover multiple parallel sessions and insider attackers, the above informal proof can also be extended to cover these more general situations. In the case of parallel sessions, one needs to investigate within the above proof, at any point of the protocol execution, whether the adversary can use the additional knowledge from being involved in parallel

session to break the protocol (which is not the case, although we omit the details of this discussion here). In the case of insider attackers, it is easy to see that if the attacker takes on the role of either the client or the server, it is straightforward for him to break the secrecy of the secret in the course of a legal protocol execution, since he is in fact in possession of the session key.

6 Implementing Secure Channels

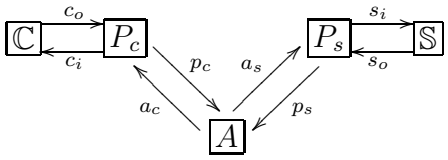
As an example for a stepwise development of a concrete program in our DSL from an abstract specification, we consider the implementation of a secure channel $\mathbb{W}$ from a client $\mathbb{C}$ to a server $\mathbb{S}$ using the handshake protocol considered in Sect. 5.

The initial requirement is that a client $\mathbb{C}$ should be able to send a message msg on $\mathbb{W}$ with intended destination a server $\mathbb{S}$ so that msg is not leaked to A. Before a security risk analysis the situation may simply be pictured as follows:

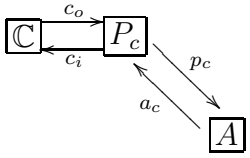


Since there are no unconnected output channels, the composition $\mathbb{C}\otimes\mathbb{W}\otimes\mathbb{S}$ obviously does not leak msg.

Suppose that the risk analysis indicates that the transport layer over which $\mathbb{W}$ is to be implemented is vulnerable against active attacks. This leads to the following model.



We would like to implement the secure channel using the (corrected) variant of the TLS handshake protocol considered in Sect. 5. Thus P_c resp. P_s are implemented by making use of the client resp. server side of the handshake protocol. Here we only consider the client side:



We would like to provide an implementation P_c such that for each $\mathbb{C}$ with $msg \in S_{\mathbb{C}}$, $\mathbb{C} \otimes P_c$ preserves the secrecy of msg (where msg represents the message that should be sent to $\mathbb{S}$). Of course, P_c should also provide functionality: perform the initial handshake and then encrypt data from $\mathbb{C}$ under the negotiated key $K \in \mathbf{Keys}$ and sent it out onto the network. As a first step, we may formulate the possible outputs of P_c as nondeterministic choices (in order to constrain the overall behaviour of P_c). We also allow the possibility for P_c to signal to $\mathbb{C}$ the readiness to receive data to be sent over the network, by sending ok on c_i.

$$
\begin{aligned}
p_c &\stackrel{\text{def}}{=} \mathit{either}\ \mathit{if}\ \mathsf{inp}(c_o) = \varepsilon\ \mathit{then}\ \varepsilon\ \mathit{else}\ \{\mathsf{inp}(c_o)\}_K \\
&\qquad \mathit{or}\ c_K \\
c_i &\stackrel{\text{def}}{=} \mathit{either}\ \varepsilon\ \mathit{or}\ \mathrm{ok}
\end{aligned}
$$

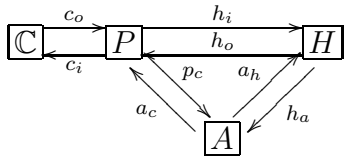
Here c_K denotes the following adaption of the (corrected) program c defined in Sect. 5 (for readability, we allow to use syntactic "macros" here, the resulting program is obtained by "pasting" the following program text in the place of c_K in the definition of p_c). For simplicity, we assume that P_c has already received the public key K_{CA} of the certification authority. We leave out the definition of c_i since at the moment we only consider the case where $\mathbb{C}$ wants to sent data to $\mathbb{S}$.

$$
\begin{aligned}
c_K \stackrel{\text{def}}{=}\ & \mathit{either}\ N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C) \\
& \quad \mathit{or}\ \mathit{case}\ \mathsf{inp}(a_c)\ \mathit{of}\ s_1 :: s_2 :: s_3 \\
& \qquad \mathit{do}\ \mathit{case}\ \mathcal{E}xt_{K_{CA}}(s_3)\ \mathit{of}\ S :: x \\
& \qquad\quad \mathit{do}\ \mathit{if}\ \mathcal{E}xt_x(\mathcal{D}ec_{K_C}(s_2)) = y :: N_C :: K_C\ \mathit{then}\ \{K\}_y \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ \mathrm{abort} \\
& \qquad\quad \mathit{else}\ \mathrm{abort} \\
& \qquad \mathit{else}\ \mathrm{abort}
\end{aligned}
$$

One can show that for any $\mathbb{C}$, the composition $\mathbb{C} \otimes P_c$ preserves the secrecy of the messages sent along c_0.

As a next step, we may split P_c into two components: the client side H of the handshake protocol (as part of the security layer) and program P (in the application layer) that receives data from $\mathbb{C}$, encrypts it using the key received

from H and sends it out on the network:



$$
\begin{aligned}
h_a \stackrel{\text{def}}{=} & \mathit{if}\ \mathsf{inp}(h_i) = \varepsilon\ \mathit{then}\ N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C) \\
& \quad \mathit{else}\ \mathit{case}\ \mathsf{inp}(a_h)\ \mathit{of}\ s_1 :: s_2 :: s_3 \\
& \quad\quad \mathit{do}\ \mathit{case}\ \mathcal{E}xt_{K_{CA}}(s_3)\ \mathit{of}\ S :: x \\
& \quad\quad\quad \mathit{do}\ \mathit{if}\ \{\mathcal{D}ec_{K_C}(s_2)\}_x = y :: N_C :: K_C\ \mathit{then}\ \{m\}_y \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{else}\ \text{abort} \\
& \quad\quad\quad \mathit{else}\ \text{abort} \\
& \quad\quad \mathit{else}\ \text{abort} \\
h_o \stackrel{\text{def}}{=} & \mathit{if}\ \mathsf{inp}(h_i) = \varepsilon\ \mathit{then}\ \varepsilon \\
& \quad \mathit{else}\ \mathit{case}\ \mathsf{inp}(a_h)\ \mathit{of}\ s_1 :: s_2 :: s_3 \\
& \quad\quad \mathit{do}\ \mathit{case}\ \mathcal{E}xt_{K_{CA}}(s_3)\ \mathit{of}\ S :: x \\
& \quad\quad\quad \mathit{do}\ \mathit{if}\ \{\mathcal{D}ec_{K_C}(s_2)\}_x = y :: N_C :: K_C\ \mathit{then}\ \text{finished} \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{else}\ \varepsilon \\
& \quad\quad\quad \mathit{else}\ \varepsilon \\
& \quad\quad \mathit{else}\ \varepsilon \\
h_i \stackrel{\text{def}}{=} & \ 0 \\
p_c \stackrel{\text{def}}{=} & \mathit{if}\ \mathsf{inp}(c_o) = \varepsilon\ \mathit{then}\ \varepsilon\ \mathit{else}\ \{\mathsf{inp}(c_o)\}_K \\
c_i \stackrel{\text{def}}{=} & \mathit{if}\ \mathsf{inp}(h_o) = \text{finished}\ \mathit{then}\ \text{ok}\ \mathit{else}\ \varepsilon
\end{aligned}
$$

We have the conditional interface refinement $P_c \overset{(D,U)}{\rightsquigarrow}_T P \otimes H$ where

- $T \subseteq \mathbf{Stream}_{O_{P_c}} \times \mathbf{Stream}_{I_{P_c}}$ consists of those $(\vec{s}, \vec{t})$ such that for any n, if $(\vec{s}(\tilde{c}_i))\lfloor_i \neq \mathit{finished}$ for all $i \leq n$ then $(\vec{s}(\tilde{c}_o))\lfloor_i = \varepsilon$ for all $i \leq n+1$
- and D and U have channel sets $I_D = \{\tilde{c}_o, \tilde{a}_c\}$, $O_D = \{c_o, a_c, a_h\}$, $I_U = \{c_i, p_c, h_a\}$ and $O_U = \{\tilde{c}_i, \tilde{p}_c\}$ and are specified by

$$
\begin{aligned}
& c_o \stackrel{\text{def}}{=} \mathsf{inp}(\tilde{c}_o), \qquad a_c \stackrel{\text{def}}{=} \mathsf{inp}(\tilde{a}_c), \qquad a_h \stackrel{\text{def}}{=} \mathsf{inp}(\tilde{a}_c), \\
& \tilde{c}_i \stackrel{\text{def}}{=} \mathsf{inp}(c_i), \qquad \tilde{p}_c \stackrel{\text{def}}{=} \mathsf{inp}(h_a)
\end{aligned}
$$

(after renaming the channels of P_c to $\tilde{c}_o, \tilde{c}_i, \tilde{p}_c, \tilde{a}_c$).

Therefore, for any $\mathbb{C}$ with $[\![\mathbb{C}]\!] \subseteq T$, we have an interface refinement $\mathbb{C} \otimes P_c \overset{(D,U)}{\rightsquigarrow} \mathbb{C} \otimes P \otimes H$. Since for any $\mathbb{C}$, the composition $\mathbb{C} \otimes P_c$ preserves the

secrecy of the messages sent along c_0, as noted above, this implies that for any $\mathbb{C}$ with $[\![\mathbb{C}]\!] \subseteq T$, the composition $\mathbb{C} \otimes P \otimes H$ preserves the secrecy of these messages by Theorem 2 (since D and U clearly have inverses).

7 Translating Programs to First-order Logic Formulas

We explain our translation from programs in the DSL defined earlier in this paper to first-order logic (FOL) formulas which can be processed by automated theorem provers (ATPs) for first-order logic such as e-SETHEO or SPASS. The formalization automatically derives an upper bound for the set of knowledge the adversary can gain.

The idea is to use a predicate $\mathsf{knows}(E)$ meaning that the adversary may get to know E during the execution of the protocol. For any data value s supposed to remain secret, one thus has to check whether one can derive $\mathsf{knows}(s)$. The set of predicates defined to hold for a given program in the DSL defined earlier is defined as follows.

For each publicly known expression E, one defines $\mathsf{knows}(E)$ to hold (in particular for the empty message ε, and in the case of the TLS variant from Sect. 5 also for the message abort). The fact that the adversary may enlarge his set of knowledge by constructing new expressions from the ones he knows (including the use of encryption and decryption) is captured by the formula in Fig. 6.

For a given program p in our DSL, we now define the first-order logic formula $\phi(p)$ used to represent p for the purposes of the security analysis. The definition is given in Figure 7 and is analogous to the definition of the formal semantics in Figure 3. For the usages of $E \in \mathbf{Exp}$ in Figure 7, the assumption is that there are n occurrences of input expressions in E, and $E(i_1, \ldots, i_n)$ is the expression derived from E by substituting these occurrences by the variables $i_1, \ldots, i_n$. Similarly, for $E' \in \mathbf{Exp}$, the assumption is that there are m occurrences of input expressions in E', and $E'(j_1, \ldots, j_m)$ is the expression derived from E' by substituting these occurrences by the variables $j_1, \ldots, j_m$. Also, we assume that the predicate $\mathsf{key}(\mathsf{K})$ is true iff K is a key.

$$
\begin{aligned}
\forall E_1, E_2.\, & (\mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2) \Rightarrow \mathsf{knows}(E_1 :: E_2) \wedge \mathsf{knows}(\{E_1\}_{E_2}) \wedge\ \mathsf{knows}(\mathcal{S}ign_{E_2}(E_1))) \\
& \wedge (\mathsf{knows}(E_1 :: E_2) \Rightarrow \mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)) \\
& \wedge (\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2^{-1}) \Rightarrow \mathsf{knows}(E_1)) \\
& \wedge (\mathsf{knows}(\mathcal{S}ign_{E_2^{-1}}(E_1)) \wedge \mathsf{knows}(E_2) \Rightarrow \mathsf{knows}(E_1))
\end{aligned}
$$

Fig. 6. Structural formulas

$$\phi(E) = \forall i_1, \ldots, i_n.(\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \Rightarrow \mathsf{knows}(E(i_1, \ldots, i_n)))$$
$$\phi(\mathit{either}\ p\ \mathit{or}\ p') = \phi(p) \wedge \phi(p')$$
$$\phi(\mathit{if}\ E = E'\ \mathit{then}\ p\ \mathit{else}\ p') =$$
$$\forall i_1, \ldots, i_n.(\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n)) \Rightarrow$$
$$(E(i_1, \ldots, i_n) = E'(i_1, \ldots, i_n) \Rightarrow \phi(p))$$
$$\wedge (E(i_1, \ldots, i_n) \neq E'(i_1, \ldots, i_n) \Rightarrow \phi(p')))$$
$$\phi(\mathit{case}\ E\ \mathit{of}\ \mathit{key}\ \mathit{do}\ p\ \mathit{else}\ p') =$$
$$\forall i_1, \ldots, i_n.(\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \Rightarrow$$
$$(\mathsf{key}(E(i_1, \ldots, i_n)) \Rightarrow \phi(p)) \wedge (\neg \mathsf{key}(E(i_1, \ldots, i_n)) \Rightarrow \phi(p')))$$
$$\phi(\mathit{case}\ E\ \mathit{of}\ x :: y\ \mathit{do}\ p\ \mathit{else}\ p') =$$
$$\forall i_1, \ldots, i_n, h, t.(\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \wedge E(i_1, \ldots, i_n) = h :: t \Rightarrow \phi(p[h/x, t/y]))$$
$$\wedge \forall i_1, \ldots, i_n.(\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \wedge \neg \exists h, t.E(i_1, \ldots, i_n) = h :: t \Rightarrow \phi(p'))$$

Fig. 7. Definition of $\phi(p)$.

The formula formalizes the fact that, if the adversary knows expressions $exp_1, \ldots, exp_n$ validating the condition $cond(exp_1, \ldots, exp_n)$, then he can send them to one of the protocol participants to receive the message $exp(exp_1, \ldots, exp_n)$ in exchange, and then the protocol continues. With this formalization, a data value s is said to be kept secret if it is not possible to derive $\mathsf{knows}(s)$ from the formulas defined by a protocol. Note that here we abstract from the message sender and receiver identities (because we do not want to make the assumption that sender and receiver identities are securely bound to the messages, which is one reason why one needs cryptographic protocols to start with) and the message order (because the adversary may be able to change the order of messages while in transit over the communication channel).

Suppose now that we are given a process $P = (I, O, L, (p_c)_{c \in O})$ where I, O, L are the sets of input, ouput, and local channels. For a program p associated with a non-local output channel, this gives a predicate $\mathsf{PRED}(p)$ where expressions of the form $\mathsf{knows}(i_k)$ (where i_k is an input received over a local input channel) are substituted by the expression $\mathsf{sent}(i_k)$. If p is associated with a local output channel, the expression $\mathsf{knows}(E(i_1, \ldots, i_n))$ in the first line of Fig. 7 is similarly substituted by $\mathsf{sent}(E(i_1, \ldots, i_n))$. Additionally, we assume $\mathsf{sent}(\varepsilon)$ as an axiom (where we recall that ε represents the empty message). These modifications capture the fact that the adversary cannot read from or write to local channels. Note that the sent predicate gives an upper bound on the data that may be sent over the local channels. This means that the adversary knowledge set is approximated from above. In particular, one will find all possible attacks, but one may also encounter "false positives", although this has not happened yet with any real examples. The advantage is that this approach is rather efficient (see Sect. 7.2 for some performance data).

The axioms in the overall first-order logic formula for a given protocol are then the conjunction of the formulas representing the publicly known expressions, the formula in Fig. 6, and the conjunction of the formulas $\mathsf{PRED}(p)$ for each

program p implementing a part of the protocol (with the modifications explained in the previous paragraph). The conjecture, for which the automated theorem prover will check whether it is derivable from the axioms, depends on the security requirements contained in the class diagram. For the requirement that the data value s is to be kept secret, the conjecture is $\mathsf{knows}(s)$. An example is given in the next section.

Note that, as before, the generalizations to cover multiple parallel sessions and insider attackers also extend to the translation to FOL. For the case of insider attackers, this is immediate since it simply results in a different protocol model, which can be translated in the same way. For multiple parallel sessions, one has data values that are parameterized over the protocol roles (such as client and server) which are modelled by free variables, and the axioms then need to be closed using forall-quantifiers for each such free variable.

7.1 Translating the TLS Variant

To explain the translation defined above, we apply it to the DSL program constructed for the TLS variant in Sect. 5. We explain this translation in a stepwise manner for each of the DSL sub-programs defined for the TLS variant.

For the output channels a_C, a_A, a_S of the certification authority, as well as the trivial local channel l, the translation is particularly easy. The first line in the definition in Fig. 7 applies and since the relevant expressions do not contain input channels, the resulting subformulas reduce to rather simple logical predicates (where $\cong$ denotes equivalence of the logical formulae):

$$\begin{aligned}
\phi(a_C) &\cong \mathsf{knows}(K_{CA}) \\
\phi(a_A) &\cong \mathsf{knows}(K_{CA} :: \mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S) :: \mathcal{S}ign_{K_{CA}^{-1}}(Z :: K_Z)) \\
\phi(a_S) &\cong \mathsf{knows}(\mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)) \\
\phi(a_S) &\cong \mathsf{sent}(0)
\end{aligned}$$

For the program defining the behavior on channel c, the first line of the definition reduces to a similarly simple formula (which will be a conjunct within the overall formula $\phi(c)$): According to the definition for the then-branch of an if-statement given in Fig. 7, we get a conjunct which simply reduces to $\mathsf{knows}(N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C))$ (since $\mathsf{sent}(\varepsilon)$ holds as a given axiom and $\phi(N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C)) \cong \mathsf{knows}(N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C))$). For the else-case in this if-statement, we get a second conjunct in the overall formula $\phi(c)$ which reduces to the following formula (and additionally formulae which imply $\mathsf{knows}(\varepsilon)$ and $\mathsf{knows}(\mathsf{abort})$, which can be ignored since that

already is part of the general axioms and to simplify the presentation):

$$\forall s_1, s_2, s_3, a_1, x, y.\ (\mathsf{knows}(s_1) \wedge \mathsf{knows}(s_2) \wedge \mathsf{knows}(s_3) \wedge \mathsf{knows}(a_1) \wedge \{s_3\}_{a_1} = S :: x \wedge \{\mathcal{D}ec_{K_C^{-1}}(s_2)\}_x = y :: N_C \Rightarrow \mathsf{knows}(\{m\}_y))$$

Finally, $\phi(s)$ reduces to the following formula (again abstracting away from $\mathsf{knows}(\varepsilon)$ and $\mathsf{knows}(\mathsf{abort})$):

$$\forall c_1, c_2, c_3, a_1, x.\ (\mathsf{knows}(c_1) \wedge \mathsf{knows}(c_2) \wedge \mathsf{knows}(c_3) \wedge \mathsf{knows}(a_1) \wedge \{c_3\}_{c_2} = x :: c_2 \Rightarrow \mathsf{knows}(N_S :: \{\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: c_1)\}_{c_2} :: a_1))$$

Altogether, the DSL program that implements the TLS variant given in Sect. 5 thus translates to a logical formula that reduces to the simplified formula given in Fig. 8. This formula is given as an axiom to the ATP in conjunction with the general axioms defined in the last section, as we will explain in the following subsection.

$$\begin{aligned}
&\mathsf{knows}(N_C :: K_C :: \mathcal{S}ign_{K_C^{-1}}(C :: K_C)) \\
\wedge\, &\forall s_1, s_2, s_3, a_1, x, y.(\mathsf{knows}(s_1) \wedge \mathsf{knows}(s_2) \wedge \mathsf{knows}(s_3) \wedge \mathsf{knows}(a_1) \wedge \\
&\quad \{s_3\}_{a_1} = S :: x \wedge \{\mathcal{D}ec_{K_C^{-1}}(s_2)\}_x = y :: N_C \Rightarrow \mathsf{knows}(\{m\}_y)) \\
\wedge\, &\mathsf{sent}(0) \\
\wedge\, &\forall c_1, c_2, c_3, a_1, x.(\mathsf{knows}(c_1) \wedge \mathsf{knows}(c_2) \wedge \mathsf{knows}(c_3) \wedge \mathsf{knows}(a_1) \wedge \\
&\quad \{c_3\}_{c_2} = x :: c_2 \Rightarrow \mathsf{knows}(N_S :: \{\mathcal{S}ign_{K_S^{-1}}(K_{CS} :: c_1)\}_{c_2} :: a_1)) \\
\wedge\, &\mathsf{knows}(K_{CA}) \\
\wedge\, &\mathsf{knows}(K_{CA} :: \mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S) :: \mathcal{S}ign_{K_{CA}^{-1}}(Z :: K_Z)) \\
\wedge\, &\mathsf{knows}(\mathcal{S}ign_{K_{CA}^{-1}}(S :: K_S)) \\
\wedge\, &\mathsf{knows}(\mathsf{abort}) \\
\wedge\, &\mathsf{knows}(\varepsilon)
\end{aligned}$$

Fig. 8. Protocol part of the translation of the TLS variant to FOL

7.2 Protocol Analysis with Automated Theorem Provers for FOL

In this subsection, we explain how the automated analysis of programs in the DSL are performed using the translation to FOL defined at the beginning of this section. For that, the resulting FOL formula is converted to the TPTP notation, which is the de-facto input notation for first-order logic automated

theorem provers [38], supported, using existing converters, by a variety of provers including e-SETHEO, Otter, SPASS, Vampire, and Waldmeister.

The prover SETHEO is an efficient automated theorem prover for first oder logic in clausal normal form, extended to the prover e-SETHEO for reasoning about equality properties [31,37]. We use e-SETHEO for verifying security protocols as a "black box": A TPTP input file is presented to the ATP and an output from the ATP is observed. No internal properties of or information from e-SETHEO is used. This allows one to use e-SETHEO interchangingly with any other ATP accepting TPTP as an input format (such as SPASS, Vampire and Waldmeister) when it may seem fit.

The results of the theorem prover have to be interpreted as follows: If the conjecture stating for example that the adversary may get to know the secret can be derived from the axioms which formalize the adversary model and the protocol specification, this means that there may be an attack against the protocol.

We then use an attack generation script programmed in Prolog to construct the attack. The script essentially implements the above FOL formalization in Prolog. The advantage of a Prolog implementation is that one can directly query the variable contents (i.e., the messages that are exchanged) when an attack is found. The disadvantage of the Prolog implementation is that it is significantly less efficient and complete in the case of secure protocols (which is why we use the FOL prover first, and the Prolog generator only if the prover found that there may be an attack). Note that the Prolog script only generates the message sequence from the attack scenario (as shown in Fig. 5; see [20] for the actual output of the tool). The user then needs to manually improve on the original DSL specification on the basis of this information.

If the conjecture cannot be derived from the axioms, this constitutes a proof that the protocol is secure with respect to the security requirement formalized as the negation of the conjecture, because logical derivation is sound and complete with respect to semantic validity for first-order logic. Note that since first-order logic in general is undecidable, it can happen that the ATP is not able to decide whether a given conjecture can be derived from a given set of axioms. However, experience has shown that for a reasonable set of protocols and security requirements, our approach is in fact practical.

In our example, e-SETHEO returns as an output that the conjection $\mathsf{knows}(\mathsf{secret})$ can be derived from the defined rules (within three seconds [2]). For this example the attack tracking tool needs around 20 seconds to produce the attack which was visualized earlier in Fig. 5. See [20] for the actual outputs of the tools.

[2] On a SunFire 3800 (4 processors, 6 GByte RAM, Solaris 9).

As explained earlier, we can fix this problem by substituting $K :: N_C$ in the second message (server to client) by $K :: N_C :: K_C$ and by including a check regarding this new message part at the client. Now the new version can be verified by the automated theorem prover approach. When e-SETHEO runs on the fixed version of the protocol it now gives back the result that the conjecture knows(secret) cannot be derived from the axioms formalizing the protocol, within 5 seconds. Again, see [20] for the tool output, which shows that, within the e-SETHEO suite, the prover "eprover" was able to establish that there is no such derivation by exhaustively trying all possibilities. Note that this result means that there actually exists no such derivation, not just that the theorem prover is not able to find it. This means in particular that the attacker cannot gain the secret knowledge anymore.

Note that this a non-trivial result: The translation from the DSL into FOL defined above accommodates the fact that, for example, the attacker can construct messages of arbitrary size from his knowledge (e.g. lists of arbitrary length, or message of arbitrary encryption depth). We leave it up to the ATPs we use (such as e-SETHEO and SPASS) to deal with this verification challenge. This allows us not only to use different ATPs in an interchangeable way, which increases the overall robustness of the approach. It also allows us to exploit the significant amount of work that has been done regarding automated theorem proving, which is already encapsulated in these tools. In particular, using our translation, the challenge of dealing with the fact that the adversary may construct messages of arbitrary size, is embedded into the challenge of dealing with the fact that there exist derivations of arbitrary size, when trying to determine whether a conjecture can be derived from a set of axioms. The latter is of course the main challenge that has to be addressed by automated theorem provers which aim for a maximum feasible level of completeness. Although we cannot hope for a complete, automated solution for problem instances of arbitrary difficulty (because of the undecidability of this problem), encouraging improvements have been made in this regards over the last years, which we aim to exploit using our approach. Note also that if necessary, the approach presented here (which is particularly suitable to derive attacks) can also be fruitfully combined with complementary approaches such as [36,10] which are targeted on deriving security proofs.

Note that the security result above of course in itself is bound to the particular execution model and the formalizations of the security requirements used here. Note also that the approach does not aim to be complete in the sense that the security analysis may falsely claim that there may be an attack against the specified system, because of the optimizing abstractions used. This, however, has not so far surfaced as a limitation in practical applications. If it would, one would be able to detect that the attack is unrealistic when generating the attack scenario using the Prolog attack generator mentioned above.

$\phi(E) = \mathsf{output}(E)$
$\phi(either\ p\ or\ p') = \phi(p) \wedge \phi(p')$
$\phi(if\ E = E'\ then\ p\ else\ p') = ((E = E' \Rightarrow \phi(p)) \wedge (E \neq E' \Rightarrow \phi(p')))$
$\phi(case\ E\ of\ key\ do\ p\ else\ p') = (\mathsf{key}(E) \Rightarrow \phi(p)) \wedge (\neg\mathsf{key}(E) \Rightarrow \phi(p')))$
$\phi(case\ E\ of\ x :: y\ do\ p\ else\ p') =$
$\quad \forall h, t.(E = h :: t \Rightarrow \phi(p[h/x, t/y])) \wedge (\neg\exists h, t.E = h :: t \Rightarrow \phi(p'))$

Fig. 9. Definition of $\phi(p)$.

A prototypical implementation of the tool-support, which performs a translation from a state machine representation of the protocol to the FOL formulas in the TPTP format, can be downloaded as open source from [20].

8 Test Case Generation

In previous sections, we explained how one can create executable but formally founded models of cryptographic protocols using the DSL presented earlier, which can be verified against security requirements such as secrecy using automated theorem provers. In this section, we explain how to use the DSL programs to provide security assurance for implementations of cryptographic protocols in other programming languages, by generating test-sequences from the DSL programs that can be used to verify these implementations.

The approach works by constructing a program in the DSL which can be formally analyzed as explained above, and then using this program to generate the test-cases for the legacy implementation. The goal is to show that the legacy implementation implements a behavior which is equivalent to that of the DSL program in a suitable sense, which in turn was verified to be secure. We explain the application at the case of implementations in the Java language. Below we first explain how abstract message sequences can be constructed using using Prolog generated from the DSL specifications. We then explain how the abstract message sequences can be concretized in order to be fed into the crypto protocol implementations.

To generate the abstract message sequences, we make use of Prolog scripts that are generated from the DSL specifications, similar to the attack generation scripts mentioned in the previous subsection. These are essentially a Prolog based implementation of the translation from the DSL specifications to FOL which was defined earlier, except that we do not need the formulas which define how the adversary can construct new knowledge, and that we can define directly how the outputs are computed from the inputs (which are generated independently) using the Prolog script, without needing the knows predicate. See Fig. 9 for the core part of this adapted translation. The input

expressions $\mathsf{inp}(c)$ (for a channel c) contained in an expression E are evaluated by substituting them with the input value at the relevant channel at a given point in time during the execution. In Fig. 10 we give a fragment of the Prolog script that is generated for the first two messages of the TLS variant from previous sections using the translation explained above (see [20] for the complete Prolog script), and in Fig. 11 one of the message sequences generated by that Prolog script.

We now explain how the abstract message sequences can be concretized in order to be fed into the crypto protocol implementations. The challenge here in the case of testing cryptographic protocols is that these make significant use of (pseudo-)random algorithms, for example to generate one-time values (nonces) or to make ciphertexts unpredictable and harder to crypto-analyze. These algorithms are designed to withstand brute-force attacks. Therefore one might not expect to be able to test a system making use of them in a sufficiently exhaustive way using black-box testing. Also, one faces difficulties when trying to compare randomly generated values that were created independently by the implementation under test and the test oracle, since these will in general be different.

Instead we make use of white-box testing. This requires to have the source code available, but fortunately that is the case for important crypto-protocol implementations in Java, such as the Java Secure Sockets Extension (JSSE) recently made open-source by Sun.

To instrument the code for the testing, we replace all occurrences of calls of methods that correspond to the cryptographic functions in our DSL (such as random number generation, key generation, encryption, decryption, and signature creation and verification) by methods that not only call these functions, but in addition also build up a table that defines a correspondence

```
%Message number 1, Sender:Client => Receiver: Server
msg(Term, Cond, Step,'Client') :-
Step = 1,
Cond = [],
Term = [[n,k_c],sign([c,k_c],inv(k_c))].
%Message number 2, Sender:Server => Receiver: Client
msg(Term, Cond, Step,'Server') :-
Step = 2,
Init_3=sign([c,Init_2],inv(Init_2)),
Cond = [[Init_1,Init_2],Init_3],
input(Cond),
keygen(Init_2,Key),
Term = [enc(sign([Key,Init_1],inv(k_s)),Init_2),sign([s,k_s],inv(k_ca))].
```

Fig. 10. Generated Prolog script for TLS variant (fragment).

```
Client -> A: [[n, k_c], sign([c, k_c], inv(k_c))]
A -> Server: [[n, k_a], sign([c, k_a], inv(k_a))]
Server -> A: [enc(sign([k1, n], inv(k_s)), k_a), sign([s, k_s], inv(k_ca))]
A -> Client: [enc(sign([k1, n], inv(k_s)), k_c), sign([s, k_s], inv(k_ca))]
Client -> A: enc(secret, k1)
```

Fig. 11. One message sequence generated by the Prolog script in Fig. 10.

between the concrete cryptographic data that is constructed and the corresponding symbolic expressions (as defined in Fig. 1). For example, we replace each occurrence of a call to a method which generates random values (such as nextBytes from java.security.SecureRandom) by a method which calls the same method, but in addition stores the return value in a list.

Similarly, we need to treat the input/output behavior of the crypto-protocol implementation necessary for the protocol communication. In Java-based implementation such as JSSE or the open-source SSL implementation Jessie, the protocol communication is implemented using message buffers to which messages to be sent are written (using the method java.io.OutputStream.write), and from which messages that are received are read. Accordingly, to instrument the code for testing, calls to java.io.OutputStream.write are replaced by calls of a new method which not only includes the original call to java.io.OutputStream.write but also records its arguments in a separate buffer. During the testing phase, the messages in that buffer are then compared to the messages that should be sent according to the given specification (as produced by the DSL program), as explained in more detail in the paragraphs below. If this comparison fails, the given test-case fails. Analogously, the calls of java.io.OutputStream.read (which reads the messages that are received from the buffer) are instrumented such that in addition to reading the messages from the buffer as usual, the values are linked to the representation on the symbolic level as input channels, as defined in Fig. 1. Note that this does not impose any constraint on the well-formedness of the input messages that arrive, since the procotol implementation should also be tested to be secure when receiving a message that is not well-formed as specified by the protocol design.

To perform the comparison of the messages that are sent with the values that should be sent according to the specification, we need to generate concrete values out of the abstract values contained in the specification. This concretization can be achieved using the mappings between abstract and concrete data that are constructed using the code instrumentation explained above. We give some more details on how this works. First, we fix a set of transaction variables TransV and define a concretization of abstract messages by mapping each abstract message type M from Fig. 1 to a sequence

$\mathsf{concrete}(M) : d_1^M, d_2^M, \ldots, d_{n_M}^M$ of concrete data elements d_i^M. d_i^M can be

- a concrete data value (corresponding to a constant sequence of bytes)
- a transaction variable (used to represent transaction data such as timestamps stored by the monitor)
- an abstract message as defined in Fig. 1.

In the last case, the expression is evaluated and the result is again concretized. The transaction variables $v \in \mathsf{TransV}$ are associated with a set $\mathsf{values}(v)$ of possible concrete values. In addition, each data element has to be assigned a field length, which we omit here for simplification. Keys k are mapped directly to a transaction variable $\mathsf{concrete}(k) : d_1^k \in \mathsf{TransV}$. The actual concretization, i.e. the values for the d_i^M and $\mathsf{values}(v)$ must be provided by the developer.

An algorithm for the security monitor is given in Fig. 12. It uses the algorithms **gen_sequence** and **verify_sequence** to generate concrete data from abstract input messages to a component, resp. to compare abstract output messages to the concrete data received. In **verify_sequence**, *first*(s) denotes the prefix of s corresponding to d_i^M, *removefirst*(s) denotes the remaining part of s. The message M is referenced by "this".

The idea of the algorithm is as follows. Constants in $\mathsf{concrete}(M)$ are passed directly to the implementation or compared to the received data. If a transaction variable v appears in $\mathsf{concrete}(M)$ for an input message, either a new concrete value is chosen for v and sent, or an already chosen value from the store *store* is sent (where the variable *store* contains a list of concrete data values and is initially empty). When data is received corresponding to v, it is either compared to the value already chosen, or the received value is added to the store. Encrypted messages, hashes and message authentication codes can be computed on sending using the data available. On reception, it is possible that the concrete byte sequence for a key or part of the data to be hashed is not yet available to the security monitor, so for verification, in **gen_sequence** arbitrary concrete values that may not correspond to the actual values would be chosen and added to the store. In this case, instead a condition is added to *conditions* (which is a variable that stores a list of logical conditions and is initially empty). The conditions are verified by the security monitor as soon as they can be evaluated. The processing of messages by the security monitor is repeated for each step of the data sequence (sending or receiving a message M_{abstr} to/from port p of the component that is monitored). Note that a fresh store must be created for each new protocol session.

As an example, for the Java implementation of the SSL protocol in the Jessie project, the message parts recorded for the ClientHello method are shown in Fig. 13.

algorithm do_check
for each step (p,M_{abstr}) in concrete data sequence
  if input message: send $gen\_sequence(M_{abstr})$ to p
  else wait for output s on p (fail on timeout)
    if verify_sequence(M_{abstr},s) = **false**: fail
  if $\exists\ c \in conditions$: c evaluatable and $\mathsf{evaluate}(c)$ = **false**: fail
algorithm gen_sequence(M_{abstr})
{compute concrete data from abstract message M_{abstr}}
$s \leftarrow \epsilon$
if $M_{abstr} = \mathsf{Encr}(k,x)$ or $M_{abstr} = \mathsf{Mac}(k,x)$ or $M_{abstr} = \mathsf{Hash}(x)$:
  $concr\_msg \leftarrow \mathsf{gen\_sequence}(x)$
  if $M_{abstr} = \mathsf{Encr}(k,x)$ or $M_{abstr} = \mathsf{Mac}(k,x)$:
    $concr\_key \leftarrow \mathsf{gen\_sequence}(k)$
  apply encryption, mac generation or hashing to $concr\_msg$;
    append to s
else determine message type M of M_{abstr};
  $(d_1^M, \ldots, d_{n_M}^M) \leftarrow \mathsf{concrete}(M)$
  for $i \in \{1 \ldots n_M\}$:
    if $d_i^M \in \mathbb{Z}$: append d_i^M to s
    else if $d_i^M \in \mathsf{TransV}$:
      if $\exists y : (d_i^M, y) \in store$: append y to s
      else choose $y \in \mathsf{values}(d_i^M)$;
        append y to s, $store \leftarrow store \cup (d_i^M, y)$
    else if $d_i^M \in$ **Exp**:
      append $\mathsf{gen\_sequence}(\mathsf{evaluate}(d_i^M[\mathbf{this} \leftarrow M_{abstr}]))$ to s
return s
algorithm verify_sequence(M_{abstr}, s)
{verify concrete data s w.r.t. abstract message M_{abstr}}
if $M_{abstr} = \mathsf{Encr}(k,x)$:
  if $\exists concr\_key : (k, concr\_key) \in store$:
    $concr\_msg \leftarrow \mathsf{decrypt}(concr\_key, s)$; $\mathsf{verify\_sequence}(x, concr\_msg)$
  else $conditions \leftarrow conditions \cup \{\mathsf{verify\_sequence}(M_{abstr}, s)\}$
else if $M_{abstr} = \mathsf{Mac}(k,x)$ or $M_{abstr} = \mathsf{Hash}(x)$:
  if $gen\_sequence(x)$ computable without changing store
    and $\exists concr\_key : (k, concr\_key) \in store$ (for $M_{abstr} = \mathsf{Mac}(k,x)$):
    $concr\_msg \leftarrow \mathsf{gen\_sequence}(x)$;
      compare hash/mac of $concr\_msg$ to s
  else $conditions \leftarrow conditions \cup \{\mathsf{verify\_sequence}(M_{abstr}, s)\}$
else determine message type M of M_{abstr};
  $(d_1^M, \ldots, d_{n_M}^M) \leftarrow \mathsf{concrete}(M)$
  for $i \in \{1 \ldots n_M\}$:
    if $d_i^M \in \mathbb{Z}$: compare($first(s)$,d_i^M); $s \leftarrow removefirst(s)$
    else if $d_i^M \in \mathsf{TransV}$:
      if $\exists y : (d_i^M, y) \in store$: $\mathsf{compare}(first(s), y)$; $s \leftarrow removefirst(s)$
      else $store \leftarrow store \cup (d_i^M, first(s))$; $s \leftarrow removefirst(s)$
    else if $d_i^M \in$ **Exp**:
      $\mathsf{verify\_sequence}(\mathsf{evaluate}(d_i^M[\mathbf{this} \leftarrow M_{abstr}]), first(s))$
      $s \leftarrow removefirst(s)$
return **false** if any comparison failed

Fig. 12. Security Monitor Algorithm

9 Related Work

Compared to research done using formal methods, less work has been done more generally using software engineering techniques for computer security. For an overview of the topic see [11].

Other work in secure software engineering which is based on stream-processing functions includes [28], which introduces a method for the formal development of secure systems based on FOCUS. It utilizes threat scenarios which are the

in Model	Send: ClientHello	by Outputstream.write in
	type.getValue()	Handshake.write
	(bout.size() >>> 16 & 0xFF)	Handshake.write
	(bout.size() >>> 8 & 0xFF)	Handshake.write
	(bout.size() & 0xFF)	Handshake.write
Pver	major	ProtocolVersion.write
	minor	ProtocolVersion.write
	((gmtUnixTime >>> 24) & 0xFF)	Random.write
	((gmtUnixTime >>> 16) & 0xFF)	Random.write
	((gmtUnixTime >>> 8) & 0xFF)	Random.write
	(gmtUnixTime & 0xFF)	Random.write
R_C	randomBytes	ClientHello.write
	sessionId.length	ClientHello.write
Sid	sessionId	ClientHello.write
	((suites.size() << 1) >>> 8 & 0xFF)	ClientHello.write
	((suites.size() << 1) & 0xFF)	ClientHello.write
Ciph[]	id[]	CipherSuite.write
	comp.size()	ClientHello.write
Comp[]	comp[2]	ClientHello.write

Fig. 13. Message stored from ClientHello

result of threat identification and risk analysis and model those attacks that are of importance to the system's security. Other examples include the work reported in [40] (and the references there), which develops an approach for secure software engineering using the CASE tool AutoFocus which includes work towards mutation-based testing [23].

In a more general context of model-based development of secure software, [12] defines role-based access control rights from object-oriented use cases. [14] demonstrates how to use UML for aspect-oriented development of security-critical systems. Design-level aspects are used to encapsulate security concerns that can be woven into the models. [17] uses UML for the risk assessment of an e-commerce system within the CORAS framework for model-based risk assessment. [13] uses UML for the design of secure databases. It proposes an extension of the use case and class models of UML using their standard extension mechanisms designing secure databases. [26] demonstrate how to deal with access control policies in UML. The specification of access control policies is integrated into UML. [27,4] show how UML can be used to specify access control in an application and how one can then generate access control mechanisms from the specifications. The approach is based on role-based access control and gives additional support for specifying authorization constraints. [6] presents an approach for the predicative specification of user rights in the context of an object oriented use case driven development process. As syntactic and semantic framework it uses first-order logic with a built-in notion of objects and classes provided with an algebraic semantics. [16] presents an extensible domain architecture for the collaborative development and management of security-critical, inter-organizational workflows where models in-

tegrate security requirements at the abstract level and are rendered in a visual language based on UML 2.0.

There is too much work on verifying cryptographic protocols to give a complete overview. Overviews of applications of formal methods to security protocols can be found for example in [1,30,35,18], some examples in [25,29,2,32]. Another approaches to using first-order logic ATPs for cryptoprotocol analysis include the following: [36] formalizes the well-known BAN logic in first-order logic and uses the ATP SETHEO to proof statements in the BAN logic. It is different from our approach which is based on the knowledge of the adversary, instead of the beliefs of the protocol participants. [39] uses the ATP SPASS to verify crypto protocols. [10] uses first-order invariants to verify cryptographic protocols against safety properties. The approach is supported by the ATP TAPS. Compared to our approach, the method does not generate counter-examples (that is, attacks) in case a protocol is found to be insecure.

So far there does not seem to exist a domain-specific language for crypographic protocols in the literature which has been used to provide assurance for crypto-protocol implementation by generating test-cases.

However, more recently there has been work towards verifying implementations of cryptographic protocols, such as [24,15,5].

10 Conclusion

We present a domain-specific language for crypto protocol implementations which can be used to construct a compact and precise yet executable representation of a cryptographic protocol. This high-level program can be translated to first order logic with equality and then verified against the security goals using automated theorem provers for first order logic. If the analysis reveals that there could be an attack against the protocol, the theorem prover is again called using an attack generator written in Prolog to produce an attack scenario. The protocol can then be corrected by the designer, and the process repeated. One can then use the DSL program to provide assurance for legacy implementations of crypto protocols by generating test-cases. We explained our approach at the hand of a variant of the Internet protocol TLS. The tools presented here can be downloaded from [20] as open-source.

Acknowledgements Help from Gernot Stenz with using e-SETHEO ist greatly appreciated, as well as feedback from Maria Spichkova about a draft of this paper. Constructive comments from the anonymous reviewers which helped clarifying several points in the paper are very gratefully acknowledged.

References

[1] M. Abadi. Security protocols and their properties. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 39–60. IOS Press, Amsterdam, 2000. 20th International Summer School, Marktoberdorf, Germany.

[2] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, Jan. 1999.

[3] V. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost? In *Conference on Computer Communications (IEEE Infocom)*, pages 717–725. IEEE Computer Society, Mar. 1999.

[4] D. Basin, J. Doser, and T. Lodderstedt. Model driven security for process-oriented systems. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, pages 100–109. ACM, 2003.

[5] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW*, pages 139–152. IEEE Computer Society, 2006.

[6] R. Breu and G. Popp. Actor-centric modeling of user rights. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering (FASE 2004)*, volume 2984 of *Lecture Notes in Computer Science*, pages 165–179. Springer-Verlag, 2004.

[7] M. Broy. A logical basis for component-based systems engineering. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. IOS Press, Amsterdam, 1999.

[8] M. Broy and G. Stefanescu. The algebra of stream processing functions. *Theor. Comput. Sci.*, 258(1-2):99–129, 2001.

[9] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.

[10] E. Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

[11] P. Devanbu and S. Stubblebine. Software engineering for security: A roadmap. In *22nd International Conference on on Software Engineering (ICSE 2000): Future of Software Engineering Track*, pages 227–239. ACM, 2000.

[12] E. Fernandez and J. Hawkins. Determining role rights from use cases. In *Workshop on Role-Based Access Control*, pages 121–125. ACM, 1997.

[13] E. Fernández-Medina, A. Martínez, C. Medina, and M. Piattini. UML for the design of secure databases: Integrating security levels, user roles, and constraints in the database design process. In Jürjens et al. [21], pages 93–106.

[14] G. Georg, R. France, and I. Ray. An aspect-based approach to modeling security concerns. In Jürjens et al. [21], pages 107–120.

[15] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real c code. In *VMCAI'05*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[16] M. Hafner, M. Alam, and R. Breu. Towards a MOF/QVT-based domain architecture for model driven security. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2006.

[17] S. Houmb, F. den Braber, M. Lund, and K. Stølen. Towards a UML profile for model-based risk assessment. In Jürjens et al. [21], pages 79–92.

[18] J. Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2004.

[19] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *27th International Conference on Software Engineering (ICSE 2005)*. IEEE Computer Society, 2005.

[20] J. Jürjens. A domain-specific language for cryptographic protocols based on streams: Experimental data. http://mcs.open.ac.uk/jj2924/publications/cryptodsl, Apr. 2008.

[21] J. Jürjens, V. Cengarle, E. Fernandez, B. Rumpe, and R. Sandner, editors. *Critical Systems Development with UML (CSDUML 2002)*, TU München Technical Report TUM-I0208, 2002. UML 2002 satellite workshop proceedings.

[22] J. Jürjens and P. Shabalin. Tools for secure systems development with UML. *Intern. Journal on Software Tools for Technology Transfer*, 9(5–6):527–544, Oct. 2007. Invited submission to the special issue for FASE 2004/05.

[23] 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. IEEE Computer Society, 2001.

[24] J. Jürjens and M. Yampolskiy. Code security analysis with assertions. In D. Redmiles, T. Ellman, and A. Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 392–395. ACM, 2005.

[25] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, Spring 1994.

[26] M. Koch and F. Parisi-Presicce. Access control policy specification in UML. In Jürjens et al. [21], pages 63–78.

[27] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *5th International Conference on the Unified Modeling Language (UML 2002)*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer-Verlag, 2002.

[28] V. Lotz. Threat scenarios as a means to formally develop secure systems. *Journal of Computer Security 5*, pages 31–67, 1997.

[29] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, 17(3):93–102, 1996.

[30] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 237–250. IEEE Computer Society, 2000.

[31] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO – The CADE-13 Systems. *Journal of Automated Reasoning (JAR)*, 18(2):237–246, 1997.

[32] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[33] B. Pfitzmann. Higher cryptographic protocols, 1998. Lecture Notes, Universität des Saarlandes.

[34] P. Ryan and S. Schneider. An attack on a recursive authentication protocol. *Information Processing Letters*, 65:7–10, 1998.

[35] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, Reading, MA, 2001.

[36] J. Schumann. Automatic verification of cryptographic protocols with SETHEO. In W. McCune, editor, *14th International Conference on Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Computer Science*, pages 87–100. Springer-Verlag, 1997.

[37] G. Stenz and A. Wolf. E-SETHEO: An automated^3 theorem prover. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000)*, volume 1847 of *Lecture Notes in Computer Science*, pages 436–440. Springer-Verlag, 2000.

[38] G. Sutcliffe and C. Suttner. The TPTP problem library for automated theorem proving, 2001. Available at http://www.tptp.org.

[39] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Computer Science*, pages 314–328, 1999.

[40] G. Wimmel. *Model-based Development of Security-Critical Systems*. PhD thesis, TU München, 2005.