Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Program in Computer Science

Bachelor's Thesis

Expi2Java – An Extensible Code Generator for Security Protocols

submitted by

Alex Busenius

on October 28, 2008

Supervisor

Prof. Dr. Michael Backes

Advisor

Cătălin Hrițcu

Reviewers

Prof. Dr. Michael Backes
Dr. Matteo Maffei

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, October 28, 2008
Alex Busenius

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, October 28, 2008
Alex Busenius

Abstract

This thesis presents expi2java, an extensible code generator for security protocols. We use a variant of Spi calculus [AG99] for the protocol specifications and complement it with an expressive type system that is designed to reject inconsistent protocols. This type system features subtyping and parametric polymorphism. It is able to handle the types of nested terms, specialized channels and even low-level term configurations. Expi2java is highly customizable, easily extensible and generates interoperable Java code. We show the flexibility of our approach by generating an implementation of the Transport Layer Security (TLS) protocol.

Acknowledgments

I am very grateful to my advisor, Cătălin Hriţcu, whose enthusiasm motivated me to keep on working on this thesis over the last months. He gave me the needed background knowledge about the type systems and semantics, shared with me many ideas, discussed the occurred problems and always provided a lot of useful comments, corrections and suggestions.

I would like to thank Alfredo Pironti, Davide Pozza and Riccardo Sisto for providing the complete source code and documentation for the spi2java framework that we used as a starting point for expi2java. The discussions we had about the design of spi2java, as well as the comments on our extensions and the report were very helpful. I also appreciate the help of Violeta Ivanova who worked with me on the seminar report and the previous version of expi2java.

I would also like to thank Dominique Unruh for the source code of the ZK Compiler tool. His contribution saved us a lot of time and effort.

In conclusion, I wish to thank the countless contributors from the Open Source community who have developed numerous powerful tools and libraries I used.

Contents

1 Introduction

1.1 Motivation

One of the important directions in security research is ensuring the correctness of the design and the implementation of security protocols. The protocols can be formally modelled and analyzed in different abstract calculi, e.g., in the Spi calculus [AG99]. These formal models can be used by various verification tools, for instance by ProVerif [Bla01], to automatically prove the desired security properties, such as authentication, confidentiality and integrity. Furthermore, the abstract models can be analyzed by many type systems [Aba99, AB01, GJ04, HJ06, BFM07, BCFM07, FGM07, BHM08a] that statically enforce some security properties.

However, a secure formal model of a protocol does not necessarily lead to a secure implementation of this protocol. One of the possible approaches to guarantee correctness of the implementation is to generate the implementation automatically from the formal protocol specification. In fact, various experimental tools for automatic code generation exist, such as CIL2Java by J. Millen and F. Muller [MM01], the Sprite tool by B. Tobler [Tob05] and Spi2java by A. Pironti, R. Sisto, L. Durante and D. Pozza [PSD04, PS07].

Ideally, both the translation from the abstract model to the implementation language and the target language should be formalized and proved correct (i.e., proved at least to preserve the security properties of the initial protocol). Unfortunately, proving the translation correct is a non-trivial task even if one considers cryptography as a fully reliable black box and uses a simple implementation language. In the case of mainstream programming languages, such as Java, C#, C or C++, this task is very hard to achieve and requires correctness proofs about a large subset of the used target language. For instance, protocols such as SSH or TLS require networking, cryptography, concurrency and rely on the low-level bitstring representation of the data.

Even though a provably sound code generation of complex protocols in a real-world programming language is still a vision, there are many other reasons to use the code generators. The alternative to generation is the manual implementation. It is error-prone, slow and requires the programmers to write a large amount of code for every used protocol. The implementation of a code generator, in contrast, is relatively small and can be tested more extensively, resulting in an increased confidence in the correctness of the generated protocol implementations. The use of the code generator also leads to lowering the overall costs by shorter development times.

One of the most important features of a code generator, aiming at the generation of real-life security protocols, is the interoperability with the existing implementations of the protocols. In order to achieve this, the implementation must exactly follow the protocol specification and be able to work with common standards, (like the X.509 standard for digital certificates), use the corresponding realizations of the cryptographic primitives, (e.g., popular block encryption schemes, such as DES, AES and RSA), and have full control over the setup of the communication channels. The user must be able to customize the generation process according to the informal details of the specification, which cannot be easily modelled in the formal calculus.

The calculus used to model the protocols must be flexible and expressive enough to model complex protocols as good as possible. It should be easy to extend the calculus and the set of cryptographic primitives and data types available for the code generation.

The implementations of security protocols are used in a large amount of existing software, thus, the integration of the generated code into the existing applications is very important. Additionally, the generated code should be efficient enough to be used as a replacement of the existing optimized implementations used by real applications.

The quality and correctness of the implementation of the code generator is crucial for the quality and correctness of the generated code. The tool should have a simple, modular design that is both flexible and easy to maintain. The code generator should be easy to use and integratable into the popular development tools.

Our long-term goal is to combine as many of these important features together as possible. We need to be able to generate large real-world protocols, support also the complex primitives such as zero-knowledge proofs, be flexible, configurable and generate provably-correct, secure code.

1.2 Contributions

In this thesis we introduce the expi2java tool, an extensible Java code generator for security protocols. Expi2java originated as an extension of the spi2java framework [PSD04] and evolved in the course of time into an independent project.

We introduce a type system with nested types, aimed at the needs of our code generator. This type system ensures that specialized data types are used in a consistent way and prevents the usage of incompatible cryptographic primitives. Using parametric polymorphism and subtyping, we can represent the complex types of nested terms with only a few core type constructors.

Another important contribution is the notion of configurations, which allow the user to specify all needed low-level information about every term with minimal effort. The configurations also offer a simple way to solve the issues that originate from the differences between the abstract representation and the concrete implementation of channels that are described in more detail in Section 4.2.

One of our first extensions to the spi2java [BI08] was using an extensible Spi calculus with user-defined constructors and destructors [AB05]. In this thesis, we have adapted this calculus to use the configurations. The resulting Extensible Spi Calculus is defined in Chapter 2.

The abstract concepts mentioned before are implemented in the expi2java tool. Expi2java is able to generate interoperable Java implementations from protocol specifications written in the Extensible Spi Calculus. The user can customize the code generation process without the need to change the implementation of the code generator. We provide a convenient way to extend the calculus, the type system and the set of the available cryptographic primitives and data types used by the code generator. The cryptographic primitives are implemented in a runtime library that relies on the standard Java cryptographic providers.

A case study on the Transport Layer Security (TLS) protocol [DA99] illustrates the potential of expi2java. The generated TLS implementation instantiates a secure AES-256 encrypted connection to a given web server using the RSA key exchange and downloads a web page using the HTTP over TLS [Res00].

1.3 Outline

Chapter 2 describes the Extensible Spi Calculus that we use to model security protocols and the concrete file format supported by expi2java. In Chapter 3 we define the type system with nested types, prove that it preserves typing on evaluation and explain how the types, configurations, constructors and destructors can be defined in our tool. The design and main features of expi2java, as well as some details on the generated implementations of the protocols are described in Chapter 4. The case study on the implementation of the TLS protocol is discussed in Chapter 5. Finally, in Chapter 6 we give a short overview of the related projects, compare expi2java to its predecessor and provide directions for future work.

2 An Extensible Spi Calculus

We use a variant of the Spi calculus, a process calculus for modelling cryptographic protocols. The calculus is the same as the one by M. Abadi and B. Blanchet [AB05], but we extend it with configurations (Section 3.2) and give a new type system for it (Chapter 3). Throughout this thesis we will refer to this calculus as the *Extensible Spi Calculus*, since it can be easily extended by the user in various ways. In this chapter we define the syntax and semantics of this calculus.

2.1 Abstract Syntax

2.1.1 Terms and Constructors

In the Extensible Spi Calculus, *terms* are used to model data and *processes* are used to model the behavior of the protocol participants and the communication between them. The set of terms contains *names* (which represent constant data), *variables* and *constructor applications*.

Table 2.1 Terms and constructors

K, L, M, N ::=		terms
	a, b, m, n, k	names
	x, y, z, v, w	variables
	$f(M_1, \ldots, M_n)$	constructor application (f of arity n)

$f \quad ::= \quad \mathsf{enc}^2_c,\ \mathsf{enca}^2_c,\ \mathsf{pk}^1_c,\ \mathsf{sign}^2_c,\ \mathsf{vk}^1_c,\ \mathsf{h}^1_c,\ \mathsf{true}^0,\ \mathsf{false}^0,\ \mathsf{pair}^2,\ \mathsf{succ}^1_c,\ \mathsf{zero}^0_c$

Notation: We use u to refer to both names and variables.
Notation: We use c, d, e and $*$ to denote configurations.
Notation: The default configuration $*$ can be omitted, i.e., $\mathsf{true} = \mathsf{true}_*$.
Notation: The numbers in constructor names denote arities, i.e., enc^2_c has arity 2.

Constructors are function symbols that are used to build terms. For instance, the constructors enc_c and enca_c represent symmetric and asymmetric encryption; pk_c extracts a public key from the given private key; sign_c constructs a digital signature; vk_c extracts the corresponding verification key from the signature key; h_c constructs hashes; the nullary constructors true and false represent the corresponding boolean values; finally, zero_c and succ_c are used to represent integers.

The only difference between the calculus from [AB05] and our calculus is the notion of *configurations*. In our Extensible Spi Calculus, every constructor and destructor has a corresponding configuration. If the configuration is not set explicitly, the default configuration $*$ is used. The default configuration is compatible with any other configuration.

The configurations have many uses, amongst others, they allow for specifying sets of types, constructors and destructors meant to be used together. For example, we can have a constructor $\mathsf{enc}_{\mathsf{DES}}$, a corresponding destructor $\mathsf{dec}_{\mathsf{DES}}$ and two types $\mathsf{SymEnc}_{\mathsf{DES}}\langle T\rangle$ and $\mathsf{SymKey}_{\mathsf{DES}}\langle T\rangle$, that together represent the DES encryption. We will describe configurations in more detail in Sections 4.2 and 3.2.

Note that the set of constructors and destructors used in this calculus is not fixed. Using a flexible file format described in Section 3.8, the user can easily add new primitives, change existing definitions and remove constructors and destructors according to his or her needs. The constructors from Table 2.1 and destructors from Table 2.2 define a set of commonly used cryptographic primitives and data types that we will use in the examples we give in this thesis.

2.1.2 Destructors

Destructors are partial functions that can be applied to terms. The semantics of destructors is defined by the reduction relation $\Downarrow$, which can either succeed and provide a term N as the result (denoted by $g(M_1, \dots, M_n) \Downarrow N$), or fail (denoted by $g(M_1, \dots, M_n) \not\Downarrow$). Every constructor used in a reduction rule should have the same configuration as the destructor, otherwise the reduction will fail, e.g., $\mathsf{dec}_{\mathsf{AES}}(\mathsf{enc}_{\mathsf{RC4}}(M, K), K) \not\Downarrow$.

Table 2.2 Destructors $\qquad g(M_1, \dots, M_n) \Downarrow N$

$g \ ::= \ \mathsf{dec}_c^2, \mathsf{deca}_c^2, \mathsf{msg}_c^1, \mathsf{ver}_c^2, \mathsf{eq}^2, \mathsf{first}^1, \mathsf{second}^1, \mathsf{pre}_c^1$

$\mathsf{dec}_c(\mathsf{enc}_c(M, K), K)$	$\Downarrow$	M	$\mathsf{first}(\mathsf{pair}(M, N))$	$\Downarrow$	M
$\mathsf{deca}_c(\mathsf{enca}_c(M, \mathsf{pk}_c(K)), K)$	$\Downarrow$	M	$\mathsf{second}(\mathsf{pair}(M, N))$	$\Downarrow$	N
$\mathsf{msg}_c(\mathsf{sign}_c(M, K))$	$\Downarrow$	M	$\mathsf{pre}_c(\mathsf{succ}_c(M))$	$\Downarrow$	M
$\mathsf{ver}_c(\mathsf{sign}_c(M, K), \mathsf{vk}_c(K))$	$\Downarrow$	M	$\mathsf{pre}_c(\mathsf{zero}_c)$	$\Downarrow$	zero_c
$\mathsf{eq}(M, M)$	$\Downarrow$	true			

Notation: We write $g(M_1, \dots, M_n) \not\Downarrow$ if none of the rules above applies, i.e., the destructor fails.

The destructors dec_c and deca_c decrypt messages encrypted with enc_c and enca_c respectively; ver_c verifies the given signature and returns the signed term on success, msg_c returns the signed term without checking the signature; eq compares two terms and returns true on success; first and second allow for extracting the corresponding components of pairs; finally, pre_c returns the predecessor of the given integer if possible or zero_c otherwise.

2.1.3 Processes

The *processes* are used to model the behavior of protocol participants and the communication between them [AB05]. A specific characteristic is that a replication process can only be followed by the input process, as in [FGM07, BHM08a], since this is the most common way to use replication in a protocol specification.

The $\mathsf{out}(M, N).P$ process outputs the message N on the channel M and then behaves as the process P; the $\mathsf{in}(M, x).P$ process inputs the message N on the channel M and then behaves as the process $P\{N/x\}$; the replication process $!\,\mathsf{in}(M, x).P$ behaves as an

Table 2.3 Syntax of processes

P, Q, R	$::=$		processes
		$\mathsf{out}(M, N).P$	output
		$\mathsf{in}(M, x).P$	input
		$!\,\mathsf{in}(M, x).P$	replicated input
		$\mathsf{new}\ a : T.P$	restriction
		$P \mid Q$	parallel composition
		$\mathbf{0}$	null process
		$\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q$	destructor evaluation

Notation: $\widetilde{M} = M_1, \ldots, M_n$.

unbounded number of $\mathsf{in}(M, x).P$ processes executed in parallel; the restriction process $\mathsf{new}\ a : T.P$ generates a fresh name a of type T and then behaves as P; the parallel composition $P \mid Q$ behaves as the processes P and Q executed in parallel; $\mathbf{0}$ does nothing; $\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q$ applies the destructor g to the terms $\widetilde{M}$ and then, in case the destructor succeeds providing the term N, it behaves as $P\{N/x\}$, otherwise it behaves as Q. We denote by $\{M/x\}$ the capture-avoiding substitution of x by M.

We denote by $\mathit{fn}(P)$ the set of free names in P, by $\mathit{fv}(P)$ the set of free variables and by $\mathit{free}(P)$ the set of free names and variables. We say that P is closed if it does not have any free variables.

2.1.4 Example: Perrig-Song Mutual Authentication Protocol

The Perrig-Song mutual authentication protocol[1] given in Figure 2.1 uses a shared key to authenticate two participants and send an encrypted message. We will use this protocol as a running example throughout this thesis.

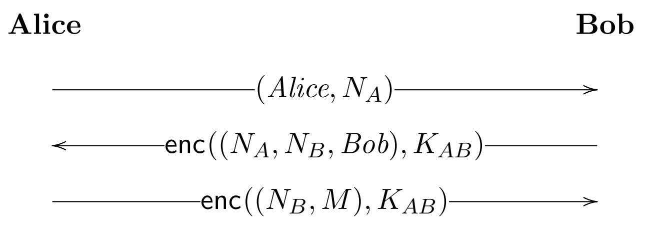
Alice
Bob
(Alice, N_A)
$\mathsf{enc}((N_A, N_B, \mathit{Bob}), K_{AB})$
$\mathsf{enc}((N_B, M), K_{AB})$

Figure 2.1: Perrig-Song mutual authentication protocol

The Perrig-Song protocol is composed of three message exchanges. First, the initiator of the protocol, Alice, sends her identity together with a fresh nonce N_A to the responder, Bob. Bob encrypts the nonce N_A, which he received from Alice together with another fresh nonce N_B and his identity with the shared key K_{AB} and sends this encryption to Alice. Alice receives the encrypted message, decrypts it with the shared key K_{AB} and checks that the nonce N'_A inside is the same as her nonce N_A. If the nonces match, she encrypts the received nonce N_B and a message M she wanted to send with the shared key K_{AB} and sends

[1]Can be found in examples for CAPSL by J. Millen et al. [MM01], at http://www.csl.sri.com/users/millen/capsl/examples.html

the resulting message to Bob. Bob decrypts the message and checks that the received nonce N'_B is the same as his nonce N_B. If the nonces match, the protocol completes successfully.

This protocol can be written in the Extensible Spi Calculus as follows:

```
new k. (
    new M, Na.
    out(c, pair(Alice, Na)).
    in(c, e).
    let msg = dec(e, k) in
        let tmp = first(msg) in
            let Na1 = first(tmp) in
                let ok = eq(Na, Na1) in
                    let Nb = second(tmp) in
                        out(c, enc(pair(Nb, M), k)).
                            0
|   in(c, x).
    let name = first(x) in
        let Na = second(tmp) in
            new Nb.
            out(c, enc(pair(pair(Na, Nb), Bob), k)).
            in(c, msg).
            let p = dec(msg, k) in
                let Nb1 = first(p) in
                    let ok = eq(Nb, Nb1) in
                        let M = second(p) in
                            0 )
```

Listing 1: Perrig-Song mutual authentication protocol in Extensible Spi Calculus

2.2 Operational Semantics

The semantics of the calculus is standard and is defined by the usual structural equivalence relation ($P \equiv Q$) and an internal reduction relation ($P \to Q$).

Structural equivalence relates the processes that are considered equivalent up to syntactic re-arrangement. It is the smallest equivalence relation satisfying the rules in Table 2.4.

Table 2.4 Structural equivalence $P \equiv Q$

(Eq-Zero-Id)	$P \mid \mathbf{0}$	$\equiv$	P
(Eq-Par-Comm)	$P \mid Q$	$\equiv$	$Q \mid P$
(Eq-Par-Assoc)	$(P \mid Q) \mid R$	$\equiv$	$P \mid (Q \mid R)$
(Eq-Scope)	$\mathsf{new}\ a : T.(P \mid Q)$	$\equiv$	$P \mid \mathsf{new}\ a : T.Q,\ \textit{if}\ a \notin \textit{fn}(P)$
(Eq-Bind-Swap)	$\mathsf{new}\ a_1 : T_1.\mathsf{new}\ a_2 : T_2.P$	$\equiv$	$\mathsf{new}\ a_2 : T_2.\mathsf{new}\ a_1 : T_1.P,\ \textit{if}\ a_1 \neq a_2$
(Eq-Ctxt)	$\mathcal{E}[P]$	$\equiv$	$\mathcal{E}[Q],\ \textit{if}\ P \equiv Q$

Where $\mathcal{E}$ stands for an evaluation context, i.e., a context of the form $\mathcal{E} = \mathsf{new}\ \widetilde{a} : \widetilde{T}.([\,] \mid P)$

Internal reduction defines the semantics of communication and destructor application. It is the smallest relation on closed processes satisfying the rules in Table 2.5.

Table 2.5 Internal reduction $P \to Q$

(Red-I/O)	$\mathsf{out}(a, M).P \mid \mathsf{in}(a, x).Q$	$\to$	$P \mid Q\{M/x\}$
(Red-!I/O)	$\mathsf{out}(a, M).P \mid !\,\mathsf{in}(a, x).Q$	$\to$	$P \mid Q\{M/x\} \mid !\,\mathsf{in}(a, x).Q$
(Red-Destr)	$\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q$	$\to$	$P\{N/x\},\quad \textit{if}\ g(\widetilde{M}) \Downarrow N$
(Red-Else)	$\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q$	$\to$	$Q,\quad \textit{if}\ g(\widetilde{M}) \not\Downarrow$
(Red-Ctxt)	$\mathcal{E}[P]$	$\to$	$\mathcal{E}[Q],\quad \textit{if}\ P \to Q$
(Red-Eq)	$P \to Q,\quad \textit{if}\ P \equiv P', P' \to Q',\ \text{and}\ Q' \equiv Q$		

2.3 File Format (Expi)

The input file format for the expi2java tool is called Expi. The concrete syntax used in Expi files is mostly compatible with the input syntax used by ProVerif [Bla01, ABF05], a popular cryptographic protocol verifier. We use the same syntax for all processes and terms, only the declarations of constructors and destructors have mandatory type annotations and are declared in included Exdef files for convenience. We will describe Exdef files in Section 3.8.

All extensions to the calculus that are not supported by ProVerif are added in form of additional annotations, written as ProVerif comments. A similar approach is used in the zero-knowledge type checker tool [BHM08b]. This allows for a simple translation into a form suitable for ProVerif by a simple preprocessing step. We have implemented this step as a part of the pretty-printer in the expi2java tool, which can transform any Expi input file into a nicely formatted ProVerif input file.

The Perrig-Song protocol from Section 2.1.4 can be written in Expi as follows:

```
(*
 * Perrig-Song mutual authentication protocol
 *)

(*# include "../exdef/default.exdef" *)

(* Free names *)
free Alice.
free Bob.
free c.


(** Named processes *)

let testA (*# public *) =
    new M;
    new Na;
    (* Msg 1. A->B: pair(Alice, Na) *)
    out(c, pair(Alice, Na));
    (* Msg 2. B->A: enc(pair(pair(Na, Nb), Bob), K_AB) *)
    in(c, emsg);
    let msg1 = dec(emsg, k) in
        let tmp = first(msg1) in
            let Na1 = first(tmp) in
                let ok = eq(Na, Na1) in
                    let Nb = second(tmp) in
                        (* Msg 3. A->B: enc(pair(Nb, M), K_AB) *)
```

```
                              out(c, enc(pair(Nb, M), k)).

let testB =
    (* Msg 1. A->B: pair(Alice, Na) *)
    in(c, tmp);
    let name = first(tmp) in
        let Na = second(tmp) in
            new Nb;
            (* Msg 2. B->A: enc(pair(pair(Na, Nb), Bob), K_AB) *)
            out(c, enc(pair(pair(Na, Nb), Bob), k));
            (* Msg 3. A->B: enc(pair(Nb, M), K_AB) *)
            in(c, mmm);
            let pp = dec(mmm, k) in
                let Nb1 = first(pp) in
                    let M1 = second(pp) in
                        let ok = eq(Nb1, Nb) in
                            0.

(** Main process *)
process
    new k;
    (testA | testB)
```

Listing 2: Perrig-Song mutual authentication protocol in Expi (untyped)

Compared to the abstract calculus described in Section 2.1, the concrete syntax used in the Expi files has several additional features (also supported by ProVerif).

A process can be given a name using the "**let** process_name = P." construct to improve the readability. Such *named processes* are often used to represent the protocol participants, as in Listing 2. In this case the main process (starting with the keyword **process**) only contains a restriction followed by the named process of the two participants executed in parallel.

The *free names* are declared using the "**free** a." construct. Every name used by a process or a term should either be previously generated using the new $a.P$ process or be declared as free. If a free name declaration has the optional **private** keyword, it is not known by the adversary in ProVerif. We support this feature for compatibility reasons.

The code parts enclosed in *(∗ ∗)* are ProVerif comments. We use them to add several extensions to the language while staying compatible with ProVerif.

One of the extensions is the *include directive*: *(∗# include "../exdef/default.exdef" ∗)*. It is used to include External Definitions files, which contain the definitions of types, configurations, constructors and destructors. The file `default.exdef` contains the set of types and cryptographic primitives used by all example protocols provided with expi2java. The syntax of Exdef files is defined in Section 3.8.

Another extension is the *process kind*, which can be specified as an optional annotation of a named process: "**let** process_name*(∗# keyword ∗)* = P.", where *keyword* is one of {public, private, skip}. The protocol participants are represented as *public* processes (default), *private* processes represent integral parts of the public processes and processes marked with *skip* are ignored during the code generation. Private processes can be used to split large public processes into several smaller processes for readability or define helper processes that should not be implemented (e.g., for verification with ProVerif). In Listing 2, both named processes are public, but we omitted the optional process kind annotation for the second process.

The last extension is the *type annotation*: `(*: <type annotation> *)`. The type annotations were omitted in Listing 2, since the type system will be introduced in the next chapter. A fully annotated example of this protocol can be found in Section 3.7, Listing 4.

All other ProVerif comments, (i.e., the ones not starting with "`(*:`" or "`(*#`") are treated as comments in expi2java as well.

3 Type System with Nested Types

We motivate the use of a type system in Section 3.1. In Section 3.2 we introduce the configurations. In Sections 3.3 and 3.4 we define our goals and explain the advantages of nested types. We define our type system in Section 3.5 and prove that it preserves typing on evaluation in Section 3.6. In Section 3.7 we give an example of typed protocol. Finally, Section 3.8 explains how the types, configurations, constructors and destructors can be defined in our tool.

3.1 The Need for a Type System

The calculus we presented in Chapter 2 didn't consider types. This is reasonable in certain settings. For example, when we verify a protocol model in ProVerif, we might only want to know if some term is leaked to the adversary and in this case we do not need to know what this term is.

However, if we want to generate code for the same protocol in an explicitly-typed language like Java, then we need to provide (or infer) a type for every term, since in the generated code terms are represented by Java variables of different types. Moreover, we also need to know the types of all methods that are used to implement processes, constructors and destructors.

Consider the following example:

```
fun pair/2.
fun enc/2.

reduc dec(enc(x, y), y) = x.
reduc id(x) = x.
reduc first(pair(x, y)) = x.
reduc second(pair(x, y)) = y.

query attacker:m;
      attacker:k.

let pA =
    out(c, enc(m, k));
    in(k, e);
    let y = dec(e, c) in (* always fails *)
        out(c, y);
        out(enc(k, m), pair(y, k)).

let pB =
    in(c, e);
    out(k, e);
    in(c, y);
```

```
23         let ekm = id(enc(k, m)) in
24             in(ekm, y).
25
26 free c.
27
28 process
29     new m; new k;
30     ( pA | pB )
```

Listing 3: Unimplementable protocol in Extensible Spi Calculus

This is a valid protocol specification written in ProVerif syntax. In addition to the constructors and destructors defined in Section 2.1 it uses the identity destructor id. The **query** in the lines 9-10 is a ProVerif-specific construct, used to define which properties should be proved. In this case, we ask whether the adversary can know the terms m or k.

This protocol is accepted by ProVerif and is even secure, since the attacker is not able to gain the knowledge of neither m, nor k. Nevertheless, it would be very difficult to implement this protocol in a usual programming language, since it uses the terms in a totally incompatible way. The term c is first used as a channel in the lines 13 and 20, where the participants send the message m encrypted with the key k. Then, in lines 14 and 21, this encrypted message is sent back over the key k and is decrypted using the channel c. Of course, this decryption fails, since $c \neq k$, but the code generator would also need to generate a code that passes the Java type-checker for the whole protocol, including the part where a pair of terms should be sent over an encryption.

One of the purposes of our type system is to reject such “unimplementable” protocols before the code generation phase.

3.2 Configurations

The configurations are a crosscutting concept that solves several problems. First, we need a convenient way to define subsets of related types, constructors and destructors without explicitly defining them with different names, since multiple similar definitions are hard to use and cause unneeded duplication. This feature is an integral part of the Extensible Spi Calculus and our type system. It is used there to detect errors, such as decrypting a DES encrypted message with an AES key.

The specifications of real-world protocols contain a lot of information that cannot be directly expressed in Spi calculus, but still should be taken into account during the code generation. For example, there are often exact rules of how the messages should look like on the bit-string level, which data types should be used, which size they should have etc. We need a way to express that using the type system in order to make a model that matches the protocol as closely as possible, ideally without any implicit assumptions on the behavior of the implementation classes. In order to limit the number of types in a manageable range, they should stay as abstract as possible and allow for defining of specialized types with different characteristics, such as AES and DES encryptions.

In order to solve these problems, we introduced the concept of configurations and implemented it as simple key-value mappings with some special keys. The configurations are defined in the Exdef files using the syntax defined in Section 3.8. There is also a way to extend the configurations, similar to the inheritance in object-oriented programming. This allows the user to define a configuration that only differ in several keys from another one in a convenient way.

3.3 Goals

Using a type system, we can statically check whether a given protocol is well-typed. This process is fast, reliable and, provided that the type system is sound, implies that every well-typed protocol is also valid according to the criteria enforced by this type system.

First, we want every term to have a type corresponding to its data type, such as integer, string, symmetric encryption, pair etc. The corresponding parameter of the constructors and destructors and their return types should also reflect their intended use.

Additionally, we also need to enforce the proper usage of constructor and destructor configurations and integrate this configurations with the types.

Since we cannot predict all the possible uses of this type system, we need a simple and small core and a possibility to extend this type system with custom types, if needed. This approach also leads to a clean design and a simpler implementation of the type checker.

3.4 The Need for Nested Types

One possible solution is to use a hierarchy of simple types, as in the spi2java framework by A. Pironti, R. Sisto, L. Durante and D. Pozza [PSD04]. Simple types fulfill most of our goals and, with some additional low-level information, allow to generate interoperable code, as it was shown on the example of an SSH client [PS07].

However, simple types also have some limitations. They are not able to carry the information from the nested terms, which means that the discarded information about the term structure needs to be provided by the user every time one of the nested terms is extracted using a destructor. This information can be recovered in some simple cases using the type inference [BI08], but for more complex protocols it leads to an increased amount of type annotations.

The type hierarchy used in the simple type system has the form of a tree with one top type called Message. This allows for defining e.g., encryption as a constructor taking two parameters, one of type Message and the other of type Key, and returning an encrypted message of type Encryption. This is desirable if we want to represent an universal encryption that can take a message of any type, convert it to Message using the subsumption rule and return an encrypted message. In case of the specialized encryption functions, for example those that should only encrypt strings, using this type system would lead to several constructors and destructors that only differ in their names and parameter types, and several specialized encryptions. It is also not possible to express specialized channels, since the in and out processes need to take the most abstract type of channels and messages.

In order to have an expressive and flexible type system we decided to use not only subtyping, but also parametric polymorphism [CG92, Pie02]. This allows us to use a small number of "generically" typed constructors and destructors and still be able to specialize them. The parametric types can be nested, which naturally corresponds to the types of the nested terms and allows us to keep more information about the inner terms even after several destructor or constructor applications. The nested types also allow for expressing the relation between the type of a channel and the type of messages sent and received over it, modelling the fact that an encryption and the corresponding key work on messages of the same type, using only a few base types.

3.5 Definitions

3.5.1 Types

Our type system has only two mandatory types, Top and $\mathsf{Channel}_c\langle T\rangle$, all other types are user-defined. Top is supertype of any other type. The type $\mathsf{Channel}_c\langle T\rangle$ must also be always present, since it is hardcoded in some of the process typing rules from Table 3.8. The other types we define in Table 3.1 represent the types of the cryptographic primitives and some common data types. They are needed for typing the constructors and destructors defined in Section 2.1.

Table 3.1 Types

$$
\begin{array}{rlllll}
T, U ::= & \mathsf{Top} & \mathsf{Channel}_c\langle T^{\circ}\rangle & \mathsf{SymEnc}_c\langle T^{+}\rangle & \mathsf{PubEnc}_c\langle T^{+}\rangle & \mathsf{Signed}_c\langle T^{+}\rangle \\
 & \mathsf{Bool}_c & \mathsf{Pair}_c\langle T^{+}, U^{+}\rangle & \mathsf{SymKey}_c\langle T^{-}\rangle & \mathsf{PubKey}_c\langle T^{-}\rangle & \mathsf{SigKey}_c\langle T^{-}\rangle \\
 & \mathsf{Int}_c & \mathsf{Hash}_c\langle T^{+}\rangle & & \mathsf{PrivKey}_c\langle T^{-}\rangle & \mathsf{VerKey}_c\langle T^{-}\rangle
\end{array}
$$

Notation:	Let $\widetilde{x} : \widetilde{T}$ denote $x_1 : T_1, \ldots, x_n : T_n$ for some n
Notation:	We use $+$, $-$ and $\circ$ to denote covariant, contravariant and invariant subtyping
Definition:	T is generative if $T \in \{\mathsf{Int}_c, \mathsf{Channel}_c\langle T\rangle, \mathsf{SymKey}_c\langle T\rangle, \mathsf{PrivKey}_c\langle T\rangle, \mathsf{SigKey}_c\langle T\rangle\}$
Note:	Two types are equal if their names, nested types and configurations are equal

The *variance* (see Pierce, TAPL [Pie02], Chapter 15.2) of every parameter in nested types controls the sense of the subtyping relation defined in Table 3.4. It is reversed for *contravariant* parameters, runs in the same direction for *covariant* parameters and requires the *invariant* parameters to be the same. The parameters of all user-defined types can have arbitrary variance, however, the variance needs to be chosen carefully, since the subtyping rules can be used to circumvent the type system on constructors and destructors applications in some cases. We have chosen the variances of the types in Table 3.1 in the way that does not allow such misuses (see Proposition 3.6.9 (Typing Destructors Consistent) and Theorem 3.6.10 (Subject-Reduction)).

Types can be declared to be *generative*. Only generative types are allowed to be used in a restriction process. This allows the user to selectively restrict the generation of fresh names only to the types where it actually makes sense (i.e., where the corresponding implementation is available) in a clean way. In the default set of types we allow to generate fresh integers that can be used as nonces in the protocols, to create fresh keys and open new communication channels using the restriction process.

Similar to the constructors and destructors, every type (with the exception of Top) also has a corresponding configuration. Two differently configured types are considered to be different by the type system. The type and the configuration of any name or variable inside a protocol can be chosen independently from each other via a type annotation. This feature is used to provide the low-level information needed for the code generation for every type in a consistent way, without the need to define new specialized primitives or types. For example, assume we want to decrement a 64 bit long nonce Na of type $\mathsf{Int}_{\mathsf{64bit}}$ and send it on a channel: "**let** Na1 = pre(Na) **in** **out**(c, Na1)". If the type system would just check that the type of Na1 is an Int, we could accidentally configure Na1 to be only 16 bit long (i.e., $\mathsf{Int}_{\mathsf{16bit}}$), and get an incorrect behavior of the generated code (the receiver would only get a part of Na).

3.5.2 Typing Rules and Judgments

Our type system is based on the typing judgments defined in Table 3.2.

Table 3.2 Typing judgments

$\Gamma \vdash \diamond$	well-formed environment
$\Gamma \vdash T <: U$	subtyping
$f : (T_1, \ldots, T_n) \mapsto T$	constructor typing
$g : (T_1, \ldots, T_n) \mapsto T$	destructor typing
$\Gamma \vdash M : T$	term typing
$\Gamma \vdash P$	well-typed process

Notation: We use $\Gamma \vdash \mathcal{J}$ to denote a typing judgment, where $\mathcal{J} \in \{\diamond,\ T <: U,\ M : T,\ P\}$

Table 3.3 Well-formed environment $\Gamma \vdash \diamond$

Env-Empty
$$\emptyset \vdash \diamond$$

Env-Binding
$$\frac{\Gamma \vdash \diamond \qquad u \notin dom(\Gamma)}{\Gamma, u : T \vdash \diamond}$$

Definition: $dom(\emptyset) = \emptyset$; $dom(\Gamma, u : T) = dom(\Gamma) \cup \{u\}$

The *typing environment* Γ is a list containing name and variable bindings of the form $u : T$. A typing environment is *well-formed*, denoted by $\Gamma \vdash \diamond$, if no name or variable is bound more than once. All other typing judgments check that the typing environment is well-formed.

Our type system supports *subtyping* with a top type Top. This allows us to omit the information about the nested types where it does not matter and makes it possible to express the types of "universal" keys, encryptions, channels etc., which should work with any type. The subtyping relation is a partial order (see Lemma 3.6.7 (Subtyping Partial Order)). It is defined by the rules in Table 3.4.

We can also represent deep type hierarchies like the one from the type system used in spi2java by nesting the types several times. For instance, the relation

$$\mathsf{Int} <: \mathsf{Name}, \mathsf{Bool} <: \mathsf{Name}, \mathsf{Name} <: \mathsf{Message}$$

can be modelled using the nested types as

$$\mathsf{Name}\langle\mathsf{Int}\rangle <: \mathsf{Name}\langle\mathsf{Top}\rangle, \mathsf{Name}\langle\mathsf{Bool}\rangle <: \mathsf{Name}\langle\mathsf{Top}\rangle, \mathsf{Name}\langle\mathsf{Top}\rangle <: \mathsf{Top},$$

assuming that $\mathsf{Name}\langle T^{+}\rangle$ is a covariant type.

Table 3.4 Subtyping $\Gamma \vdash T <: U$

Sub-Refl
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash T <: T}$$

Sub-Top
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash T <: \mathsf{Top}}$$

Sub-Nested
$$\frac{\begin{array}{rlcl} \forall i \in [1,n]. & (\mathit{variance}(T,i) = + & \Rightarrow & \Gamma \vdash T_i <: U_i) \\ \wedge & (\mathit{variance}(T,i) = - & \Rightarrow & \Gamma \vdash U_i <: T_i) \\ \wedge & (\mathit{variance}(T,i) = \circ & \Rightarrow & T_i = U_i) \end{array}}{T\langle T_1,\dots,T_n\rangle <: T\langle U_1,\dots,U_n\rangle}$$

Notation: $\mathit{variance}(T,i)$ returns the variance in the i-th parameter of the type T (see Table 3.1)

The types of all constructors and destructors from Section 2.1 are defined in Tables 3.5 and 3.6. Most of them are implicitly parametrized by the one or more types T and U that can be replaced by any type from Table 3.1, resulting in a well-typed instance of the corresponding constructor or destructor.

Table 3.5 Typing constructors $f : (T_1,\dots,T_n) \mapsto T$

$\mathsf{enc}_c : (T, \mathsf{SymKey}_c\langle T\rangle) \mapsto \mathsf{SymEnc}_c\langle T\rangle$	$\mathsf{true} : () \mapsto \mathsf{Bool}$
$\mathsf{enca}_c : (T, \mathsf{PubKey}_c\langle T\rangle) \mapsto \mathsf{PubEnc}_c\langle T\rangle$	$\mathsf{false} : () \mapsto \mathsf{Bool}$
$\mathsf{pk}_c : (\mathsf{PrivKey}_c\langle T\rangle) \mapsto \mathsf{PubKey}_c\langle T\rangle$	$\mathsf{pair} : (T, U) \mapsto \mathsf{Pair}\langle T, U\rangle$
$\mathsf{sign}_c : (T, \mathsf{SigKey}_c\langle T\rangle) \mapsto \mathsf{Signed}_c\langle T\rangle$	$\mathsf{succ}_c : (\mathsf{Int}_c) \mapsto \mathsf{Int}_c$
$\mathsf{vk}_c : (\mathsf{SigKey}_c\langle T\rangle) \mapsto \mathsf{VerKey}_c\langle T\rangle$	$\mathsf{zero}_c : () \mapsto \mathsf{Int}_c$
$\mathsf{h}_c : (T) \mapsto \mathsf{Hash}_c\langle T\rangle$	

Table 3.6 Typing destructors $g : (T_1,\dots,T_n) \mapsto T$

$\mathsf{dec}_c : (\mathsf{SymEnc}_c\langle T\rangle, \mathsf{SymKey}_c\langle T\rangle) \mapsto T$	$\mathsf{eq} : (T, T) \mapsto \mathsf{Bool}$
$\mathsf{deca}_c : (\mathsf{PubEnc}_c\langle T\rangle, \mathsf{PrivKey}_c\langle T\rangle) \mapsto T$	$\mathsf{first} : (\mathsf{Pair}\langle T, U\rangle) \mapsto T$
$\mathsf{msg}_c : (\mathsf{Signed}_c\langle T\rangle) \mapsto T$	$\mathsf{second} : (\mathsf{Pair}\langle T, U\rangle) \mapsto U$
$\mathsf{ver}_c : (\mathsf{Signed}_c\langle T\rangle, \mathsf{VerKey}_c\langle T\rangle) \mapsto T$	$\mathsf{pre}_c : (\mathsf{Int}_c) \mapsto \mathsf{Int}_c$

All types inside a constructor or destructor type should have the same configuration as the corresponding constructor or destructor, as far as it makes sense (but it should always be possible to use a different configuration with the type variables, of course). This allows us, for example, to detect the attempt to use a DES key $k : \mathsf{SymKey}_{\mathsf{DES}}\langle\mathsf{Int}_c\rangle$ to decrypt an AES encrypted message $M : \mathsf{SymEnc}_{\mathsf{AES}}\langle\mathsf{Int}_c\rangle$, i.e., in a destructor application $\mathsf{dec}_{\mathsf{AES}}(M, k)$. Other uses of the configurations are discussed in Sections 3.2 and 4.2.

The judgment $\Gamma \vdash M : T$ checks that the message M has the type T using the term typing rules defined in Table 3.7. The types of names and variables can be simply determined using the corresponding binding from the typing environment. The constructor applications have the type given by the corresponding constructor type, provided that the types of all arguments match the constructor type. Any term having type T can also be seen as having any supertype of T by the *subsumption* rule.

Table 3.7 Typing terms $\Gamma \vdash M : T$

$$\textsc{Env}\quad \frac{\Gamma \vdash \diamond \qquad u : T \in \Gamma}{\Gamma \vdash u : T} \qquad\qquad \textsc{Sub}\quad \frac{\Gamma \vdash M : T \qquad \Gamma \vdash T <: T'}{\Gamma \vdash M : T'}$$

$$\textsc{Constr}\quad \frac{f : (T_1, \dots, T_n) \mapsto T \qquad \forall i \in [1,n].\ \Gamma \vdash M_i : T_i}{\Gamma \vdash f(M_1, \dots, M_n) : T}$$

The typing judgment $\Gamma \vdash P$ checks whether the process P is well-typed using the process typing rules from Table 3.8. The output process "$\mathsf{out}(M,N).P$" is well-typed, if the term M has the channel type $\mathsf{Channel}_c\langle T\rangle$, the term N has the type T and the process P is well-typed. The input process "$\mathsf{in}(M,x).P$" is well-typed, if the term M has the channel type $\mathsf{Channel}_c\langle T\rangle$ and the process P is also well-typed assuming that x has the type T. The restriction process "$\mathsf{new}\ a : T.P$" is well-typed, if the type T is generative according to Table 3.1 and the process P is well-typed assuming that a has type T. The parallel composition "$P \mid Q$" is well-typed, if both processes P and Q independent of each other are also well-typed. The null process "0" is always well-typed. Finally, the destructor evaluation process "$\mathsf{let}\ x = g(M_1, \dots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q$" is well-typed, if the process Q is well-typed, all parameters of the destructor g have the corresponding parameter types of the destructor type and the process P is well-typed assuming that x has the destructor return type.

Table 3.8 Typing processes $\Gamma \vdash P$

$$\textsc{Proc-Out}\quad \frac{\Gamma \vdash M : \mathsf{Channel}_c\langle T\rangle \qquad \Gamma \vdash N : T \qquad \Gamma \vdash P}{\Gamma \vdash \mathsf{out}(M,N).P} \qquad \textsc{Proc-[Repl]-In}\quad \frac{\Gamma \vdash M : \mathsf{Channel}_c\langle T\rangle \qquad \Gamma, x : T \vdash P}{\Gamma \vdash [!]\mathsf{in}(M,x).P}$$

$$\textsc{Proc-Stop}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}} \qquad \textsc{Proc-New}\quad \frac{T\ is\ \mathsf{generative} \qquad \Gamma, a : T \vdash P}{\Gamma \vdash \mathsf{new}\ a : T.P} \qquad \textsc{Proc-Par}\quad \frac{\Gamma \vdash Q \qquad \Gamma \vdash P}{\Gamma \vdash P \mid Q}$$

$$\textsc{Proc-Des}\quad \frac{g : (T_1, \dots, T_n) \mapsto T \qquad \forall i \in [1,n].\ \Gamma \vdash M_i : T_i \qquad \Gamma, x : T \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash \mathsf{let}\ x = g(M_1, \dots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q}$$

3.6 Proofs

The proof technique is standard [WF94]. We are going to show that process typing is preserved by evaluation (subject-reduction).

We first show that typing judgments can be only derived for well-formed typing environments.

Lemma 3.6.1 (Well-formed Environment)

If $\Gamma \vdash \mathcal{J}$, *then* $\Gamma \vdash \diamond$.

Proof. By induction on the length of the typing derivation of $\Gamma \vdash \mathcal{J}$. All base cases are trivial, since there we explicitly check that the typing environment is well-formed. □

The weakening lemma states that if a typing environment Γ proves a judgment $\mathcal{J}$, then every well-formed extension of Γ still proves $\mathcal{J}$. Thus, extending the set of typing assumptions means extending the set of provable judgments.

Lemma 3.6.2 (Weakening)

If $\Gamma, \Gamma'' \vdash \mathcal{J}$ *and* $\Gamma, \Gamma', \Gamma'' \vdash \diamond$, *then* $\Gamma, \Gamma', \Gamma'' \vdash \mathcal{J}$.

Proof. The only typing rule explicitly checking for the absence of a term in the typing environment is Env-Binding and, by hypothesis, $\Gamma, \Gamma', \Gamma'' \vdash \diamond$. The proof is by induction on the length of the typing derivation of $\Gamma, \Gamma'' \vdash \mathcal{J}$. □

The strengthening lemma states that if a judgment $\mathcal{J}$ can be proved by a typing environment Γ, then the judgment $\mathcal{J}$ can be proved after removal of name and variable bindings not occurring in $\mathcal{J}$ from Γ.

Lemma 3.6.3 (Strengthening)

If $\Gamma, \Gamma'', \Gamma' \vdash \mathcal{J}$, $dom(\Gamma'') \cap free(\mathcal{J}) = \emptyset$ *and* $\Gamma, \Gamma' \vdash \diamond$, *then* $\Gamma, \Gamma' \vdash \mathcal{J}$.

Proof. The proof proceeds by induction on the length of the typing derivation of $\Gamma, \Gamma'', \Gamma' \vdash \mathcal{J}$, and case analysis on the last applied rule. For rule Env, we know that $\Gamma, \Gamma'', \Gamma' \vdash u : T$ and thus by the premises of the rule that $(u : T) \in \Gamma, \Gamma'', \Gamma'$. Since $dom(\Gamma'') \cap \{u\} = \emptyset$ we have that $(u : T) \in \Gamma, \Gamma'$. From the hypothesis of the lemma we have that $\Gamma, \Gamma' \vdash \diamond$, so by applying Env we conclude that $\Gamma, \Gamma' \vdash u : T$. All the other cases are trivial. □

We denote by $\Gamma \vdash \mathcal{J}\{M/x\}$ the capture-avoiding substitution of x by M inside an arbitrary typing judgment $\mathcal{J}$.

Definition 3.6.4 (Substitution)

- $(\Gamma \vdash \diamond)\{M/x\} = \Gamma \vdash \diamond$

- $(\Gamma \vdash T <: U)\{M/x\} = \Gamma \vdash T <: U$
- $(\Gamma \vdash N : T)\{M/x\} = \Gamma \vdash N\{M/x\} : T$
- $(\Gamma \vdash P)\{M/x\} = \Gamma \vdash P\{M/x\}$

Note that the substitution applies only to terms. The substitution lemma is a standard tool for proving the preservation of types at run-time. During process evaluation, variables are instantiated by terms, and the substitution lemma states that all typing judgments are preserved by the type-preserving substitution of variables.

Lemma 3.6.5 (Substitution)

If $\Gamma, x : T \vdash \mathcal{J}$ *and* $\Gamma \vdash M : T$*, then* $\Gamma \vdash \mathcal{J}\{M/x\}$.

Proof. We proceed by cases, depending on the judgment $\mathcal{J}$:

Case $\mathcal{J} = \diamond$ and $\mathcal{J} = T <: U$. Trivial by Lemma 3.6.2 (Weakening), since the substitution does not apply.

Case $\mathcal{J} = M : U$. The proof proceeds by induction on the length of the derivation of $\Gamma, x : T \vdash M : U$, and case analysis on the last applied rule.

The first base case is ENV, where $\Gamma, x : T \vdash u : U$ for some u and by the premises of the rule $\Gamma, x : T \vdash \diamond$ and $u : U \in \Gamma, x : T$. If $u = x$, then we are done since $U = T$ (by ENV-BINDING), $\Gamma \vdash M : T$ (by hypothesis), and $(u : U)\{M/u\} = M : T$. If $u \neq x$, then $u : U \in \Gamma$ (since $u : U \in \Gamma, x : T$ and $u \neq x$), $\Gamma \vdash \diamond$ (by inverting ENV-BINDING from $\Gamma, x : T \vdash \diamond$), so by ENV we have that $\Gamma \vdash u : U$, as desired. The induction steps follow directly from the induction hypothesis.

Case $\mathcal{J} = P$. The proof is simple and uses the previous judgments

□

The next lemma shows that the order of the bindings and formulas inside a well-formed typing environment does not matter.

Lemma 3.6.6 (Exchange)

If $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \vdash \mathcal{J}$ *and* $\Gamma_1, \Gamma_3, \Gamma_2, \Gamma_4 \vdash \diamond$*, then* $\Gamma_1, \Gamma_3, \Gamma_2, \Gamma_4 \vdash \mathcal{J}$.

Proof. Trivial since no judgment depends on the order of the elements of Γ. The proof is by induction on the length of the derivation of $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \vdash \mathcal{J}$. □

The next lemma shows that the subtyping relation is a partial order.

Lemma 3.6.7 (Subtyping Partial Order)

Reflexivity $\Gamma \vdash T <: T$

Antisymmetry *If* $\Gamma \vdash T <: U$ *and* $\Gamma \vdash U <: T$ *then* $T = U$

Transitivity *If* $\Gamma \vdash T_1 <: T_2$ *and* $\Gamma \vdash T_2 <: T_3$ *then* $\Gamma \vdash T_1 <: T_3$

Proof. Reflexivity follows directly from SUB-REFL.

We prove Antisymmetry by induction on the derivation of $\Gamma \vdash U <: T$ and case analysis on the last applied rule.

SUB-TOP We have $T = \mathsf{Top}$ and by assumption it also holds that $\Gamma \vdash T <: U$. Since the only case where $\Gamma \vdash \mathsf{Top} <: U$ can be true is SUB-REFL, we also have that $T = U$.

SUB-NESTED We have that there exist types T', $T_1, \dots, T_n$ and $U_1, \dots, U_n$ s.t. $T = T'\langle T_1, \dots, T_n\rangle$ and $U = T'\langle U_1, \dots, U_n\rangle$. Furthermore, we have that $\forall i \in \mathcal{V}^+ : \Gamma \vdash U_i <: T_i$, $\forall i \in \mathcal{V}^- : \Gamma \vdash T_i <: U_i$ and $\forall i \in \mathcal{V}^\circ : T_i = U_i$, where $\mathcal{V}^+ \subseteq [1, n]$, $\mathcal{V}^- \subseteq [1, n]$ and $\mathcal{V}^\circ \subseteq [1, n]$ denote the subsets of indices of covariant, contravariant and invariant parameters respectively.

In order to prove $T_i = U_i$ for all $i \in [1, n]$, we do a case analysis on the subtyping rule applied to $\Gamma \vdash T <: U$, i.e., $\Gamma \vdash T'\langle T_1, \dots, T_n\rangle <: T'\langle U_1, \dots, U_n\rangle$.

SUB-NESTED We get that $\forall i \in \mathcal{V}^+ : \Gamma \vdash T_i <: U_i$, $\forall i \in \mathcal{V}^- : \Gamma \vdash U_i <: T_i$ and $\forall i \in \mathcal{V}^\circ : T_i = U_i$. Now we can apply the induction hypothesis to conclude that $\forall i \in [1, n] : T_i = U_i$.

The other cases are trivial.

The proof for transitivity is by induction on the derivation of $\Gamma \vdash T_2 <: T_3$ and case analysis on the last applied rule.

SUB-NESTED In this case we have that there exist types T, $T_1', \dots, T_n'$ and $U_1, \dots, U_n$ s.t. $T_2 = T\langle T_1', \dots, T_n'\rangle$, $T_3 = T\langle U_1, \dots, U_n\rangle$ and $\forall i \in \mathcal{V}^+ : \Gamma \vdash T_i' <: U_i$, $\forall i \in \mathcal{V}^- : \Gamma \vdash U_i <: T_i'$ and $\forall i \in \mathcal{V}^\circ : T_i' = U_i$, where $\mathcal{V}^+ \subseteq [1, n]$, $\mathcal{V}^- \subseteq [1, n]$ and $\mathcal{V}^\circ \subseteq [1, n]$ are the subsets of indices of covariant, contravariant and invariant parameters respectively, as in the previous case. Furthermore we also get from the assumption that $\Gamma \vdash T_1 <: T\langle T_1', \dots, T_n'\rangle$

Now we do a case analysis on the rule applied to $\Gamma \vdash T_1 <: T_2$

SUB-NESTED We get that there exist types $T_1'', \dots, T_n''$ s.t. $T_1 = T\langle T_1'', \dots, T_n''\rangle$ and $\forall i \in \mathcal{V}^+ : \Gamma \vdash T_i'' <: T_i'$, $\forall i \in \mathcal{V}^- : \Gamma \vdash T_i' <: T_i''$ and $\forall i \in \mathcal{V}^\circ : T_i'' = T_i'$. We apply the induction hypothesis to T_i'', T_i' and U_i for all $i \in [1, n]$ and conclude that $\forall i \in \mathcal{V}^+ : \Gamma \vdash T_i'' <: U_i$, $\forall i \in \mathcal{V}^- : \Gamma \vdash U_i <: T_i''$ and $\forall i \in \mathcal{V}^\circ : T_i'' = U_i$ and therefore also $\Gamma \vdash T_1 <: T_3$ as desired.

The other cases, SUB-REFL and SUB-TOP are trivial. □

The next lemma states that well-typing is preserved by structural equivalence.

Lemma 3.6.8 (Structural Equivalence Preserves Typing)

If $\Gamma \vdash P$ *and* $P \equiv Q$*, then* $\Gamma \vdash Q$*.*

Proof. The proof is by induction on the length of the derivation of $P \equiv Q$ and case analysis on the last applied rule.

Case (Eq-Scope) In the scope extrusion case we assume that $\Gamma \vdash \mathsf{new}\ a : T.(P \mid Q)$ and $a \notin fn(P)$ and we show that $\Gamma \vdash P \mid \mathsf{new}\ a : T.Q$.

From the assumption, by inverting the rules Proc-New and Proc-Par, we obtain that that $\Gamma, a : T \vdash P$ and $\Gamma, a : T \vdash Q$. Since $a \notin fn(P)$ we can apply Lemma 3.6.2 (Weakening) to $\Gamma, a : T \vdash P$ and obtain $\Gamma \vdash P$. From $\Gamma, a : T \vdash Q$ by Proc-New we get $\Gamma \vdash \mathsf{new}\ a : T.Q$. By Proc-Par this implies that $\Gamma \vdash P \mid \mathsf{new}\ a : T.Q$.

Case (Eq-Ctxt) The inductive case corresponds to the closure of structural equivalence under application of evaluation contexts.

For $\mathcal{E} = \mathsf{new}\ \widetilde{a} : \widetilde{T}.([\,] \mid R)$ we have that $\Gamma \vdash \mathcal{E}[P]$, $\mathcal{E}[P] \equiv \mathcal{E}[Q]$, and $P \equiv Q$. From $\Gamma \vdash \mathcal{E}[P]$ by inverting Proc-New and Proc-Par we get that $\Gamma, \widetilde{a} : \widetilde{T} \vdash P$ and $\Gamma, \widetilde{a} : \widetilde{T} \vdash R$. From $\Gamma, \widetilde{a} : \widetilde{T} \vdash P$ and $P \equiv Q$, by the induction hypothesis we get that $\Gamma, \widetilde{a} : \widetilde{T} \vdash Q$. Now, using Proc-Par and Proc-New we can conclude that $\Gamma \vdash \mathcal{E}[Q]$.

The case $P \mid \mathbf{0} \equiv P$ (Eq-Zero-Id) is trivial

The cases $P \mid Q \equiv Q \mid P$ (Eq-Par-Comm), $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$ (Eq-Par-Assoc) and $\mathsf{new}\ a_1 : T_1.\mathsf{new}\ a_2 : T_2.P \equiv \mathsf{new}\ a_2 : T_2.\mathsf{new}\ a_1 : T_1.P$ (Eq-Bind-Swap) follow by Lemma 3.6.6 (Exchange). $\square$

The following proposition states that the reduction rules for destructors are consistent with the typing rules.

Proposition 3.6.9 (Typing Destructors Consistent)

If $g : (T_1, \ldots, T_n) \mapsto T$, $g(M_1, \ldots, M_n) \Downarrow N$ and $\Gamma \vdash M_i : T_i$ for all $i \in [1, n]$ we also have that $\Gamma \vdash N : T$.

Proof. The proof is by case analysis on the destructor g.

Case $\mathsf{deca}_c(\mathsf{enca}_c(M, \mathsf{pk}_c(K)), K) \Downarrow M$. From the definition in Table 3.6 we know that for all T the type of this destructor is $\mathsf{dec}_c : (\mathsf{PubEnc}_c\langle T\rangle, \mathsf{PrivKey}_c\langle T\rangle) \mapsto T$. Using the assumption, we get that $\Gamma \vdash \mathsf{enca}_c(M, \mathsf{pk}_c(K)) : \mathsf{PubEnc}_c\langle T\rangle$ and $\Gamma \vdash K : \mathsf{PrivKey}_c\langle T\rangle$ for some T.

Now we show that $\Gamma \vdash M : T$ by induction on the derivation of $\Gamma \vdash \mathsf{enca}_c(M, \mathsf{pk}_c(K)) : \mathsf{PubEnc}_c\langle T\rangle$ and a case analysis on the last applied rule.

Constr $\Gamma \vdash M : T$ follows directly from the definition $\mathsf{enca}_c : (T, \mathsf{PubKey}_c\langle T\rangle) \mapsto \mathsf{PubEnc}_c\langle T\rangle$ in Table 3.5.

Sub Since $\mathsf{PubEnc}_c\langle T^+\rangle$ is covariant, we have that there exists a type T' s.t. $\Gamma \vdash T' <: T$ and $\Gamma \vdash \mathsf{enca}_c(M, \mathsf{pk}_c(K)) : \mathsf{PubEnc}_c\langle T'\rangle$. Finally, by applying the induction hypothesis and Sub we can conclude that $\Gamma \vdash M : T$.

Case $\mathsf{first}(\mathsf{pair}(M, N)) \Downarrow M$. From the definition in Table 3.6 we know that for all T the type of this destructor is $\mathsf{first} : (\mathsf{Pair}\langle T, U\rangle) \mapsto T$. Using the assumption, we also get that $\Gamma \vdash \mathsf{pair}(M, N) : \mathsf{Pair}\langle T, U\rangle$ for some T.

Now we show $\Gamma \vdash M : T$ by induction on the derivation of $\mathsf{pair}(M, N)$ and a case analysis on the last applied rule.

CONSTR $\Gamma \vdash M : T$ follows from the definition $\mathsf{pair} : (T, U) \mapsto \mathsf{Pair}\langle T, U\rangle$ in Table 3.5.

SUB We have that there exists a type T' s.t. $\Gamma \vdash T' <: T$ and, since $\mathsf{Pair}\langle T^+, U^+\rangle$ is covariant, that $\Gamma \vdash \mathsf{pair}(M, N) : \mathsf{Pair}\langle T', U\rangle$. Finally, by applying the induction hypothesis and SUB we can conclude that $\Gamma \vdash M : T$.

Case $\mathsf{pre}_c(\mathsf{succ}_c(M)) \Downarrow M$. From the definition in Table 3.6 we know that for all T the type of this destructor is $\mathsf{pre}_c : (\mathsf{Int}_c) \mapsto \mathsf{Int}_c$. Using the assumption, we now also have that $\mathsf{succ}_c(M) : \mathsf{Int}_c$.

Now we show $\Gamma \vdash M : \mathsf{Int}_c$ by a case analysis on the last applied rule.

CONSTR $\Gamma \vdash M : \mathsf{Int}_c$ follows from the definition $\mathsf{succ}_c : (\mathsf{Int}_c) \mapsto \mathsf{Int}_c$ in Table 3.5.

SUB We have that there exists a type T' s.t. $\Gamma \vdash T' <: \mathsf{Int}_c$ and $\Gamma \vdash \mathsf{succ}_c(M) : T'$. The only type that can satisfy $\Gamma \vdash T' <: \mathsf{Int}_c$ is Int_c (by SUB-REFL), therefore we can immediately conclude that $\Gamma \vdash M : \mathsf{Int}_c$.

The cases dec_c and ver_c are very similar to deca_c. All of them have the same structure (up to constructor and type names) since the types of their parameters are also covariant, like $\mathsf{PubEnc}_c\langle T^+\rangle$ in the proof of deca_c.

The cases second and msg_c have the same structure as the destructor first.

The cases pre_c and eq are trivial, since here $N = \mathsf{zero}_c$ for the former and $N = \mathsf{true}$ for the latter, and $\mathsf{true} : \mathsf{Bool}$, $\mathsf{zero}_c : \mathsf{Int}_c$ by CONSTR (other rules do not apply here). □

The next theorem shows that well-typed processes stay well-typed during evaluation. This property is also known as *preservation* [WF94, Pie02].

Theorem 3.6.10 (Subject-Reduction)

If $\Gamma \vdash P$ *and* $P \rightarrow Q$ *then* $\Gamma \vdash Q$.

Proof. The proof is by induction on the derivation of $P \rightarrow Q$ and case analysis on the last applied rule.

Case (RED-I/O) We assume $\Gamma \vdash \mathsf{out}(a, M).P \mid \mathsf{in}(a, x).Q$ and $\mathsf{out}(a, M).P \mid \mathsf{in}(a, x).Q \rightarrow P \mid Q\{M/x\}$. By PROC-PAR we obtain that $\Gamma \vdash \mathsf{out}(a, M).P$ and $\Gamma \vdash \mathsf{in}(a, x).Q$. The former by PROC-OUT implies that $\exists T$ s.t. $\Gamma \vdash a : \mathsf{Channel}_c\langle T\rangle$, $\Gamma \vdash M : T$ and $\Gamma \vdash P$, while from the latter by PROC-IN we infer that $\exists T'$ s.t. $\Gamma \vdash a : \mathsf{Channel}_c\langle T'\rangle$ and $\Gamma, x : T' \vdash Q$. The type of a can only originate from one of the following rules:

ENV We get $a : \mathsf{Channel}_c\langle T\rangle \in \Gamma$ and $a : \mathsf{Channel}_c\langle T'\rangle \in \Gamma$, but $\Gamma \vdash \diamond$, so $T = T'$.

SUB We get that w.l.o.g. $\Gamma \vdash \mathsf{Channel}_c\langle T\rangle <: \mathsf{Channel}_c\langle T'\rangle$, but since channels are invariant by definition in Table 3.1, we can also infer from SUB-NESTED that $T = T'$.

Now, from Lemma 3.6.5 (Substitution) it follows that $\Gamma \vdash Q\{M/x\}$. As a consequence of the rule PROC-PAR we obtain that $\Gamma \vdash P \mid Q\{M/x\}$.

Case (RED-!I/O) This case is very similar to the previous one.

Case (Red-Destr) Let $\text{let } x = g(\widetilde{M}) \text{ in } P \text{ else } Q \rightarrow P\{N/x\}, \Gamma \vdash \text{let } x = g(\widetilde{M}) \text{ in } P \text{ else } Q$ and $g(\widetilde{M}) \Downarrow N$. By Proc-Des we get that $g : (T_1, \ldots, T_n) \mapsto T$, $\forall i \in [1, n]. \Gamma \vdash M_i : T_i$, and $\Gamma, x : T \vdash P$. By Proposition 3.6.9 (Typing Destructors Consistent) we infer that $\Gamma \vdash N : T$. By Lemma 3.6.5 (Substitution) we can finally infer that $\Gamma \vdash P\{N/x\}$.

Case (Red-Else) Immediate from Proc-Des.

Case (Red-Ctxt) Let $\mathcal{E} = \text{new } \widetilde{a} : \widetilde{T}.([\ ] \mid R)$ and assume that $\Gamma \vdash \mathcal{E}[P]$ and $P \rightarrow Q$. From $\Gamma \vdash \mathcal{E}[P]$ by reverting rule Proc-New we get that $\Gamma, \widetilde{a} : \widetilde{T} \vdash P \mid R$. From this by reverting rule Proc-Par we infer that $\Gamma, \widetilde{a} : \widetilde{T} \vdash P$ and $\Gamma, \widetilde{a} : \widetilde{T} \vdash R$. Since $\Gamma, \widetilde{a} : \widetilde{T} \vdash P$ and $P \rightarrow Q$ by the induction hypothesis we obtain that $\Gamma, \widetilde{a} : \widetilde{T} \vdash Q$. By rules Proc-Par and Proc-New we conclude that $\Gamma \vdash \mathcal{E}[Q]$.

Case (Red-Eq) Assume that $\Gamma \vdash P$ and $P \equiv P'$ and $P' \rightarrow Q'$ and $Q' \equiv Q$. From $\Gamma \vdash P$ and $P \equiv P'$ by Lemma 3.6.8 (Structural Equivalence Preserves Typing) we obtain that $\Gamma \vdash P'$. By the induction hypothesis this yields $\Gamma \vdash Q'$. By applying Lemma 3.6.8 again we conclude that $\Gamma \vdash Q$.

$\square$

The subject-reduction implies that the type system cannot be circumvented. However, we have not yet formalized the precise semantic properties that are enforced by this type system.

3.7 A Fully Annotated Example

We use the syntax "Type@Configuration<parameter types>" for parametrized types in the Expi and Exdef files. In the Expi files the types are put into the type annotations of the form "*(∗: type ∗)*". The type annotations are currently mandatory for all free name declarations "**free** a*(∗: type ∗)*." and restriction processes "**new** a*(∗: type ∗)*;".

At the moment, our type checker for expi2java cannot find out how to instantiate the type variables in constructor and destructor types, since we do not do any type inference. In order to still be able to type check the polymorphic types, we rely on the user to provide an additional type annotation for every application of a parametric constructor or destructor. These type annotations have the form *(∗ :[types]∗)*, where *types* is a comma-separated list containing the types that should be used to instantiate the constructor or destructor. For example, an application of the pair constructor on the names a : $\mathsf{Int}_{\mathsf{Nonce}}$ and b : Bool needs to be written as "pair*(∗ :[Int@Nonce, Bool]∗)*(a, b)".

This is the Perrig-Song protocol from Section 2.1.4 with all needed type annotations:

```
(*
 * Perrig-Song mutual authentication protocol
 *)

(*# include "../exdef/default.exdef" *)

(* Free names *)
free Alice(*: String *).
free Bob(*: String *).
```

```
private free M(*: String *).


(** Named processes *)

let testA (*# public *) =
    new Na(*: Int@Nonce *);
    (* Msg 1. A->B: pair(Alice, Na) *)
    out(c1, pair(*:[String, Int@Nonce]*)(Alice, Na));
    (* Msg 2. B->A: enc(pair(pair(Na, Nb), Bob), K_AB) *)
    in(c2, emsg);
    let msg1 = dec(*:[Pair<Pair<Int@Nonce, Int@Nonce>, String>]*)(emsg, k) in
        let tmp = first(*:[Pair<Int@Nonce, Int@Nonce>, String]*)(msg1) in
            let Na1 = first(*:[$Nonce, $Nonce]*)(tmp) in
                let ok(*: Bool *) = eq(*:[$Nonce]*)(Na, Na1) in
                    let Nb = second(*:[Int@Nonce, Int@Nonce]*)(tmp) in
                        (* Msg 3. A->B: enc(pair(Nb, M), K_AB) *)
                        out(c3, enc(*:[Pair<Int@Nonce, String>]*)(
                                    pair(*:[Int@Nonce, String]*)(Nb, M),
                                    k)).

let testB =
    (* Msg 1. A->B: pair(Alice, Na) *)
    in(c1, tmp);
    let name = first(*:[String, Int@Nonce]*)(tmp) in
        let Na = second(*:[String, Int@Nonce]*)(tmp) in
            new Nb(*:Int@Nonce*);
            (* Msg 2. B->A: enc(pair(pair(Na, Nb), Bob), K_AB) *)
            out(c2, enc(*:[Pair<Pair<Int@Nonce, Int@Nonce>, String>]*)(
                        pair(*:[Pair<Int@Nonce, Int@Nonce>, String]*)(
                            pair(*:[Int@Nonce, Int@Nonce]*)(Na, Nb),
                            Bob),
                        k));
            (* Msg 3. A->B: enc(pair(Nb, M), K_AB) *)
            in(c3, mmm);
            let pp = dec(*:[Pair<Int@Nonce, String>]*)(mmm, k) in
                let Nb1 = first(*:[Int@Nonce, String]*)(pp) in
                    let M1 = second(*:[Int@Nonce, String]*)(pp) in
                        let ok(*:Bool*) = eq(*:[Int@Nonce]*)(Nb1, Nb) in
                            0.

(** Main process *)
process
    new c1(*: Channel@TcpIpChCfg<Pair<String, Int@Nonce>> *);
    new c2(*: Channel@TcpIpChCfg<SymEnc@SymEncCfg<
                            Pair<Pair<Int@Nonce, Int@Nonce>, String>>> *);
    new c3(*: Channel@TcpIpChCfg<SymEnc@SymEncCfg<
                            Pair<Int@Nonce, String>>> *);
    new k(*: SymKey@SymKeyCfg<Pair<Top, Top>> *);
    (testA | testB)
```

Listing 4: Perrig-Song protocol with type annotations

We use a type “String” and several configurations in this protocol, which are defined in the default Exdef files provided with expi2java. Since the type “String” is not generative, we need to declare the term M as a free name. We use the **private** keyword to make sure that M is still not visible to the adversary in ProVerif.

The channel c had to be split into three names c1, c2 and c3, one for each message exchange. This is necessary, since the type $\mathsf{Channel}_c\langle T^\circ\rangle$ is invariant and we need to specify the exact type of each message according to the process typing rule Proc-In. An alternative would be to use tagged union types [Car04, Pie02], but they are not supported by the current type system.

As the previous example in Listing 4 shows, the type annotations for the nested types can quickly become very lengthy and hard to read. The *type definitions* can shorten these type annotations a lot.

3.8 File Format (Exdef)

Besides the Expi files described in Section 2.3, we use another file format, called *Exdef*, which stands for *External Definitions*. Exdef files are used to define types, configurations, constructors and destructors, which can be used in the Expi files via the include directive.

This is the Exdef file used with the Perrig-Song protocol:

```
(* Perrig-Song protocol configuration *)

include "../exdef/default.exdef"


type generative Nested<X+, Y-, Z0>. (* example type definition *)


type String@UTF8StringCfg.

config IdentifierCfg(class = "Identifier").

typedef $Identifier = String@IdentifierCfg.


config ChA = TcpIpChCfg_(
    variable     = "cA",
    timeout_ms   = "20000",
    port         = "2121"
).

config ChB = TcpIpChCfg_(
    variable     = "cB",
    timeout_ms   = "20000",
    port         = "2121"
).


typedef $Msg1    = Pair<$Identifier, $Nonce>.

typedef $Nonces  = Pair<$Nonce, $Nonce>.
typedef $Data    = Pair<$Nonces, $Identifier>.
typedef $Msg2    = SymEnc@SymEncCfg<$Data>.

typedef $MData   = Pair<$Nonce, String>.
typedef $Msg3    = SymEnc@SymEncCfg<$MData>.

typedef $KeyAB  = SymKey@SymKeyCfg<Pair<Top, Top>>.
```

Listing 5: Exdef file for the Perrig-Song protocol

The comments can be specified inside *(∗ ∗)*. Exdef files can also include other Exdef files using the **include** " file .exdef" directive, similar to the include construct from the Expi files.

New *types* can be defined using the "**type** [**generative**] Type[@Config][<parameters>]." construct. This defines a new type with the name "Type", optionally adds it to the set of generative types, sets the configuration "Config" as the default for this type and defines the number and variance of its parameters. The default configuration is used in cases where no configuration is explicitly provided. In order to generate code, every type used in a protocol must either have a default configuration or the configuration must be set in all type annotations using this type. The type parameters use a syntax that is similar to the abstract types in Table 3.1.

Two example type definitions can be found in the lines 6 and 9 of Listing 5. In line 6 we define a generative type named "Nested" with three parameters. The first parameter is covariant, the second is contravariant and the last one is invariant. The type "String" defined in line 9 has no parameters and uses a configuration named "UTF8StringCfg" as a default. The definitions of the default types can be found in Listing 7.

The *configurations* are defined using the "**config** Name [= BaseName](key = "value", ...)." constructs. They are handled like key-value mappings and can either be defined from scratch or by extending existing configurations. If the base configuration contains some keys with the same name as the keys in the new configuration, they are redefined by the new mapping. This allows us to change only some of the settings, e.g., the port of an TCP/IP channel, without copying all other settings. Some keys have a special meaning, this is explained in detail in Section 4.2.

Another feature are the *type definitions* (or short, *typedefs*). They are declared using "**typedef** $Name = Type" constructs. The names of typedefs always start with a $ sign to make them visually distinguishable from the types. Unlike types, the typedefs can neither be configured, nor have type parameters (but the mapped types can). The type definitions are pure syntactic sugar, they are textually replaced with the types they are mapped to on parsing.

The use of the typedefs can significantly shorten the size of the type annotations, as shown on the example of the Perrig-Song protocol (using the Exdef file from Listing 5):

```
(*
 * Perrig-Song mutual authentication protocol
 *)

(*# include "perrig-song.exdef" *)

(* Free names *)
free Alice(*: $Identifier *).
free Bob(*: $Identifier *).
private free M(*: String *).


(** Named processes *)

```

```
let testA =
    new Na(*: $Nonce *);
    (* Msg 1. A->B: pair(Alice, Na) *)
    out(c1, pair(*:[$Identifier, $Nonce]*)(Alice, Na));
    (* Msg 2. B->A: enc(pair(pair(Na, Nb), Bob), K_AB) *)
    in(c2, emsg);
    let msg1 = dec(*:[$Data]*)(emsg, k) in
        let tmp = first(*:[$Nonces, $Identifier]*)(msg1) in
            let Na1 = first(*:[$Nonce, $Nonce]*)(tmp) in
                let ok(*: Bool *) = eq(*:[$Nonce]*)(Na, Na1) in
                    let Nb = second(*:[$Nonce, $Nonce]*)(tmp) in
                        (* Msg 3. A->B: enc(pair(Nb, M), K_AB) *)
                        out(c3, enc(*:[$MData]*)(
                                    pair(*:[$Nonce, String]*)(Nb, M), k)).

let testB =
    let c1(*: Channel@ChB<$Msg1> *) = accept(*:[$Msg1]*)(c1) in
    let c2(*: Channel@ChB<$Msg2> *) = accept(*:[$Msg2]*)(c2) in
    let c3(*: Channel@ChB<$Msg3> *) = accept(*:[$Msg3]*)(c3) in
    (* Msg 1. A->B: pair(Alice, Na) *)
    in(c1, tmp);
    let name = first(*:[$Identifier, $Nonce]*)(tmp) in
        let Na = second(*:[$Identifier, $Nonce]*)(tmp) in
            new Nb(*: $Nonce *);
            (* Msg 2. B->A: enc(pair(pair(Na, Nb), Bob), K_AB) *)
            out(c2, enc(*:[$Data]*)(pair(*:[$Nonces, $Identifier]*)(
                                        pair(*:[$Nonce, $Nonce]*)(Na, Nb),
                                        Bob), k));
            (* Msg 3. A->B: enc(pair(Nb, M), K_AB) *)
            in(c3, mmm);
            let pp = dec(*:[$MData]*)(mmm, k) in
                let Nb1 = first(*:[$Nonce, String]*)(pp) in
                    let M1 = second(*:[$Nonce, String]*)(pp) in
                        let ok(*: Bool *) = eq(*:[$Nonce]*)(Nb1, Nb) in
                            0.

(* Main process *)
process
    new c1(*: Channel@ChA<$Msg1> *);
    new c2(*: Channel@ChA<$Msg2> *);
    new c3(*: Channel@ChA<$Msg3> *);
    new k(*: $KeyAB *);
    (testA | testB)
```

Listing 6: Perrig-Song protocol annotated using the typedefs

In comparison to the previous examples, we create local copies of the channels c1, c2 and c3 in process testB. We use the destructor accept defined in the line 49 of Listing 7 to model the server side of the protocol. This destructor is modelled in the Extensible Spi Calculus as an identity function to ensure that the semantics of the input and output processes defined by the rules RED-I/O and RED-!I/O are preserved. The implementation of accept listens on the port specified in the configuration of the given channel and returns an initialized channel when a connection is made.

The constructors and destructors are also defined inside the Exdef files. Since the set of constructors from Table 2.1 and destructors from Table 2.2 covers the most common cryptographic primitives, we define them together with the types from Table 3.1 in an Exdef file named “default.exdef”, shown in Listing 7 below. This file is used by all example protocols. Another file named “configs.exdef” contains a long list of all supported configurations.

```
(* default external definitions *)

include "configs.exdef"

(* cryptographic primitives *)
type            SymEnc<T+>.
type generative SymKey<T->.
type            PubEnc<T+>.
type            PubKey<T->.
type generative PrivKey<T->.
type            Signed<T+>.
type generative SigKey<T->.
type            VerKey<T->.
type            Hash<T+>.

(* data types *)
type            Pair@PairCfg<X+, Y+>.
type            Bool@BoolCfg.
type generative Int.

(* default typedefs *)
typedef $Nonce = Int@NonceCfg.

(* constructors *)
fun enc    : [T]. (T, SymKey<T>) -> SymEnc<T>.
fun enca   : [T]. (T, PubKey<T>) -> PubEnc<T>.
fun pk     : [T]. (PrivKey<T>) -> PubKey<T>.
fun sign   : [T]. (T, SigKey<T>) -> Signed<T>.
fun vk     : [T]. (SigKey<T>) -> VerKey<T>.
fun h      : [T]. (T) -> Hash<T>.
fun true   : Bool.
fun false  : Bool.
fun pair   : [T, U]. (T, U) -> Pair<T, U>.
fun succ   : (Int) -> Int.
fun zero   : Int.

(* destructors *)
reduc dec(enc(*:[T]*)(x, y), y) = x      : [T]. (SymEnc<T>, SymKey<T>) -> T.
reduc deca(enca(*:[T]*)(x, pk(*:[T]*)(y)), y) = x
                                         : [T]. (PubEnc<T>, PrivKey<T>) -> T.
reduc ver(sign(*:[T]*)(x, k), vk(*:[T]*)(k)) = x
                                         : [T]. (Signed<T>, VerKey<T>) -> T.
reduc msg(sign(*:[T]*)(x, y)) = x        : [T]. (Signed<T>) -> T.
reduc eq(x, x) = true                    : [T]. (T, T) -> Bool.
reduc first(pair(*:[T, U]*)(x, y)) = x   : [T, U]. (Pair<T, U>) -> T.
reduc second(pair(*:[T, U]*)(x, y)) = y  : [T, U]. (Pair<T, U>) -> U.
reduc pre(succ(x)) = x;
      pre(zero) = zero()                 : (Int) -> Int.
reduc accept(c) = c                      : [T]. (Channel<T>) -> Channel<T>.
```

Listing 7: The default Exdef file

The constructors are declared using the “**fun** name : type.” constructs. The destructor declarations have a similar syntax to the syntax used in ProVerif, but additionally specify the destructor type: “**reduc** name(terms) = result : type.”. The type of both constructors and destructors has the form "[parameters]. (argument types) −> result type". The *parameters* is a list of the type parameters, allowed in the argument types and return type. If the list of parameters or argument types is empty, it can be omitted, e.g., the constructor declaration "**fun** true : Bool." is equivalent to "**fun** true : []. () −> Bool.".

The *result* term in the destructor declaration can only contain (arbitrary) constructors and names or variables that are also present in its parameters *terms*. The configurations of the destructor parameters should match the configurations of the corresponding parameters of the destructor type. Note that a destructor can have multiple reduction rules (like pre_c), in this case, all of them must have the same number of parameters and the same type.

4 Code Generation

The most important feature of expi2java is code generation. In Section 4.1 we describe the overall design of the tool, discuss the features of the code generator and some challenges that arose during the implementation. Section 4.2 explains the aspects of configurations that are important for the code generator. Finally, in Section 4.3 we summarize the purpose of the runtime library used by the generated code.

4.1 Design

4.1.1 Overview

The overall design of expi2java framework is shown in Figure 4.1. There are two arts of input files. The Expi file contains the protocol and the Exdef files contain the low-level information for the code generation. The input files are parsed, checked for consistency and transformed into an internal in-memory representation. This data is then used as the input to the type checker. Afterwards, the data is passed to the code generator that produces the code using user-provided code templates. The generated code uses a *runtime library* that is described in Section 4.3. Additionally, a *pretty-printer* can be used to output a formatted version of the input files in ProVerif syntax.

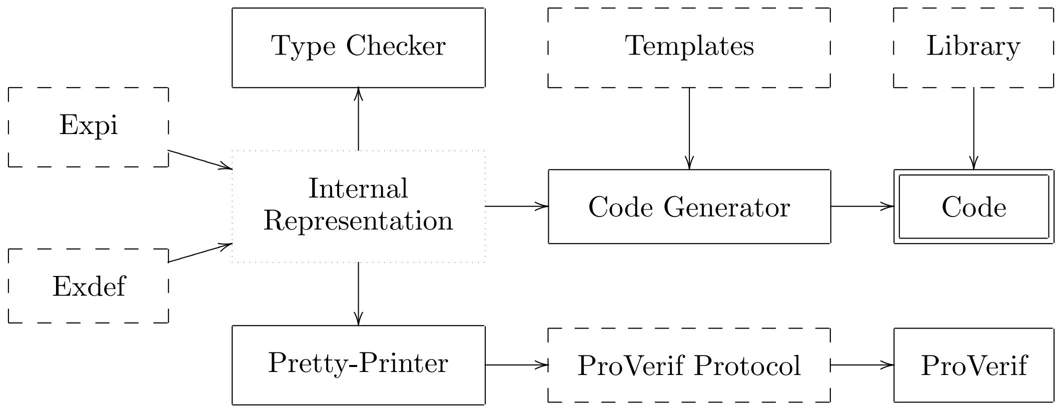

Figure 4.1: Overall design of expi2java

We will not go into the implementation details of expi2java here, but rather give an overview of the functionality.

4.1.2 Translation of the Extensible Spi Calculus to Java

The target language of the code generator and the implementation language of the tool is Java. We require at least version 1.6 to build the tool, the runtime library and the generated code[1].

The implementation of the Extensible Spi Calculus defined in Section 2.1 is split into two parts. All user-defined constructs (i.e., types, constructors and destructors) can vary between different protocols and are implemented inside the runtime library that is used only by the generated code and can be changed as needed. Everything else is implemented directly in the generated code.

For example, the following lines from the testA process in the Perrig-Song protocol:

```
    new Na(*: $Nonce *);
    (* Msg 1. A->B: pair(Alice, Na) *)
    out(c1, pair(*:[$Identifier, $Nonce]*)(Alice, Na));
    (* Msg 2. B->A: enc(pair(pair(Na, Nb), Bob), K_AB) *)
    in(c2, emsg);
    ...
```

result in the following Java code:

```
// new Na(*: Int@NonceCfg *); out...
Nonce Na_44 = null;
Na_44 = new Nonce(this._configurations.get("NonceCfg"), "Na_44");
try {
    // out(c1, pair(*:[String@IdentifierCfg, Int@NonceCfg]*)(Alice, Na)); in...
    Pair<Identifier, Nonce> pair_48__Alice_41_Na_44__9
            = Pair.pairCtor(this._configurations.get("PairCfg"), Alice_41, Na_44);
    c1_116.send(pair_48__Alice_41_Na_44__9);

    // in(c2, emsg); let...
    // "emsg" id=50
    SymEnc<Pair<Pair<Nonce, Nonce>, Identifier>> emsg_50 = null;
    try {
        emsg_50 = c2_117.receive(new SymEnc<Pair<Pair<Nonce, Nonce>, Identifier>>(
                    new Pair<Pair<Nonce, Nonce>, Identifier>(
                        new Pair<Nonce, Nonce>(
                            new Nonce(this._configurations.get("NonceCfg")),
                            new Nonce(this._configurations.get("NonceCfg")),
                            this._configurations.get("PairCfg")),
                        new Identifier(this._configurations.get("IdentifierCfg")),
                        this._configurations.get("PairCfg")),
                    this._configurations.get("SymEncCfg")));
        emsg_50.setVarName("emsg_50");


        ...
    } finally {
        // cleanup
        if (emsg_50 != null) { emsg_50.deinitialize(); }
    }
} finally {
    // cleanup
    if (Na_44 != null) { Na_44.deinitialize(); }
}
```

Listing 8: Generated Java code

[1]In theory, version 1.5 should suffice too, since it also supports generics, but we have not tested it.

By convention, all free names are treated as input parameters of the protocol, i.e., they must be initialized by modifying generated code manually and deinitialized afterwards. This feature should be used when the protocol depends on some dynamic values provided by the application that uses the protocol. The names, generated inside the protocol using the "$\mathsf{new}\ a : T.P$" process are treated as local protocol variables, i.e., they are created and initialized as soon as the control flow reaches the corresponding restriction process and are deinitialized after the process P finishes. The initial value of the names generated in a restriction process can be set in the configuration.

The Spi names and variables correspond to Java variables. In Listing 8, the Spi name Na corresponds to the Java variable Na_44, where 44 is an unique identifier of this term. The variable Na_44 is initialized with a fresh nonce in line 3 and is deinitialized in the **finally** section in line 32. The deinitialize method must be implemented by all implementation classes and should perform the needed finalization steps like closing open files and sockets.

The constructors and destructors are implemented in the runtime library as static methods with a special signature. The constructors must be implemented in the class that represents their return type. The destructors are commonly used to perform some computation on terms of a specific type, thus we require them to be implemented in the class that represents the type of their first parameter. This ensures that for example, enc_c and dec_c are both implemented in the class that represents the symmetric encryption.

An example of a constructor application can be found in lines 5-7 of Listing 8. The configuration that should be used to initialize the result is given as a parameter to the pairCtor method that implements the pair constructor.

The input process corresponds to a receive method and the output process corresponds to a send method of the channel type in the runtime library, as can be seen in lines 14 and 8 of Listing 8 respectively. The destructor evaluation assigns the result of the destructor application to a Java variable and proceeds with the correct process. The parallel composition creates two threads, one for each process, runs them and waits until both threads terminate. The replicated input "$!\,\mathsf{in}(M, x).P$" is implemented as an endless loop that first inputs some data using the input process, initializes the variable x, then creates a new thread for the process P and runs it without waiting for its termination.

In the implementation, unlike in the abstract calculus, where only the destructor evaluation can fail, any process (except the null process) or constructor application can fail at any point, e.g., because of an "out of memory" error. In this case, we deinitialize all local names and variables and abort the protocol with an exception. The parallel composition fails if one of the processes running in parallel fails. The replicated input only fails if the input process fails, disregarding how the subsequent process P running in a thread terminates.

4.1.3 Current Limitations

There are also some cases that are possible in the Extensible Spi Calculus as defined in Section 2.1, but can not be implemented due to the current limitations of the type system or implementation of the runtime library.

The operational semantics of the Extensible Spi Calculus from Table 2.5 defines a *synchronous* behavior of the input and output processes. This means that an output process $\mathsf{out}(M, N).P$ starts the process P only after the message N was *received* by another input process. Another alternative is the *asynchronous* approach as in [Bou92], where output process do not have a continuation process and multiple messages are sent using the parallel composition. However, the current implementation uses a different approach, commonly used in the real-life protocols. Our implementation classes simply start the next process P right after the message was *sent*, since it is difficult to decide whether the message was received without an explicit confirmation message. In this setting, the messages are sent asynchronously, but the order in which they are sent is preserved as in synchronous case. This might cause the protocols that rely on the synchronous message delivery to fail, but should in general not make them less secure.

4.1.4 Templates

The code generator uses a set of *templates* provided with expi2java to generate the code. In general, the templates are pieces of Java code with placeholders that are replaced by the code generator with the corresponding data of the currently generating protocol. This allows the user to adapt the generated code to the needs of the target application by extending the provided templates. There are two kinds of templates, *class templates* and *code templates*.

The class templates are used as the base for the classes that represent the protocol. We generate one class for the main process, this class is named according to the protocol name and is used to start the protocol from the *main application*. The main application is a template with an example implementation of the protocol usage. The user can adapt it to fit into the actual application that he or she is developing or provide a custom template to generate a class that can be directly integrated with the existing code. We also generate a class for every thread in the protocol and a helper class containing the configuration parameters.

The code templates are small pieces of Java code (only a couple of lines) that are used to generate code for different process and term types. The use of code templates allows the user to change the produced code (within certain limits) without the need to change the implementation of the code generator. This can be used, e.g., to use another implementation of the runtime library, than the default one discussed in Section 4.3.

4.2 Configurations

During the code generation phase, we need to provide a lot of low-level information about every term. Mandatory is to know which Java class is used to implement each term. Besides the class name, there are also various settings, like the employed algorithm, padding or key length, that are used by the implementation and need to be provided by the user. In order to minimize the amount of data that needs to be entered by the user, these low-level information should be kept centralized, e.g., there should be a way to define the AES configuration and use it with the constructors and destructors, types of the keys, encrypted messages etc.

Another issue is the need to split a channel name in order to type it, as we did in the Perrig-Song example in Section 3.7. These split channels need to be *merged* together to “logical” channels in the generated code, but often in a different way as they would be represented in the untyped Extensible Spi Calculus. The implementation of a TCP/IP channel, for instance, needs to share a reference to a server socket among all Java variables that represent the channel names used in the participant process that acts as a server, and another reference of the client socket among the Java variables that represent all other parts of the same channel.

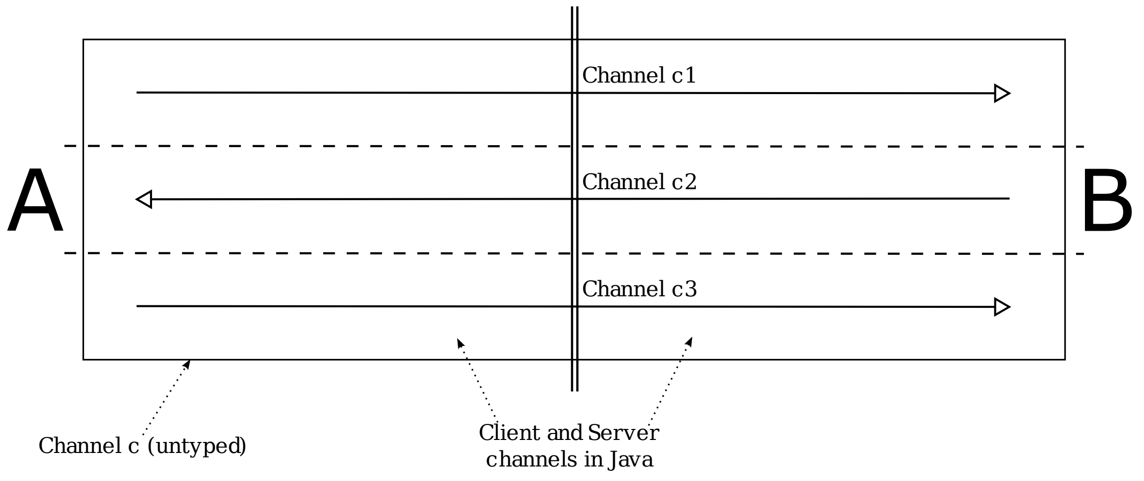

Figure 4.2: Channel splitting in Perrig-Song protocol

As shown in Figure 4.2, the Perrig-Song protocol uses one channel name c in the untyped version, three channel names c1, c2, c3 in the typed version and two Java channels in the generated code.

We have implemented channel merging using special implementation classes, called *aliases*. An alias controlls the number of instances of the aliased implementation class, similar to the singleton pattern. The user can define virtual variables in the generated code using the configuration key “variable”. The Aliases use this configuration key to create one instance of the aliased class for every virtual variable. Every instance of an alias in the generated code that belongs to the same virtual variable, forwards all method calls to the corresponding instance of the aliased class, effectively acting as one Java variable.

In most cases, different configurations in the Exdef files correspond to different configurations in the type system. However, there is also a possibility to bundle several configurations into one “virtual” configuration for the type system. For example, we need different low-level settings for the types $\mathsf{PubKey}_{\mathsf{AES}}\langle T\rangle$, $\mathsf{PrivKey}_{\mathsf{AES}}\langle T\rangle$ and $\mathsf{PubEnc}_{\mathsf{AES}}\langle T\rangle$, but they all should belong to the same configuration called “AES” in the type system. We have a special key “bundle” that allows for bundling of several different configurations into one virtual configuration.

```
(* Simple configuration *)
config IdentifierCfg(
    class              = "Identifier"
).
```

```

(* AES Bundle *)
config abstract AES_(
    bundle              = "AES Encryption",
    algorithm           = "AES",
    keylength           = "256",
    provider            = "SunJCE"
).
config AESEncCfg = AES_(
    class               = "AESEnc",
    mode                = "CTR",
    padding             = "PKCS5Padding"
).
config AESKeyCfg = AES_(
    class               = "AESKey",
    iv                  = "1234567887654321"
).
```

Listing 9: AES bundle

The AES bundle is shown in Listing 9. The bundle key is set in the base configuration AES_ and is inherited by AESEncCfg and AESKeyCfg. If the bundle key is not set, as in the simple configuration above, the name of the configuration is used instead.

Every configuration must provide the name of the implementation class in the “class” key. The settings for this class are implementation-specific and is treated as all other keys. The implementation classes are given these settings as a mapping and are free to use them however they like.

4.3 Runtime Library

The runtime library is a set of classes used to implement the behavior of the types, constructors and destructors in the generated code. Obviously, these classes cannot be generated automatically, since they need to provide low-level implementation that depends on the intended use of the protocol, e.g., the actual cryptographic algorithm corresponding to a constructor and destructor. The user is free to change existing types, constructors and destructors or add new ones and must be able to also provide the implementation in these cases without the need to change the code generator.

The library provides several interfaces that define abstract methods used by the generated code. All implementation classes need to implement these interfaces.

We provide the implementation of the default types and primitives defined in Tables 2.1, 2.2 and 3.1, as well as several helper classes (like implementation of aliases discussed in Section 4.2) in the standard expi2java distribution.

The types are implemented as generic Java classes. Every type parameter corresponds to a generic parameter in the Java realization. This allows us to model the nested types in a convenient way, avoiding many casts and instance-of checks. The implementation classes for generative types must contain a special Java constructor that is used to create new names in the implementation of the restriction process. The classes implementing channels must also support the send() and receive() methods that are used in the realization of the input and output processes.

All implementation classes (except channels) must support special serialization and deserialization methods that allow full control over the binary format as needed by many protocols. Of course, the default Java serialization can be used internally if no other format is required.

Parametrized constructors and destructors are implemented using generic methods. This allows us to use a simpler implementation suitable for all instantiations of types and integrates nicely with the generic implementation of nested types. However, we had to overcome the limitation of the Java type system with respect to generics. The generic type parameters in Java are always invariant [IPW01], but in our type system, the variance is user-defined. For example, the first argument of the destructor $\mathsf{dec}_c : (\mathsf{SymEnc}_c\langle T\rangle, \mathsf{SymKey}_c\langle T\rangle) \mapsto T$ has a covariant type $\mathsf{SymEnc}_c\langle T^+\rangle$ and the second argument has the contravariant type $\mathsf{SymKey}_c\langle T^-\rangle$. This makes it possible to use this destructor with arguments of types $\mathsf{SymEnc}_c\langle \mathsf{Int}_c\rangle$ and $\mathsf{SymKey}_c\langle \mathsf{Top}\rangle$ and return a result of type Int_c, since $\mathsf{SymKey}_c\langle \mathsf{Top}\rangle <: \mathsf{SymKey}_c\langle \mathsf{Int}_c\rangle$ by SUB-NESTED and therefore the second argument can be given the type $\mathsf{SymKey}_c\langle \mathsf{Int}_c\rangle$ by SUB.

A realization of this destructor in Java with one generic parameter would give a typing error in such cases. In the following example, generic parameter T corresponds to the type variable T:

```
public static <T extends IExpi>
T decDtor(SymEnc<T> enc, SymKey<T> key) {
    ...
}
```

Listing 10: Implementation of the dec_c destructor with one generic parameter

We worked around this problem by using several generic parameters, one for each occurrence of the type variable in the constructor or destructor type, and specifying the possible relation between them manually, using the Java type bounds. For instance, the dec_c destructor is implemented using the following signature:

```
public static <U extends IExpi, V extends U, T extends V>
V decDtor(SymEnc<T> enc, SymKey<U> key) {
    ...
}
```

Listing 11: Implementation of the dec_c destructor

The relation between the generic parameters U, V and T reflects the fact that in our type system, the parameter T of the encryption can be a subtype of the return type V and the parameter U of the key can be a supertype of the return type V.

It is sometimes desireable for a protocol to return results after finishing executing. In order to do this, we have implemented a “ReturnChannel” that stores copies of all terms that are sent over it. The stored terms can be retrieved by name using the provided methods. Such channel needs to be specified as a free name, otherwise it will be deinitialized after the protocol ends.

In principle, the complete runtime library can be replaced by another implementation. The configurations for the existing types need to be updated in this case, of course. If the naming conventions or the signatures of the library methods used to implement the Extensible Spi Calculus are changed, the templates and code snippets also need to be updated accordingly.

5 Case Study: TLS Protocol

Our goal was to generate an implementation of a simple web client that uses a popular cryptographic protocol to communicate with a normal web server. We used HTTP over TLS [Res00] to request a web page and display the received data. The intention was to implement the complete protocol in the Extensible Spi Calculus using the runtime library only for cryptographic primitives and data types. This case study was an important and challenging test for all the features of expi2java and demonstrates the potential of this tool.

We give a short overview of the TLS protocol and reason about our choices for this case study in Section 5.1. Section 5.2 describes the implemented part of the protocol. The model of TLS in the Extensible Spi Calculus and current limitations are described in Section 5.3. In Section 5.4 we discuss the results of this case study and compare our implementation with others.

5.1 Transport Layer Security

Currently, most real-world protocols are defined using informal technical specifications. Most standards for the networking protocols used in internet are developed by the Internet Engineering Task Force[1] (IETF) and are available in form of Requests for Comments[2] (RFCs). These specifications are long documents written in English and may contain errors and ambiguities, even though they normally use dictionaries that define the meaning of the important keywords and phrases like “must”, “shall”, “should”, “may”, “required” or “optional” [Bra97]. These protocols use concrete cryptographic primitives like the SHA1 hash function or RSA encryption, are focused on existing networking protocols and standards and are defined from the exact data format on the bit-string level up to general recommendations for the developers that want to implement the protocol.

The Transport Layer Security (TLS) protocol [DA99] was also designed by IETF and published as RFC 2246. It is a cryptographic protocol used to provide authentication, communication privacy and data integrity between two communicating parties over untrusted channels. The TLS protocol provides a secure transport layer for many popular application protocols such as HTTP, FTP, SMTP, NNTP, and XMPP using an underlying transport protocol, usually TCP. The Transport Layer Security protocol is the successor of the Secure Sockets Layer (SSL) protocol that was found flawed in various ways [WS96].

The TLS specification defines several sub-protocols. During the *TLS Handshake*, the participants can agree on the security parameters to be used in the later steps, authenticate

[1]Official IETF site: `http://www.ietf.org/`

[2]The official source for RFCs: `http://www.rfc-editor.org/`

themselves, generate and exchange the session keys and finally instantiate the secure connection. Once the handshake is complete, the *TLS Application Data Protocol* can be used as a transport layer for any application protocol. In addition to encryption, the integrity of the application data is verified using a MAC. The *TLS Record Layer* applies encryption and MAC transparently, which ensures interoperability with the existing applications. Besides, TLS allows the participants to notify each other about various error conditions using the *Alert Protocol*, which supports a large number of encryption schemes and allows to resume previously established connections.

5.2 Implemented Part of HTTP/TLS

TLS is a very complex protocol, it supports many encryption schemes and optional parameters. Moreover, there are several optional extensions that can be used by both client and server. However, for the purpose of HTTP/TLS, which is probably the most widespread use of this protocol, these extensions are not crucial.

There are three versions of the TLS protocol and several extensions. We have implemented the TLS 1.0 protocol with AES extension [Cho02]. The choice of the version 1.0 from the year 1999 over the more recent version 1.1 published in 2006 is motivated by its popularity.

We have implemented the required parts of the TLS Handshake protocol and the Application Data protocol. The Alert Protocol and the abbreviated handshake used to resume a previous connection are not supported. We only support one, but quite common encryption scheme, AES-256 CBC with RSA key exchange, and SHA1 HMAC. The encryption and HMAC algorithms can be changed by adapting the configurations, but require several changes, since some values that we handle as constants depend on the block size and key length. Another key exchange algorithm, such as Diffie-Hellman key exchange, would also require changes in the protocol.

The TLS handshake protocol basically consists of nine messages grouped into two message exchanges, which are shown in Figure 5.1.

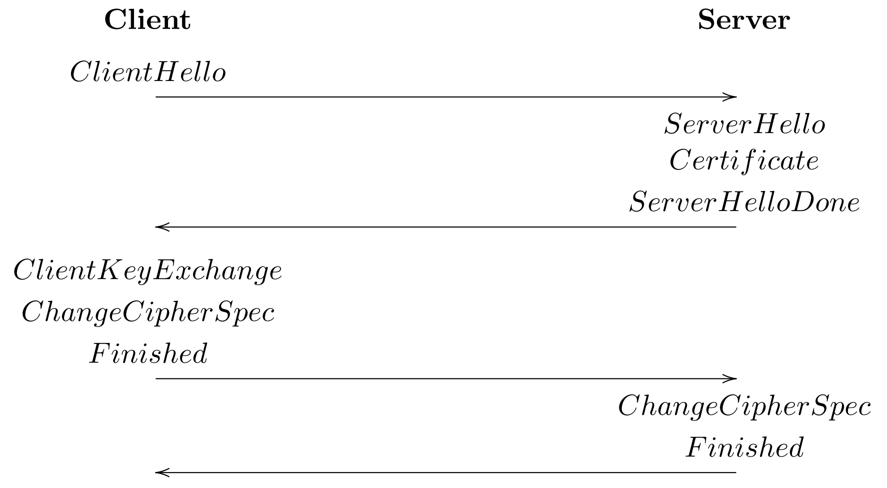

Figure 5.1: Message flow in the TLS handshake

First, the client sends the *ClientHello* message with the current timestamp, a fresh client nonce, a list of supported encryption modes and a request for a new session. The server responds with the current timestamp, a fresh server nonce, new session ID and the selected encryption mode from the received list of modes supported by the client. Then the server sends its certificate chain and finishes the hello exchange with an empty *ServerHelloDone* message.

The client checks the validity of the server's certificate, encrypts a fresh nonce with the public key of the server and sends the encryption in the *ClientKeyExchange* message. Then the client sends the *ChangeCipherSpec*, indicating that all following messages will be encrypted, and generates the "master secret", shared keys, initialization vectors and HMAC keys using a custom PRF and the nonces. This PRF has two input parameters, a "seed" and a "secret", and generates arbitrary amount of pseudo-random data using the MD5 and SHA1-based HMACs. Afterwards, the client hashes all previous messages with MD5 and SHA1 and uses them together with the master secret as the arguments for the PRF to generate a pseudo-random data of fixed size. This data is MAC-ed, encrypted and sent to the server in the *Finished* message.

The server decrypts the last nonce, generates the master secret and the keys in the same way as the client, verifies the received data and confirms the switch to encrypted mode by the *ChangeCipherSpec* message. Then the server calculates the server finished message in a similar way as the client, but using also the new messages and sends it to the client. The client finishes the handshake by verifying the last *Finished* message.

After a successful handshake the client can proceed with the Application Data protocol that encapsulates the application level protocol, in our case, HTTP.

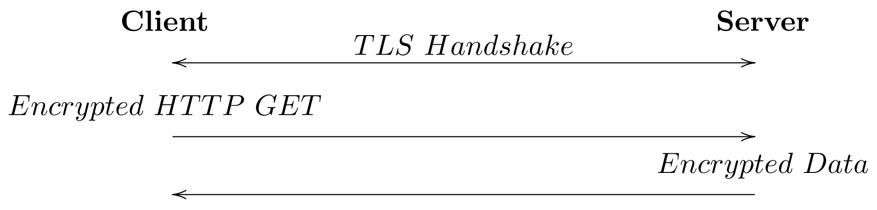

Figure 5.2: Message flow in HTTP/TLS

Every message is MAC-ed and encrypted with the previously generated keys. We use a simple HTTP GET request to retrieve the root HTML document of the specified web site. The server responds with the HTML headers concatenated with the HTML code of the requested page. The URL of the web page is specified in the configuration of the channel and can be easily changed.

It would be possible to implement the rest of the TLS protocol. For example, we could add support for multiple encryption modes and branch on the mode selected by the server using the `eq` destructor, or add a custom constructor that changes the configuration of the keys according to the encryption mode ID. The message sizes would need to be adapted accordingly in each case.

However, the complete TLS support was not the intent of this case study. The implementation helped us to refine the implementation of expi2java and the runtime library. We could experiment with different ways to express the complex behavior and data structures in the Extensible Spi Calculus and our type system. During the implementation we added several new data types and explored the possibilities of expressing user interaction.

5.3 The Model in Extensible Spi Calculus

The model of the TLS protocol performs all steps of the TLS handshake from the view of the client. We check the validity of the message MACs and the correctness of the received server finished message in the handshake and abort on failure. The received server certificate chain is also checked by the implementation class. A current limitation is that this implementation class does not check the validity of the root certificate in the chain, since it would require storing a local list of some certificates of several well-known authorities. The MAC of the received HTML data is not checked, because we currently cannot find out the size of a received message that is included into the MAC.

The structure of all messages is modelled completely in Extensible Spi Calculus, with the exception of the data format of encryptions and certificates that is hardcoded in the serialization methods of the corresponding implementation classes. In addition to the data mentioned in Section 5.2, every message also contains a header of fixed content, size of the message and some other less important fields. Since the size and the content of the handshake messages is fixed, we use many integer constants that are initialized to the correct values set in the configuration.

```
(** Protocol start: ClientHello, ServerHello **)
let client =
    new chOut(*: $ChOut *);
    (* Msg 1 : ClientHello = ((hshake_hdr, ch_size), ((ch_header,
                                    (version, c_time)), (client_random,
                                    (ch_session, ch_mode)))) *)
    new ch_size(*: $CHSize *);
    new ch_header(*: $CHHdr *);
    new c_time(*: $Timestamp *);
    new client_random(*: $TLSRandom *);
    new ch_session(*: $CHSession *);
    new ch_mode(*: $CHModes *);
    let hshake_msgs = id(*:[$HandshakeMsg1]*)(
                        pair(*:[$CHHead1up, Pair<$TLSRandom, $CHFoot>]*)(
                          pair(*:[$CHHdr, $CHHead1]*)(
                              ch_header,
                              pair(*:[$TLSVer,$Timestamp]*)(version,c_time)),
                          pair(*:[$TLSRandom, $CHFoot]*)(
                              client_random,
                              pair(*:[$CHSession,$CHModes]*)(
                                  ch_session,
                                  ch_mode)))) in
        out(chOut, pair(*:[$CHHead00, $HandshakeMsg1]*)(
                    pair(*:[$TLSHshakeHdr,$CHSize]*)(hshake_hdr, ch_size),
                    hshake_msgs));
        ...
```

Listing 12: The ClientHello message from the TLS protocol

The first message of the TLS protocol, the ClientHello message is shown in Listing 12. We create the output channel, initialize a set of constants used in this message, then compose them together and send over the channel. The name hshake_msgs contains the ClientHello message without transport layer headers.

The default library classes supported generation of integer names with specific size and value and nonces of arbitrary length in the restriction process. We only needed to extend the integer implementation to support unusual data formats used in TLS, for example it often uses 24-bit unsigned integers and not only the common 16, 32 and 64 bit versions.

The amount of different data in messages is often very large, which causes very deep nested types. The typedefs described in Section 3.8 were very helpful to keep the readability in a manageable range. However, the amount of type annotations is still very large, the configuration file for the TLS protocol is about 400 lines long and contains about 100 typedefs, which shows that type inference is crucial if we want to specify large protocols.

TLS uses common hashing and encryption algorithms, padding methods and block cipher modes of operation. These settings could be easily set in the configuration of the corresponding cryptographic primitives. The data is sent over the TCP/IP channel, configured to use the port 443, the default port for HTTPS.

We had to add several new cryptographic primitives. First, we added a constructor hmac_c for the keyed hash functions that are used throughout the protocol. Furthermore, we added a special constructor to change the initialization vector needed by the encryption in CBC mode. A small problem with initialization vectors is that they need to be set to the last encrypted block on every encryption. We hardcoded this behavior into the implementation class. A better alternative would be to use a special encryption constructor that would return not only the encryption, but also the new initialization vector.

A bigger challenge was the custom PRF function used in TLS. The problem is that in TLS, it is used with completely incompatible parameter types, since they are simply interpreted as a bitstring. The return type is also difficult to specify, since the returned byte sequence is interpreted as nonces, symmetric keys, initialization vectors or just integers as needed. We could model this PRF as a constructor prf of type [X]. (Top, Top) −> X., but in the current setting this constructor would need to be implemented in every implementation class in the runtime library. The simplest way to implement this function was to define a special channel implementation, and use it in the protocol as a black box. This channel buffers all messages sent over it, and on receive, generates a long enough pseudo-random sequence and initializes the received messages with it.

We have added a “ConsoleChannel” class that can be used for user interaction. All messages sent over it are printed to STDOUT, a receive on this channel reads a line from STDIN and initializes the message accordingly. A generative type ConstString was added, which can be used to generate text messages and display them to the user. We used these types to define simple error handling mechanism in the TLS protocol. If some decryption or verification step fails, we display a corresponding message and abort the protocol. Some status messages on the handshake progress and the received HTML data are also displayed in this way.

5.4 Results

During the development phase, the generated TLS implementation was tested on the local Apache 2.2.9 web server installation using the OpenSSL 0.9.8h library on Gentoo Linux. For debugging purposes, we used a custom build of the OpenSSL library that outputted a lot of internal information like generated keys and more informative error messages.

The final version of the generated code was tested with various web sites, among others, https://banking.postbank.de/, https://bugs.gentoo.org/, https://www.nsa.gov/, https://www.bnd.de/ and https://www.openssl.org/. The site of BND used an invalid certificate. All other tested web servers that supported the used encryption scheme (e.g., the NSA site does not support AES) accepted the handshake and responded to the encrypted HTTP GET request.

The protocol specification in Extensible Spi Calculus and the corresponding Exdef file that contains TLS-specific definitions are about 400 physical lines long each, resulting in 800 lines of specification. The size of the TLS-specific implementations in the library is about 300 physical lines, or 150 source lines of code[3] (SLOC). The generated implementation is about 7000 physical lines long (about 5700 SLOC). The protocol was implemented by the author of this thesis in about four days, including reading the protocol specification, extending the runtime library, fixing bugs in the generator and debugging the protocol and the implementation of the custom classes. The implementation time could be shortened at least by one day if we had used the custom build of the OpenSSL library right from the beginning.

For a very rough comparison, the size of the TLS-related files in OpenSSL is about 4800 physical lines of C code (3200 SLOC). Of course, this comparison is rather imprecise, since the OpenSSL implementation is written in a completely different language and supports almost all features of TLS, but gives a rough idea about the orders of magnitude of the expected code size.

Another TLS implementation is PureTLS[4]. It is written in Java using external cryptographic providers, and do not support any TLS extensions, which is very similar to our implementation. The size of the TLS-related classes is about 7800 physical lines of code (4000 SLOC), although this also contains the networking routines that are implemented in the runtime library in our case.

The predecessor of our tool, spi2java, was used to generate an interoperable implementation of an SSH client [PS07]. The SSH protocol is much simpler than TLS. Besides, the protocol specification was kept very abstract and most of the logic was implemented in the runtime library. The generated implementation was about 2000 physical lines long and supported only the handshake. Unfortunately, we do not have any information about the size of the configuration file that had to be written by the programmer and the time that was needed to implement and configure the SSH protocol, this makes it difficult to estimate the productivity increase offered by expi2java in comparison to spi2java.

This case study shows that our approach is flexible enough to be used with complex real-life protocols. The Extensible Spi Calculus and the type system with nested types, together

[3]According to the David A. Wheeler's 'SLOCCount' tool
[4]Available at http://www.rtfm.com/puretls/

with the configurations can be extended to support custom cryptographic primitives with complex behavior without changes in the implementation of the tool. The code generated by expi2java is interoperable with other implementations. The size of the generated code is in the same order of magnitude as the size of comparable implementations, however, the size of the protocol specifications and configuration that needed to be written by the programmer is significantly smaller than the size of hand-written implementations.

6 Conclusion

6.1 Contributions

Types In this thesis, we have introduced a type system with nested types for the Extensible Spi Calculus. We use parametric polymorphism and subtyping [CG92, Pie02] to achieve the flexibility and expressiveness needed for expressing real-life protocols. The type system not only expresses the data type of each term, but also contains the information about the structure of the nested terms, and complements the concept of constructors and destructors. This allows us to detect more errors early in the modelling process. We have incorporated the concept of configurations into the type system, thus ensuring the correct usage of cryptographic primitives. The type system can express many complex types, such as specializations of channels, encryptions etc. using a reasonably small set of core types and can be easily extended to have more types if needed. We have proved that the processes stay typed when evaluated, thus ensuring that the type system cannot be circumvented.

Configurations We have introduced the configurations to specify subsets of closely related types, constructors and destructors in a consistent way and to provide the implementation-specific low-level information about the types, needed for the code generation. The calculus from [AB05] was extended to support these configurations to ensure the correct usage of the specialized types together with the constructors and destructors. With some runtime support, we can define several related Spi names to belong to special virtual variables that are mostly used to represent split channels as one. This allows us to generate a working implementation of real-world protocols even though the implementation of the network channels has some implicit assumptions about the protocol structure that cannot be modelled with the currently used type system. The concrete syntax of the configurations is simple, easy to use, avoids redundant information and is flexible enough to support arbitrary additional primitives and their future implementations.

Implementation We have developed an implementation of the expi2java code generator. Expi2java is a major step forward from the original spi2java and supports all concepts described above. Although we expect the tool to still evolve over time, we made a point on keeping expi2java as extensible as possible without code changes. We support flexible file formats (Expi and Exdef) that allow to extend the set of used types, constructors and destructors with ease. The only thing that the user will need to implement is the additional cryptographic primitives. We provide definitions and implementation of the most common data types and cryptographic primitives sufficient to generate working implementations of many protocols.

The potential of the expi2java was shown by generating an interoperable implementation of the TLS protocol from about 400 lines of specification (Expi file) and 400 lines of configuration (additional Exdef file). The generated implementation performs a TLS handshake with a normal web server and downloads a web page using the HTTP protocol over the established secure connection. We support the AES-256 CBC encryption scheme with the RSA key exchange. With a small amount of work we were able to also implement the HMAC and a custom PRF.

6.2 Related Work

The idea of a code generator for security protocols is not new. In fact, several tools for automatic code generation have been developed over the past years. One of the early approaches is the AGVI toolkit by A. Perrig, D. Song and D. Phan [PSP01] that focuses on the generation of new protocols and uses the Athena [SBP01] protocol analyzer for verification of the formal model, but can also generate Java implementations.

The CIL2Java tool by J. Millen and F. Muller [MM01] can generate Java code from the CAPSL intermediate language CIL [Mil97, DM99]. CIL2Java was designed for demonstration purposes, it is not extensible and only implements a few cryptographic primitives.

SPEAR II by S. Lukell, C. Veldman and A. Hutchison [LVH03] is a tool aimed at the rapid protocol engineering. It has a graphical user interface and generates Java code that uses the ASN.1 standard for data encoding and various cryptographic libraries.

The spi2java framework by A. Pironti, R. Sisto, L. Durante and D. Pozza [PSD04, PS07] is the first code generator designed to be flexible and configurable. It uses the Spi calculus [AG99] as the input language, together with a simple type system, supports type inference and provides a way to configure the protocols with the low-level information needed for an interoperable protocol implementation. The generated implementation of an SSH client [PS07] demonstrates that interoperability with standard implementations is indeed achievable.

There is also another project that uses a similar approach, the Sprite tool by B. Tobler [Tob05]. The code generator from the Sprite tool is also called spi2java.

Spi2F# by T. Tarrach is a prototype tool aimed at the generation of protocol implementations that preserve the security properties of the formal model rather than at the interoperability [Tar08]. It uses two type systems, the type system from [FGM07, BHM08a] to check the source language [AB05] and the F7 type-checker to check the target language (a subset of F#). The preservation of the security properties is shown by proving that the typed translation between the source and the target language preserves typing.

6.3 Relation with Spi2java

The previous version of expi2java originated as an extension of the spi2java framework by R. Sisto et al. The main difference to the original spi2java was the input language. Instead of the Spi calculus [AG99] we used the more flexible Spi calculus with constructors and destructors from [AB05] that was further extended to the Extensible Spi Calculus in expi2java. The previous hardcoded translation of cryptographic primitives to Java code was replaced by a generic code generation algorithm. Finally, we refactored most of the spi2java code, making it simpler to maintain and preparing the tool for the future extensions.

In expi2java, we rewrote the remaining parts and redesigned the implementation of the type system and code generation. Expi2java no longer supports any of the input files used in the original spi2java; we built new parsers for the Expi and Exdef file formats based on the ProVerif parser from the ZK Compiler tool by D. Unruh [BMU08]. Only parts of the runtime library and some helper classes could be adapted in expi2java due to the large amount of changes.

The original spi2java uses a very simple type system. In expi2java we changed that to a more expressive and flexible one. Many features of the new type system with nested types that we discuss in Chapter 3, directly address the issues we have found in the original spi2java. The parametric types prevent the incorrect usage of constructors and destructors and eliminate a lot of implicit assumptions on the behavior of type implementations. For instance, in spi2java the decryption has the top type Message as the return type and uses a downcast to the actual term type in implementation, which may fail on runtime.

Another disadvantage of the original spi2java is the complicated workflow. The protocol file in spi2java does not contain any type information, the types are specified for every term using an additional XML configuration file. Besides the type, all low-level details must also be specified for every term, causing a lot of redundancy. For the example protocols we used, the size of these XML configuration files is between 250 lines for very simple protocols like the Secure Andrew RPC [BAN89], up to 700 lines and more for bigger protocols, like the Needham-Schroeder protocol. In contrast to that, the specification of the Needham-Schroeder protocol in the Extensible Spi Calculus together with embedded type annotations and all protocol-specific configurations is only 210 lines long.

A lot of information in the spi2java XML configuration files is generated automatically. Besides that, spi2java can infer a significant amount of type information, as far as it is possible in the simple type system. However, this advantage is diminished by the fact that the user still must adapt the low-level information or manually specialize many types. In expi2java, we introduced the configurations and typedefs, which avoid redundancy and are simpler to use. Still, we no longer do any type inference at the moment.

The terms in the XML configurations used in spi2java are referenced with an unique term ID. Since this ID is dependent on the position of the term in the protocol specification, even the smallest change in the protocol also changes the IDs of many terms in the file that is generated by the tool. The hand-made changes in the old XML file must be transferred to the new file manually, making an iterative writing or refining of a protocol specification for large protocols extremely tedious. The type annotations inside the protocol specification files we use in expi2java are more convenient in this respect and make it much easier to write large protocol specifications like the TLS protocol we discussed in Chapter 5. The possibility to generate and test the code for a partially implemented protocol without the need to merge several-hundred-lines-long XML files after every change is very convenient.

The types can be added or changed in both the original spi2java and expi2java in a similar way, by changing the type definition and implementing the type in the runtime library. However, in the more expressive and flexible type system with nested types this is often not needed, since the behavior of types can also be adapted by using other type parameters or configurations. The Spi calculus from spi2java is very limited in comparison to the Extensible Spi Calculus.

6.4 Future Work

Type Inference A very useful extension would be a type inference algorithm for nested types with a corresponding implementation. The current implementation of the type system assumes a fully annotated protocol specification. The type annotations must be specified by the user for every name, constructor and destructor application, resulting in a quite large amount of redundant information. A type inference algorithm for the type system with nested types would make it possible to omit most type annotations, resulting in more readable protocol specifications. The nested types store a lot of information about the term structure, which is helpful for the type inference. This should make it possible to infer even more type information than it was possible in the simple type system used in spi2java. It would also be beneficial if some of the type information provided by various type systems for security could be reused for code generation.

Sum Types Extending the type system with sum types [Pie02], which combine several types into one, could solve the channel splitting issues. Using the sum types, we could type every channel to the combined type of all messages we send over it and select one type when needed.

Zero-Knowledge Another very interesting direction would be to extend the type system, the code generator and the runtime library to handle zero-knowledge. Implementing symbolic zero-knowledge is a complex and challenging task that would allow for generation of a whole new class of protocols, such as Civitas [CCM08]. The protocols using zero-knowledge proofs can be analyzed by the type system from [BHM08a, BMU08, BU08].

Provable Translation It would be interesting to exchange some ideas with the approach of Spi2F# by T. Tarrach. This tool uses essentially the same input language and a type system for security that rejects insecure protocol models. By incorporating this type system into expi2java we would eliminate an important cause for security-related errors.

Combining the interoperability and flexibility of expi2java with the provable preservation of the security properties is highly desirable, but also difficult. The first important step into this direction would be to formalize a subset of Java that is sufficient to implement real-life protocols. Then we could try to formalize and prove a translation from the Extensible Spi Calculus to this fragment of Java.

TLS The implementation of the TLS protocol could also be improved. The current implementation supports only some of the features of TLS and is limited to one encryption scheme. We could check for the problems during the handshake and alert the server as defined in the specification. More important is, however, to model all the features of TLS in the Extensible Spi Calculus without special runtime support. Additionally, we could try to verify the model of the TLS protocol with ProVerif.

Synchronous I/O The implementation of the synchronous input and output from the Extensible Spi Calculus should follow the abstract semantics. It is important to implement the same behavior of the processes as expected, since some protocol specifications may rely on that.

Implementation There is also room for improvement in the implementation of expi2java. For instance, we could add some basic syntactic sugar to the calculus, i.e., allow constructors in let processes instead of using them in an identity destructor or support tuples of arbitrary length instead of the nested pairs.

It would be nice to have a possibility for a dynamic reconfiguration of protocols in a generic way. For example, it could be used to reconfigure the keys after negotiating the key length in the TLS protocol. The set of provided cryptographic primitives and data types could also be extended. More flexible and configurable implementations of the cryptographic primitives and the corresponding constructors and destructors would make the implementation of other protocols easier. For example, many real-world protocols use block encryption schemes with a dynamic initialization vector, and in this case the encryption constructor should not only take the message and the key as arguments, but also the initialization vector.

In order to make expi2java more useful in praxis, support of other target languages (e.g., C# or F#) is desirable. A GUI or an integrated plug-in for an IDE like Eclipse would make the tool more user-friendly and allow a more efficient, interactive workflow.

Bibliography

[AB01] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Proc. 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2030 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, 2001.

[AB05] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.

[Aba99] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[ABF05] M. Abadi, B. Blanchet, and C. Fournet. Automated verification of selected equivalences for security protocols. In *Proc. 20th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 331–340. IEEE Computer Society Press, 2005.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.

[BAN89] Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. Technical Report 39, Digital Systems Research Center, February 1989.

[BCFM07] M. Backes, A. Cortesi, R. Focardi, and M. Maffei. A calculus of challenges and responses. In *Proc. 5th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 101–116. ACM Press, 2007.

[BFM07] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563–617, 2007.

[BHM08a] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Type-checking zero-knowledge. To appear in 15th ACM Conference on Computer and Communications Security (CCS), October 2008. Full version available at `http://www.infsec.cs.uni-sb.de/~hritcu/publications/zk-types-full.pdf`.

[BHM08b] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Type-checking zero-knowledge. Implementation available at `http://www.infsec.cs.uni-sb.de/projects/zk-typechecker`, 2008.

[BI08] Alex Busenius and Violeta Ivanova. Extending the input language of Spi2Java. Technical report, Saarland University, April 2008. `http://www.infsec.cs.uni-sb.de/teaching/WS07/Seminar/reports/conspi.pdf`.

[Bla01] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE Computer Society Press, 2001.

[BMU08] Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *Proc. 29th IEEE Symposium on Security and Privacy*, pages 202–215. IEEE Computer Society Press, 2008.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus, May 1992. INRIA Research Report 1702.

[Bra97] S. Bradner. *Key words for use in RFCs to indicate Requirement Levels*, March 1997. RFC 2119 (Best Current Practice).

[BU08] Michael Backes and Dominique Unruh. Computational soundness of symbolic zero-knowledge proofs against active attackers. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, pages 255–269. IEEE Computer Society Press, 2008.

[Car04] Luca Cardelli. *Type Systems*, Chapter 97. CRC Handbook of Computer Sience and Engineering, 2nd edition, February 2004.

[CCM08] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: A secure voting system. In *Proc. 29th IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society Press, 2008.

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

[Cho02] P. Chown. *Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)*, June 2002. RFC 3268 (Informational).

[DA99] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*, January 1999. RFC 2246 (Proposed Standard).

[DM99] G. Denker and J. Millen. CAPSL intermediate language. In *FLoC Workshop on Formal Methods and Security Protocols*, 1999.

[FGM07] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF)*, pages 31–45. IEEE Computer Society Press, 2007.

[GJ04] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3):435–484, 2004.

[HJ06] Christian Haack and Alan Jeffrey. Pattern-matching spi-calculus. *Information and Computation*, 204(8):1195–1263, 2006.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, Mai 2001.

[LVH03] Simon Lukell, Christopher Veldman, and Andrew Hutchison. Automated attack analysis and code generation in a multi-dimensional security protocol engineering framework. In *Southern African Telecommunications Networks and Applications Conference*. University of Cape Town, 2003.

[Mil97] Jonathan Millen. CAPSL: Common Authentication Protocol Specification Language. Technical Report MP 97B48, The MITRE Corporation, 1997.

[MM01] J. Millen and F. Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT press, 2002.

[PS07] Alfredo Pironti and Riccardo Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *International Symposium on Computers and Communications (ISCC)*, January 2007.

[PSD04] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2Java: Automatic cryptographic protocol java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA 2004)*, pages 400–405. IEEE Computer Society Press, 2004.

[PSP01] A. Perrig, D. Song, and D. Phan. AGVI – Automatic Generation, Verification, and Implementation of security protocols. In *Proc. Computer Aided Verification'01 (CAV)*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[Res00] E. Rescorla. *HTTP Over TLS*, Mai 2000. RFC 2818 (Informational).

[SBP01] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.

[Tar08] Thorsten Tarrach. Spi2F# – A prototype code generator for security protocols. Bachelor's Thesis, 2008.

[Tob05] Benjamin Tobler. A structured approach to network security protocol implementation. Master's thesis, University of Cape Town, November 2005.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WS96] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *In Proceedings of the Second UNIX Workshop on Electronic Commerce*, pages 29–40. USENIX Association, 1996.