AnBx: Automatic Generation and Verification of Security Protocols Implementations

Paolo Modesti

School of Computing Science, Newcastle University, UK
paolo.modesti@newcastle.ac.uk

Abstract The *AnBx* compiler is a tool for automatic generation of Java implementations of security protocols specified in a simple and abstract model that can be formally verified. In our model-driven development approach, protocols are described in *AnBx*, an extension of the Alice & Bob notation. Along with the synthesis of consistency checks, the tool analyses the security goals and produces annotations that allow the verification of the generated implementation with ProVerif.

Keywords: security protocols, Java code generation, applied formal methods, verification

1 Introduction

In the Internet era, organisations and individuals heavily depend on the security of the network infrastructure and its software components. Security protocols play a key role in protecting communications and user's digital assets, but evidence shows [1] that despite considerable efforts, their implementation remains challenging and error-prone. In fact, low-level implementation bugs that need to be manually patched, are discovered even in ubiquitous protocols like TLS and SSH which are thoroughly tested. Indeed, a robust implementation requires the specification of the (defensive) consistency checks on the received data that need to be performed to control that the protocol is running according to the specification. However, it is important to recognize that while some checks on reception are trivially derived from the narrations (verification of a digital signature, comparison of the agent's identities), others are more complex and managing them can be a challenging task even for an expert programmer.

To counter this problem, we propose a model-driven development approach that allows automatic generation of a program, from a simpler and abstract model that can be formally verified. In this paper, we present the *AnBx Compiler and Code Generator*[1], a tool for automatic generation of Java implementations of security protocols specified in the simple Alice & Bob (*AnB*) notation [2] (or its extension *AnBx* [3]), suitable for agile prototyping.

In addition to the main contribution of an end-to-end *AnB* to Java compiler, this paper extends our previous work [4] providing a formalization of the compiler, focusing on the generation of consistency checks (enhancing on [5]) and

[1] Available at `http://www.dais.unive.it/~modesti/anbx/`

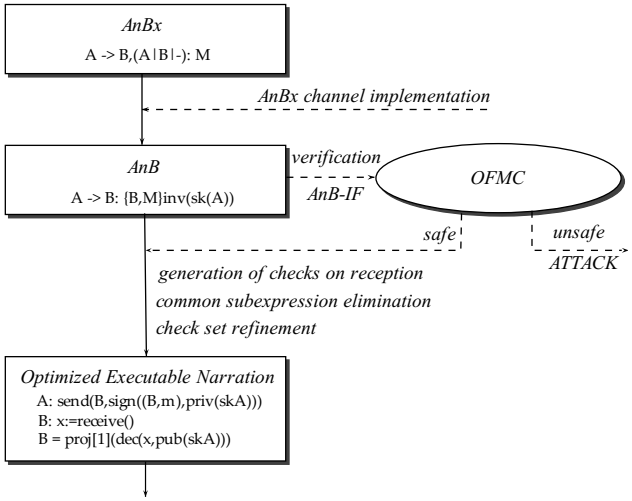

Figure 1. Compiler front-end: pre-processing, verification, *ExecNarr* optimization

on the generation of annotations of the security goals that are necessary for the verification of the implementation with ProVerif [6].

Outline of the paper In §2 we describe the architecture of the *AnBx Compiler and Code Generator*. The translation from *AnB* to the intermediate format and the construction of the implementation are described in §3. §4 focuses on the verification of the implementation and in §5 we conclude by discussing related and future work.

2 Architecture of the *AnBx* compiler

In this section, we present an overview of the compiler, which is developed in Haskell, by illustrating all the steps in the automatic Java code generation of security protocols from an *AnBx* or *AnB* model (Figures 1 and 2).

Pre-Processing and Verification $AnBx \rightarrow AnB \rightarrow$ (verification)
The *AnBx* protocol is lexed, parsed and then compiled to *AnB*, a format which can be verified with the OFMC model checker [7]. The compiler can also directly read protocols in *AnB*. The *AnBx* language is described in §3.1.

Front-end $AnB \rightarrow ExecNarr \rightarrow Opt\text{-}ExecNarr$
At this stage, if the verification is successful, the *AnB* specification can be compiled into an *executable narration* (*ExecNarr*), a set of actions that operationally encodes how agents are expected to execute the protocol. The core of this phase (§3) is the automatic generation of the consistency checks derived

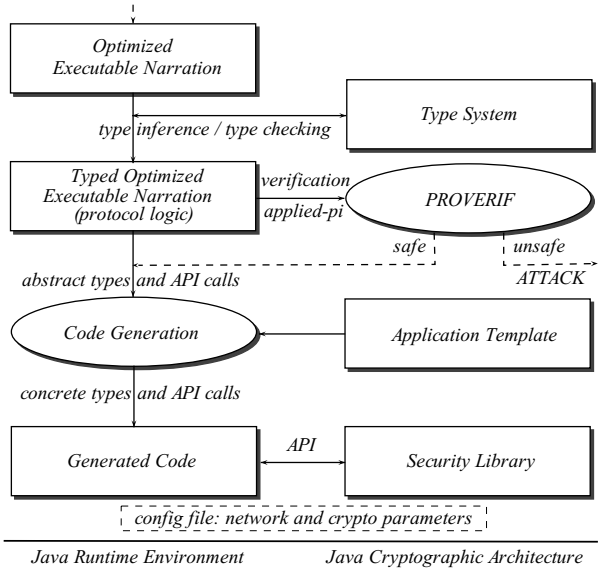

Figure 2. Compiler back-end (Type System, Code generator, Verification) and runtime support

from the static information of the protocol narrations. Checks are expressed by means of consistency formulas; the tool applies some simplification strategies which offer good results in practice, in order to reduce the number of generated formulas. A further step is the generation of the *optimized executable narration* (*Opt-ExecNarr*) [4], which applies some optimization techniques, including common subexpression elimination (CSE), which in general are useful to generate efficient code. Considering the set of cryptographic operations, which are computationally expensive, the code is optimized, in order to reduce the overall execution time. To this end, variables are instantiated to store partial results and a reordering of assignment instructions is performed with the purpose of minimizing the number of cryptographic operation.

Back-end *Opt-ExecNarr* $\rightarrow$ (protocol logic) + (application logic) $\rightarrow$ *Java*

The final stage is the generation of the Java source code from the *Opt-ExecNarr*. The previous phases are fully language independent from the target programming language considered. Moreover, we designed a versatile tool that allows for a wide range of user customizations. We summarize here the main components and their characteristics:[2]

Code generation strategy We make a distinction between the *protocol logic* and the *application logic.* The latter is implemented by means of parametrized application template files written in the target language which can be

[2] A detailed description of the compiler's back-end is available in [8].

customized by the user. This helps the integration of the generated code in larger applications. The templates are instantiated with the information (the *protocol logic*) derived from the optimized executable narration. We model the *protocol logic* by means of a language independent intermediate format called *Typed-Opt-ExecNarr*, which is, in essence, a typed representation of the *Opt-ExecNarr*. This is useful to parametrize the translation and to simplify the emission of code in other programming languages.

Type System Building the *Typed-Opt-ExecNarr* requires a type system modelling a typed abstract representation of the security-related portion of a generic procedural language supporting a rich set of abstract cryptographic primitives. The type system infers the type of expressions and variables insuring that the generated code is well-typed. It has the additional benefit of detecting at run time whether the structure of the incoming messages is equal to the expected one, according to the protocol specification.

Code emission This is performed by instantiating the protocol templates, i.e., the skeleton of the application, using the information derived from the protocol logic. It is worth noting that only at this final stage the language specific features and their API calls are actually bound to the protocol logic. To this end, two mappings are required. One between the abstract and the concrete types; the other one between the abstract actions and the concrete API calls.

Security API The run-time support relies on the cryptographic services offered by the Java Cryptography Architecture (JCA). In order to connect to the JCA, we designed an API for security which wraps, in an abstract way, the JCA interface and implements the custom classes necessary to encode the generated programs in Java. The *AnBxJ* library offers a high degree of generality and customization, since the API does not commit to any specific cryptographic solution (algorithms, libraries, providers). Moreover, the library provides access in an abstract way to the communication primitives used to exchange messages in the standard TCP/IP network environment. The generated code comes along with a configuration file that allows the developer to customize the deployment of the application at the cryptographic (keystore location, aliases, cipher schemes, key lengths, etc.) and network level (IP addresses, ports, etc.) without requiring to regenerate the application.

Verification of the implementation The *Typed-Opt-ExecNarr* can be translated into Applied pi-calculus and verified with ProVerif [6]. This requires that the *AnB* security goals are analysed and specific annotations modelling the security properties are generated along the compilation chain. The verification of the implementation is described in §4.

3 Construction of the Implementation

We now describe how protocols in *AnB* can be compiled into *ExecNarr*. The goal is to obtain an operational description of the actions each agent has to perform, including the informative checks on reception of messages.

```
Protocol: CreditCard
Types:
        Agent C,M,A;
        Certified C,M,A;
        Function [Agent,Agent -> Number] ccn
Knowledge:
        C: C,M,A;
        M: C,M,A;
        A: C,M,A;
        C,A share ccn(C,A)
Actions:
        C -> M,(C|A|A): ccn(C,A)
        M -> A,(C|A|A): ccn(C,A)
Goals:
        A weakly authenticates C on ccn(C,A)
        ccn(C,A) secret between C,A
```

Fig. 3. *AnBx* protocol - Example

3.1 The *AnBx* language

The *AnBx* language [3] is built as an extension of *AnB*, whose description and formal semantics are available in [2]. *AnBx* uses channels as the main abstraction for communication, providing different authenticity and/or confidentiality guarantees for message transmission, including a novel notion of *forwarding* channels, enforcing specific security guarantees from the message originator to the final recipient along a number of intermediate forwarding agents. The translation from *AnBx* to *AnB*, can be parametrized using different channel implementations, by means of different cryptographic operations.

The example in Figure 3 depicts a (hyper-simplified) communication pattern common in e-commerce protocols like iKP [9] and SET [10]. To complete a payment, a customer `C` needs to send its credit card number `ccn(C,A)` to the acquirer `A`, through the merchant `M`, as these protocols do not contemplate direct exchange of messages between `C` and `A`. The goal of the protocol is dual: the secrecy of the credit card should not be compromised and `A` should be convinced that message has originated from `C`.

Some peculiarities of the *AnBx* syntax are the following. In `Type` section `Certified C,M,A` declares that these agents can digitally sign and encrypt messages. The function `ccn` with signature `[Agent,Agent -> Number]` is used to model abstractly a credit card number. Concretely, the statement in the `Knowledge` section `C,A share ccn(C,A)` means that `C` and `A` know the credit card number before the protocol execution. The action `C -> M,(C|A|A): ccn(C,A)`, means that the payload is digitally signed by `C`, verifiable by `A`, and confidential for `A`.

Goals in *AnBx* specify the security properties that the protocol is meant to convey. They can also be translated into low level goals suitable for the verification with various tools. We support three standard *AnB* goals:

Weak Authentication goals have the form `B weakly authenticates A on M` and are defined in terms of non-injective agreement [11];

Authentication goals have the form `B authenticates A on M` and are defined in terms of injective agreement on the runs of the protocol, assessing the freshness of the exchange;

Secrecy goals have the form `M secret between A1,...,An` and are intended to specify which agents are entitled to learn the message M at the end of a protocol run.

3.2 Protocol Compilation

The intermediate format used by the compiler (*ExecNarr*) is composed by two sections: a *declaration* and the actual *narration*. The *declaration* includes the initial knowledge of each agent, the names generated by them and the names that are assumed to be initially known only by a subset of agents, similarly to the `share` construct. The syntax of *ExecNarr*, which extends the one presented in [5], is shown in Table 1. The *agents* are taken from set of agent names **A**, and the *messages* are built upon set of names **N**. We also consider a set of user-defined functions **F**. It is assumed that **A**,**F**,**N** are mutually disjoint.

As a first step of the compilation process, we need to derive the *declaration* section from the *AnB* agent's knowledge mapping the `Knowledge` of the protocol. A function $\tau : \mathbf{M_{AnB}} \rightarrow \mathbf{M}$ translates the *AnB* messages to their equivalent in *ExecNarr*, where $\mathbf{M_{AnB}}$ and **M** are the sets of messages in the two formats.

A core component of the translation from *AnB* to the executable narration format is the computation of the checks on reception extending and refining the ideas proposed by Briais and Nestmann [5].

However, we improve [5] on three directions. First, a major contribution of the present paper is the translation of security goals allowing for verification of the implementation of the protocol (§4). Second, we support a richer language allowing to model a larger class of real-world protocols, introducing operators like *hmac*, *kap*, *kas* and user defined functions. *kap* and *kas* are used to model the basic operations on keys which are available in key agreement protocols like Diffie-Hellman [12]. They satisfy the algebraic property $kas(kap(g, x), y) \approx kas(kap(g, y), x)$, given the pre-shared parameter g. Third, we dramatically improved the performance of the compiler as shown in [4].

The *AnB* actions are translated to produce an operational description of the steps each agent has to perform. Atomic exchanges of the form $A \rightarrow B : M$ are compiled to a more specific set of basic actions:

emission $A : \texttt{send}(B, E)$ of a message expression E (evaluating to M);
reception $B : x := \texttt{receive}()$ of a message and its binding to a fresh variable name x, where $x \in \mathbf{V}$, the set of variables, mutually disjoint from **A**, **F**, **N**;
check $B : \phi$ for the validity of the formula ϕ from the point of view of agent B.

In addition, we define two additional basic actions that may be performed during the protocol execution and goal annotations:

scoping $A : \texttt{new}\, k$, represents the creation and scope of private names;
assignment $A : x := E$, the variable x assume the value of the expression E.
goal event $A : \gamma$, a goal annotation γ from the point of view of agent A.

expressions **E**			
E, F	::=	a	*name*
	|	A	*agent's name*
	|	x	*variable*
	|	$hash(E)$	*hashing*
	|	$pub(E)$	*public key*
	|	$priv(E)$	*private key*
	|	$(E_1, ..., E_n)$	*tuple* * $E_i \in \mathbf{E},\ i \in \{1..n\}$
	|	$\pi_i(E)$	*i−th projection* *
	|	$enc(E, F)$	*asymmetric encryption*
	|	$encS(F, F)$	*symmetric encryption* *
	|	$dec(E, F)$	*asymmetric decryption*
	|	$decS(E, F)$	*symmetric decryption* *
	|	$hmac(E, F)$	*hmac* *
	|	$kap(E, F)$	*key agreement half key* *
	|	$kas(E, F)$	*key agreement full key* *
	|	$E(F)$	*function* *
formulae			
ϕ	::=	$[E = F]$	*equality check*
	|	$[E : \mathbf{M}]$	*well−formedness test*
	|	$inv(E, F)$	*inversion test*
events*			
Q	::=	**witness**|**request**	
	|	**wrequest**|**secret**	
goal labels*			
L	::=	l	*goal label*
goals events*			($A_1, ..., A_n$ *are agent's names*)
γ	::=	$Q(L, E, (A_1, ..., A_n))$	*goal event*
actions			(A, B *are agent's names*)
I	::=	A : **new** k	*fresh name generation*
	|	$A : send(B, E)$	*message emission*
	|	$A : x := receive()$	*message reception*
	|	$A : x := E$	*assignment* *
	|	$A : \phi$	*check*
	|	$A : \gamma$	*goal event* *
narrations			
N	::=	ϵ	*empty narration*
	|	$I; N$	*non empty narration*
declarations			
D	::=	A **knows** M	*initial knowledge* (M *is a ground expression*)
	|	A **generates** n	*fresh name generation*
	|	**private** k	*private name*
protocol			
P	::=	$D; P \mid N$	*declarations + narration*

Table 1. Syntax of the executable narrations (Extensions with respect to [5] are marked with *. Moreover, previously pairs ($E.F$) were used instead of tuples ($\star$).)

$$\text{ANA}-\text{INI}\ \frac{(M,E) \in K}{(M,E) \in \mathcal{A}_0(K)}$$

$$\text{ANA}-\text{OP1}\ \frac{(op(M),E) \in \mathcal{A}_n(K)}{(op(M),E) \in \mathcal{A}_{n+1}(K)}\ op \in \{pub, priv, hash\}$$

$$\text{ANA}-\text{OP2}\ \frac{(op(M,N),E) \in \mathcal{A}_n(K)}{(op(M,N),E) \in \mathcal{A}_{n+1}(K)}\ op \in \{hmac, kap, kas\}$$

$$\text{ANA}-\text{FUN}\ \frac{(M(N)),E) \in \mathcal{A}_n(K) \quad M \in \mathbf{F}}{((M(N)),E) \in \mathcal{A}_{n+1}(K)}$$

$$\text{ANA}-\text{PROJ}\ \frac{((M_1,...,M_m),E) \in \mathcal{A}_n(K)}{((M_i,\pi_i(E)) \in \mathcal{A}_{n+1}(K)}\ i \in \{1..m\}$$

$$\text{ANA}-\text{DEC}\ \frac{(enc(M,N),E) \in \mathcal{A}_n(K) \quad (inv(N),F) \in \mathcal{S}(\mathcal{A}_n(K))}{(M,dec(E,F) \in \mathcal{A}_{n+1}(K)}$$

$$\text{ANA}-\text{DECS}\ \frac{(encS(M,N),E) \in \mathcal{A}_n(K) \quad (inv(N),F) \in \mathcal{S}(\mathcal{A}_n(K))}{(M,decS(E,F) \in \mathcal{A}_{n+1}(K)}$$

$$\text{ANA}-\text{DEC}-\text{REC}\ \frac{(op(M,N),E) \in \mathcal{A}_n(K) \quad (inv(N),F) \notin \mathcal{S}(\mathcal{A}_n(K))}{(op(M,N),E) \in \mathcal{A}_{n+1}(K)}\ op \in \{enc, encS\}$$

$$\text{ANA}-\text{NAM}-\text{REC}\ \frac{(M,E) \in \mathcal{A}_n(K) \quad M \in \mathbf{N} \cup \mathbf{A}}{(M,E) \in \mathcal{A}_{n+1}(K)}$$

Table 2. Analysis ANA-rules

Generation of consistency checks Formulas ϕ on received messages are described by a conjunctions of three kinds of checks:

equality $[E = F]$ denoting the comparison of two expressions E and F;
well-formedness $[E : \mathbf{M}]$ denoting the verification of whether the projections and decryption contained in E are likely to succeed;
inversion $inv(E, F)$ denoting the verification that E and F evaluate to inverse messages.

Since consistency checks will have to operate on (*message*,*expression*) pairs, the representation of the agent's knowledge must be generalized. The idea is that a pair (M, E) denotes that an expression E is equivalent to the message M. For this reason, it is necessary to introduce the notion of *knowledge sets*, and two operations on them: *synthesis* reflecting the closure of knowledge sets using message constructors; *analysis* reflecting the exhaustive recursive decomposition of knowledge pairs as enabled by the currently available knowledge.

Formally these sets and operations are defined as follows with the necessary adaptations from [5]:

Definition 1 (Knowledge).
Knowledge sets $K \in \mathbf{K}$ *are finite subsets of* $\mathbf{M} \times \mathbf{E}$.
The analysis $\mathcal{A}(K)$ *of* K *is* $\bigcup_{n \in \mathbb{N}} \mathcal{A}_n(K)$ *where the sets* $\mathcal{A}_i(K)$ *are the smallest sets satisfying the* ANA*-rules in Table 2.*

$$\textsf{SYN}-\textsf{OP1}\ \frac{(M,E)\in\mathcal{S}(K)}{(op(M),op(E))\in\mathcal{S}(K)}\ op\in\{pub,priv,hash\}$$

$$\textsf{SYN}-\textsf{OP2}\ \frac{(M,E)\in\mathcal{S}(K)\quad(N,F)\in\mathcal{S}(K)}{(op(M,N),op(E,F))\in\mathcal{S}(K)}\ op\in\{hmac,kas,kap\}$$

$$\textsf{SYN}-\textsf{ENC}\ \frac{(M,E)\in\mathcal{S}(K)\quad(N,F)\in\mathcal{S}(K)}{(op(M,N),op(E,F))\in\mathcal{S}(K)}\ op\in\{enc,encS\}$$

$$\textsf{SYN}-\textsf{TUPLE}\ \frac{(M_1,E_1)\in\mathcal{S}(K)\quad ...\quad (M_m,E_m)\in\mathcal{S}(K)}{((M_1,...,M_m),(E_1,...,E_m))\in\mathcal{S}(K)}\ i\in\{1..m\}$$

$$\textsf{SYN}-\textsf{FUN}\ \frac{(M,E)\in\mathcal{S}(K)\quad(M,F)\in\mathcal{S}(K)\quad M\in\mathbf{F}}{(M(N),(E(F))\in\mathcal{S}(K)}$$

$$\textsf{SYN}-\textsf{KAP}\ \frac{(M,E)\in\mathcal{S}(K)\quad(N,F)\in\mathcal{S}(K)\quad M\in\mathbf{N}}{(kap(M,N),kap(E,F))\in\mathcal{S}(K)}$$

$$\textsf{SYN}-\textsf{KA}-\textsf{EQ}\ \frac{(kas(kap(M,N),O),kas(kap(E,F),G)\in\mathcal{S}(K)\quad M\in\mathbf{N}}{(kas(kap(M,O),N),kas(kap(E,G),F)\in\mathcal{S}(K)}$$

Table 3. Synthesis SYN-rules

The synthesis $\mathcal{S}(K)$ *of* K *is the smallest subset of* $\mathbf{M}\times\mathbf{E}$ *containing* K *and satisfying the* SYN*-rules in Table 3. In addition, we define a variant of the* synthesis $\mathcal{S}^\star(K)$ *of* K *as the smallest subset of* $\mathbf{M}\times\mathbf{E}$ *containing* K *and satisfying the* SYN*-rules in Table 3 excluding the* SYN-ENC *rule.*

With respect to the original work [5] we defined $\mathcal{S}^\star$ and we added the SYN-rules SYN-OP2, SYN-FUN, SYN-TUPLE, SYN-KAP, SYN-KA-EQ and the ANA-rules ANA-OP2, ANA-FUN, ANA-PROJ in order to support a more expressive language like *AnB*. These new rules are necessary to generalize the notion of synthesis and analysis with functions and operators defined in *AnB*, and previously unavailable in the original work. It is worth noting that the SYN-KA-EQ rule is necessary to model the algebraic equivalence $kas(kap(g,x),y)\approx kas(kap(g,y),x)$. More equational theories could be supported by adding ad-hoc rules.

During the protocol execution the initial knowledge set is extended, according to the information learned by the reception actions: the expected message and the corresponding expression.

Definition 2 (Consistency Checks).
Let K *be a knowledge set. Its* consistency formula $\Phi(K)$ *is defined as follows:*

$$\begin{aligned}\Phi(K) := &\textstyle\bigwedge_{(M,E)\in K}[E:\mathbf{M}]\\ &\wedge\ \textstyle\bigwedge_{(M,E_i)\in K\wedge(M,E_j)\in\mathcal{S}^\star(K)\wedge E_i\neq E_j}[E_i=E_j]\\ &\wedge\ \textstyle\bigwedge_{(M,E_i)\in K\wedge(inv(M),E_j)\in S(K)} inv\,(E_i,E_j)\end{aligned}$$

The first conjunction clause checks that all expressions can be evaluated, the second checks that if there are several ways to build a message M, then all the corresponding expressions must evaluate to the same value. We can see here that $\mathcal{S}^\star$ is introduced to avoid computing any equality check which requires

synthesizing new terms using symmetric and asymmetric encryption. In fact, in concrete implementations, non-deterministic encryption schemes are employed and therefore, those checks are going to fail anyway. It is important to underline that this does not undermine the robustness of the application because we just prune checks failing due to the over approximation of the abstract model. The third conjunction clause checks that if it is possible to generate a message M and its inverse $inv(M)$, then the corresponding expressions must also be mutually inverse. The generation of the consistency formulas implies comparing pairs taken from K, with pairs taken from the synthesis of K. Knowledge sets can often be simplified without loss of information, i.e. without undermining the computation of the consistency formula.

Definition 3 (Irreducibles).
Let K be a knowledge set, $OP_1 = \{pub, priv, hash\}$ the set of the unary operators, $OP_2 = \{enc, encS, hmac, kap, kas\}$ the set of binary operators and $\mathbf{F}$ the set of user-defined functions. The set of irreducibles *$\mathcal{I}(K)$ is defined as*
$\mathcal{I}(K) = irr\left(\mathcal{A}(K)\right)$, *where*

$$\begin{aligned} irr(K) \quad := \quad & \{(M,E) \in K \mid M \in \mathbf{A} \cup \mathbf{N}\} \\ \bigcup \quad & \{((M_1,...,M_n),E) \in K \mid \forall F\,(M_i,F) \notin \mathcal{S}(K)\ \forall i \in \{1..n\}\} \\ \bigcup_{op \in OP_1 \cup \mathbf{F}} \quad & \{(op(M),E) \in K \mid \forall F\,(M,F) \notin \mathcal{S}(K)\} \\ \bigcup_{op \in OP_2} \quad & \{(op((M,N),E) \in K \mid \forall F\,(M,F) \notin \mathcal{S}(K) \wedge \forall G\,(N,G) \notin \mathcal{S}(K)\} \end{aligned}$$

Let $\sim$ denote the equivalence relation on $\mathbf{M} \times \mathbf{E}$ induced by $(M,E) \sim (N,F) \iff M = N$. $rep(K)$ denotes the result of deterministically selecting one representative element for each equivalent class induced by $\sim$ on K.

Compilation The above notions are the elements required to compile the *AnB* protocol to *ExecNarr*. The translation function keeps track of the global information regarding variables used, private names, generated names, and agents' local knowledge. To model the latter we define a function $\boldsymbol{k} : \mathbf{A} \rightarrow \mathbf{K}$, mapping agents' names to their current knowledge.

The compilation of $A \rightarrow B : M$ checks that M can be synthesized by A, instantiate a new variable x and adds the pair (M,x) to the knowledge of B. The consistency formula $\Phi(\mathcal{A}(K'_B))$ of the analysis of the updated knowledge K'_B defines the checks ϕ to be performed by B at run-time.

Our compilation process extends the one formalized in [5] in two fundamentals aspects. First, it considers an extended language as described above. Second, it handles the generation of events related to security goals that was previously not considered. The compilation can be summarized as follows: if $A \neq B$ and $\exists E.\,(\tau(M),E) \in \mathcal{S}(\boldsymbol{k}(A))$, we can compile the *AnB* action $A \rightarrow B : M$ as a sequence of basic actions in *ExecNarr*. In detail:

$A : \gamma_A$
$A : \texttt{send}(B,E)$
$B : x := \texttt{receive}()$
$B : \phi$

$B : \gamma_B$

where x is a fresh variable storing the incoming message, $\boldsymbol{k}(A)$ and $\boldsymbol{k}(B)$ are the partial mappings of the knowledge set for the two agents, $K'_B = \boldsymbol{k}(B) \cup \{(M, x)\}$ is the updated knowledge of the agent B, $\phi = \Phi(\mathcal{A}(K'_B))$ is the formula representing the consistency checks, γ_A and γ_B are sets of goal annotations, computed as we explain in the next section. The updated knowledge of the agent B, in the reduced form, $\boldsymbol{k'}(B) = rep(\mathcal{I}(K'_B))$, is made available for the compilation of the next protocol action.

4 Verification of the Implementation

4.1 Compiling Security Goals

The standard approach of verification tools like OFMC [7] and ProVerif [6] is to model secrecy goals as reachability properties and authentication goals as correspondence assertions. In order to verify the implementation, *AnB* security goals must be analysed and specific annotations (events) modelling the security properties need to be generated along the compilation chain. To build the annotations, our approach is inspired by the translation from *AnB* to IF done in OFMC [13]. However, since IF is not suitable to encode consistency checks in an imperative style as the one used by *ExecNarr*, we found it practical to translate our encoding into Applied pi-calculus which can be verified by ProVerif.

Let **G** be the set of goals of the *AnB* protocol. Abstractly, *authentication goals* can be expressed in the general form $g \coloneqq ((A_1, A_2), goaltype, M)$ where A_2 is the "originator/sender" agent, A_1 is a "recipient/receiver" agent, and M is the message that the goal g is meant to convey. In *ExecNarr*, the structure of a single goal annotation γ for an agent A is $Q(L, E, (A_1, A_2))$, where Q is a goal event (`wrequest` or `request` or `witness`), L is a goal label, E is an expression that represents the message M from the perspective of A, and A_1, A_2 are the agent's names. Goal labels must be unique for each goal and corresponding assertions must share the same label. Instead *secrecy goals* have the abstract form $g \coloneqq ((A_1, ...A_n), secret, M)$ where $A_1, ...A_n$ is a list of agents' names (the secrecy set), and M is the message meant to stay secret among the agents. In *ExecNarr* the structure of a single goal annotation γ for an agent A is $Q(L, E, (A_1, ...A_n))$ where Q is a goal event `secret`, L is a goal label, E is an expression that represents the message M from the point of view of A, and $A_1, ...A_n$ the secrecy set. Since annotations for the secrecy goals are generated in a different way, we first discuss only the authentication goals.

Authentication goals Initially, we consider two identical copies of the **G** set, named G^0_S and G^0_R. During the compilation process, these two sets are analysed and consumed, from the point of view of the sender and the receiver respectively. Consumed means that for each action $A \to B : M$, the compiler considers only the goals for which it is possible to synthesize the message specified in the goal according to the current agent's knowledge and for those generates

the corresponding annotations. Once these messages are synthesized, and the annotation is generated, these goals are removed from the goal sets. We recall here that for goals expressed by means of a correspondence assertion, one *begin* event must be generated on the sender side and one *end* event must generated on the receiver side.

We compile all the protocol actions in sequence with the following procedure. Given a protocol action $A \rightarrow B : M$, an authentication goal $g := ((B', A'), goaltype, M')$ and a sender goal set G_S, a subset G'_S is computed:

$$G'_S = \{g \in G_S | \exists E. (\tau(M'), E) \in \mathcal{S}(\boldsymbol{k}(A)) \wedge A = A'\}$$

This is the set of goals where the message M' can be synthesized by A. Then for each $g \in G'_S$ the compiler generates a *begin* event. We denote γ_A the set of all generated events on the A side at this step. For example, if g is a weak authentication goal, the following event is generated: `witness(_wauth_MSGBA,E,(B',A))`, where the event type Q=`witness` and goal label L=`_wauth_MSGBA`.

Similarly, given the receiver set of goal G_R, a subset G'_R is computed:

$$G'_R = \{g \in G_R | \exists E. (\tau(M'), E) \in \mathcal{S}(\boldsymbol{k'}(B)) \wedge B = B'\}$$

It should be noted that this time we synthesize M' from $\boldsymbol{k'}(B)$, the updated local knowledge of B in the reduced form, which includes $\{(M, x)\}$, the incoming message M and the associated variable x. Therefore, we try to generate the *end* event as soon as the receiving agents can synthesize the goal message, but we position them, in the generated code, after the last usage of the message (checks included). For each $g \in G'_R$ the compiler generates an *end* event. We denote γ_B the set of all generated event on the B side. For example, if g is a weak authentication goal, the following event is generated: `wrequest(_wauth_MSGBA,E,(B,A'))`, where the event type Q=`wrequest` and goal label L=`_wauth_MSGBA`.

The labels of the two events must be identical, in order to link them, when proving the agreement. To this end, we underline that in order to have a precise verification of the security goals, it is crucial the position where these annotations are placed into the generated code. This guarantees that the goals are "reachable" (in ProVerif) it also makes the verification more efficient, as it strengthens the corresponding property [14].

Modelling the injective agreement is similar, we just replace the `wrequest` predicate with `request`. It should be noted that the generation of the two corresponding assertions (*begin*/*end* events) in general implies compiling two different actions. This is the reason why we consider two sets of goals G_S and G_R. The authentication goal in Figure 3 is a clear example of this as the two agents never exchange a message directly. In fact, managing only a single set of authentication goals may result in an imprecise translation in the cases where only one of the agents involved in the action, can synthesize the goal expression but not the other. After compiling this protocol action, we compute two new sets of goals $G''_R = G_R \setminus G'_R$, $G''_S = G_S \setminus G'_S$, and then apply the procedure to the next action, and so on.

Typed-Opt-ExecNarr	*Applied pi*
A : **new** k	`new` k
$A : send(B, E)$	$out\,(ch, E)$
$A : x := receive()$	$in\,(ch, x)$
$A : x := E$	`let` $x = E$ `in`
$A : E = F$	`if` $eq\,(E, F)$ `then`
$A : wff\,(E)$	`if` $eq\,(E, E)$ `then`
$A : inv\,(E, F)$	`if` $eq\,(decS\,(encS\,(E, F), F)), E)$ `then`
$A : Q\,(L, E, (A_1, ..., A_n))$	$\begin{cases} out\,(ch, encS\,(L, E)) & if\ Q = \textbf{secret} \\ \texttt{event}\ Q + L\,(E, A_1, ..., A_n) & otherwise \end{cases}$

Table 4. Translation of executable narrations into Applied pi (where + is the concatenation operator, *ch* is the plain channel, *eq* is the equality function)

Secrecy goals In order to verify secrecy goals, verification tools investigate whether an expression can become available to the attacker. At the current step of the compilation we generate one `secret` event for every agent belonging to the secrecy set $\{A_1, .., A_n\}$, provided the secret message M' can be synthesized by the agent. For agent A_i is checked if $\exists E.\,(\tau\,(M'), E)$. Then an event of the form `secret(<label>,E,(A1,..,An))` is generated and appended at the end of the actions for each agent. The label must be the same for all events associated with this goal.

4.2 Translation into Applied pi and Verification with ProVerif

After the generation of the *ExecNarr*, the compiler performs an optimization step and generates a typed representation of the implementation called *Typed-Opt-ExecNarr*. This language-independent format is used for the code emission in the target language (Java) which is done mapping one action of *Typed-Opt-ExecNarr* to one action in Java. The verification of the soundness of the translation up to this step is done with ProVerif. The translation into Applied pi requires several steps: (1) the generation of a prelude that includes cryptographic primitives, constructors, destructors and security goals used in the protocol, the definition of (2) a specific process which models the agents' actions for each agent, (3) a main process than orchestrates the agent's processes, and (4) an initialization process that initialize the whole system. The generation of the prelude is rather standard with cryptographic primitives defined as usual in ProVerif.

For the definition of *authentication goals* we consider the annotations of *end* events $Q\,(L, E, (A_1, A_2))$ and for each of them we generate the following goal definition: "`query m:bitstring, a1:bitstring, a2:bitstring`" + *inj* + "`event(`" + Q + L + "`(m,a1,a2)) ==> `" + *inj* + "`event(witness`" + L + "`(m,a1,a2))`" where *inj*= "`inj-`" if Q= "`request`" (strong authentication), otherwise is the empty string (weak authentication). It should be noted that since this is a general definition of the goal, we can freely use generic parameters

```
(* Process M *)
let process_M(A:bitstring,C:bitstring,M:bitstring,InvpkM:bitstring,InvskM:
    bitstring,honestC:bitstring,honestA:bitstring) =
in(ch,VAR_M_R0:bitstring);
out(ch,VAR_M_R0);
if C = honestC && A = honestA then
out(ch,encS(InvskVSMM,InvskM));
out(ch,encS(InvpkVPMM,InvpkM));0.
(* Process C *)
let process_C(A:bitstring,C:bitstring,M:bitstring,CcnCA:bitstring,pkA:
    bitstring,InvpkC:bitstring,InvskC:bitstring,honestM:bitstring,honestA:
    bitstring) =
event witness_wauth_CCNCAAC(CcnCA,A,C);
out(ch,enc(sign((A,CcnCA),InvskC),pkA));
if M = honestM && A = honestA then
out(ch,encS(CCNCACA,CcnCA));
out(ch,encS(InvskVSCC,InvskC));
out(ch,encS(InvpkVPCC,InvpkC)); 0.
(* Process A *)
let process_A(A:bitstring,C:bitstring,M:bitstring,CcnCA:bitstring,skC:
    bitstring,InvpkA:bitstring,InvskA:bitstring,honestC:bitstring,honestM:
    bitstring) =
in(ch,VAR_A_R1:bitstring);
let VAR_A_DDAR1VPAUSC:bitstring = verify(dec(VAR_A_R1, InvpkA), skC) in
if eq(A,proj_1_2(VAR_A_DDAR1VPAUSC)) then
if eq(CcnCA,proj_2_2(VAR_A_DDAR1VPAUSC)) then
if eq(decS(encS(dec(VAR_A_R1,InvpkA),dec(VAR_A_R1,InvpkA)),dec(VAR_A_R1,
    InvpkA)),dec(VAR_A_R1,InvpkA)) then
if C = honestC && M = honestM then
out(ch,encS(CCNCACA,CcnCA));
out(ch,encS(InvskVSAA,InvskA));
out(ch,encS(InvpkVPAA,InvpkA));
event wrequest_wauth_CCNCAAC(CcnCA,A,C); 0.
```

Figure 4. Translation into Applied pi of the Example (fragment)

as `m`,`a1`,`a2`. For the definition of the *secrecy goals* **secret** $(L, E, (A_1, ...A_n))$ we define: "`free` " $+ L +$ "`:bitstring[private].query attacker(` " $+ L +$ "`)`".

For the generation of the agent's process, the translation of actions is described in Table 4. Secrecy events are translated into outputs of encrypted terms. We encrypt the label L with the expression E which is used as key. If the key is compromised the expression becomes known by the attacker, and then L; therefore, the goal is violated.

In Figure 4, we show a fragment of the translation of the three processes in Applied pi of the example Figure 3. For each agent a process is generated. Process `M` does not contain events annotations because `M` acts only as a blind forwarder of the message from `C` to `A`. Agent `C` registers a `witness` event linked to the credit card number `CcnCA` and outputs a signed and encrypted message on the public channel `ch`. More interestingly, `A` receives a message on the public channel, decrypts the message and verifies the digital signature of `A`. Then `C` checks if the payload is equal to the information already possessed and then, if the check is successful, registers a `wrequest` event linked to the credit card number. If the check fails, the correspondence cannot be proved, and therefore there is an attack. Each agent's process is parametrized and actions are translated according to Table 4. The parameters of the process are the free names of each

process, plus the `honestX` parameters which are used to distinguish the runs of the honest agents from the runs which may include the intruder (in this case goals are trivially violated). For the generation of the main process, we consider the parallel execution of an arbitrary number of sessions. The process that initializes the system declares the agent names, sends them on the plain channel and makes them available to the attacker along with the public keys. Moreover, the shared values (as the credit card number) are declared. These parameters are passed to the main process and then to the single processes. An unbounded number of instances of the initialization process are generated to define the most general instantiation of the protocol.

4.3 Experimental Results and Tool Evaluation

To experiment and validate our approach we considered a test protocol suite, which includes, along with the *AnB* examples in the OFMC distribution, complex e-commerce protocols like SET [10] and iKP [9]. We compared, for this set of protocols, the results of the analysis performed by OFMC and ProVerif, in order to check if they provide the same assessments in terms of protocol safety or detection of (known) attacks. Although this is not a formal proof of correctness, we think that this comparison may provide a significant experimental evidence of the soundness of the translation steps along the compilation chain from *AnB* to Applied pi. No new attacks, in Dolev-Yao intruder model, should have been introduced. We found that the two tools provide the same results, with the following caveats. Firstly, ProVerif cannot prove injective agreements if freshness is achieved using sequence numbers. However, if non-injective agreements can be proved, and the sequence number is used as a parameter in annotations, the injectivity, given the uniqueness of the number, can also be derived, but, for a fully automated proof, it would be necessary to use tools able to model set membership, for example Set-pi [15]. It should be noted that this is not a limitation of the Applied pi language itself but a consequence of how ProVerif models sets. Secondly, it is worth noting that while ProVerif verifies for an unbounded number of sessions, for large protocols like SET and iKP OFMC struggles to verify two sessions. Therefore, a direct comparison in these cases may not be immediate. On the performance side we found that ProVerif is generally faster than OFMC, but in a few cases is unable to terminate the analysis. In these cases a few techniques like reordering of terms in a message or tagging or mentioning explicitly the arguments in `new` instructions may help ProVerif to terminate. However, it should be noted that in the latter case the analysis preformed by ProVerif could be less precise, therefore, the previous techniques should be preferable.

On the formal side, the soundness of the translation from *AnBx* to *AnB*, for a specific channel implementation, has been proven in [16]. At the moment, we do not verify the concrete Java code which may be part of the future work. However, we believe that the verification of *Typed-Opt-ExecNarr* is a crucial step for the validation of the protocol implementation, being the last step before code emission.

5 Related Work and Conclusions

The tool presented in this paper allows for the specification of security protocols in *AnBx*, an extension of the Alice & Bob notation, and automatically generates Java implementations, including the checks on receptions, which are crucial for building robust code. Some tools proposed in the past required the manual encoding of consistency checks and, in contrast with those using process calculi as an input language [17,18,19], we think that an intuitive specification language makes the model-driven approach more suitable for a larger audience of developers. JavaSPI [20], an evolution of Spi2Java, uses Java both as a modelling and as an implementation language. Our abstract specification is succinct, while, for example, the Spi calculus requires long specification files and type annotations [18], which are also required in [21]. Instead, apart from a few naming conventions, our tool delegates the duty to generate well-typed code entirely to the type system.

Two recent works considered the generation of implementations from an Alice & Bob specification. The first one, SPS [22], uses the notion of *formats* to abstract the structure of real-world protocols and computes the checks on reception proving the correctness of the translation with respect to the semantics of [2]. The tool automatically generates JavaScript specifications for the execution environment of the FutureID project [23], which may require some manual encoding. The other one [24] proposes a translation from an Alice & Bob specification into an intermediate representation verifiable with Tamarin [25]; the paper illustrates how to derive the checks but the tool does not generate concrete implementations in a programming language. Instead, our tool generates with one-click Java code that is directly runnable, thanks to the support of the integrated *AnBxJ* security library.

We can currently verify the abstract model with OFMC, and deriving annotations from the security goals the implementation (up to the code emission) with ProVerif. For future work, it would be important to verify the final Java code, along with trying to build a mechanized proof of correctness of the translation chain. Another possible extension could be the generation of interoperable implementations. However, *AnBx* is meant more as a design language rather than a mere specification language and therefore, from this point of view, is more amenable for designing new applications or re-engineering existing protocols. A further opportunity could be to plug the tool into an existing Integrated Development Environment (IDE) such as Eclipse [26] experimenting with professional programmers the effectiveness of the model-driven approach proposed by the *AnBx compiler* in a more realistic software development environment.

Acknowledgements This work was partially supported by the EU FP7 Project no. 318424, "FutureID: Shaping the Future of Electronic Identity" (futureid.eu). The author thanks Michele Bugliesi, Thomas Groß and Sebastian Mödersheim for useful discussions and Bruno Blanchet for his support on the use of the ProVerif tool.

References

1. Avalle, M., Pironti, A., Sisto, R.: Formal verification of security protocol implementations: a survey. Formal Aspects of Computing **26**(1) (2014) 99–123
2. Mödersheim, S.: Algebraic properties in Alice and Bob notation. In: International Conference on Availability, Reliability and Security (ARES 2009). (2009) 433–440
3. Bugliesi, M., Modesti, P.: AnBx-Security protocols design and verification. In: Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security: Joint Workshop, ARSPA-WITS 2010, Springer-Verlag (2010) 164–184
4. Modesti, P.: Efficient Java code generation of security protocols specified in AnB/AnBx. In: Security and Trust Management - 10th International Workshop, STM 2014, Proceedings. (2014) 204–208
5. Briais, S., Nestmann, U.: A formal semantics for protocol narrations. Theor. Comput. Sci. **389** (December 2007) 484–511
6. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Computer Security Foundations Workshop, IEEE, IEEE Computer Society (2001) 0082–0082
7. Basin, D., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. International Journal of Information Security **4**(3) (2005) 181–208
8. Modesti, P.: Efficient Java code generation of security protocols specified in AnB/AnBx. Technical Report CS-TR-1422, School of Computing Science, Newcastle University (2014)
9. Bellare, M., Garay, J., Hauser, R., Herzberg, A., Krawczyk, H., Steiner, M., Tsudik, G., Van Herreweghen, E., Waidner, M.: Design, implementation, and deployment of the iKP secure electronic payment system. IEEE Journal on Selected Areas in Communications **18**(4) (2000) 611–627
10. Bella, G., Massacci, F., Paulson, L.: Verifying the SET purchase protocols. Journal of Automated Reasoning **36**(1) (2006) 5–37
11. Lowe, G.: A hierarchy of authentication specifications. In: CSFW'97. IEEE Computer Society Press (1997) 31–43
12. Denker, G., Millen, J.: CAPSL and CIL language design. Technical Report SRI-CSL-99-02, SRI International Computer Science Laboratory (1999)
13. Mödersheim, S.: Algebraic properties in Alice and Bob notation (extended version). Technical Report RZ3709, IBM Zurich Research Lab (2008)
14. Blanchet, B., Smyth, B., Cheval, V.: ProVerif 1.91: Automatic cryptographic protocol verifier, user manual and tutorial. (2015)
15. Bruni, A., Modersheim, S., Nielson, F., Nielson, H.R.: Set-pi: Set membership pi-calculus. In: Computer Security Foundations Symposium (CSF), IEEE (2015) 185–198
16. Bugliesi, M., Calzavara, S., Mödersheim, S., Modesti, P.: Security protocol specification and verification with AnBx. Technical Report CS-TR-1479, School of Computing Science, Newcastle University (2015)
17. Tobler, B., Hutchison, A.: Generating network security protocol implementations from formal specifications. Certification and Security in Inter-Organizational E-Service (2005) 33–54
18. Backes, M., Busenius, A., Hriţcu, C.: On the development and formalization of an extensible code generator for real life security protocols. In: NASA Formal Methods. Springer (2012) 371–387

19. Pironti, A., Pozza, D., Sisto, R.: Formally based semi-automatic implementation of an open security protocol. Journal of Systems and Software **85**(4) (2012) 835–849
20. Avalle, M., Pironti, A., Pozza, D., Sisto, R.: JavaSPI: A framework for security protocol implementation. International Journal of Secure Software Engineering **2**(4) (2011) 34–48
21. Millen, J., Muller, F.: Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International (December 2001)
22. Almousa, O., Mödersheim, S., Viganò, L.: Alice and Bob: Reconciling formal models and implementation. In Bodei, C., Ferrari, G.L., Priami, C., eds.: Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday. Volume 9465 of Lecture Notes in Computer Science. Springer International Publishing (2015) 66–85
23. FutureID Consortium: FutureID Project `http://www.futureid.eu`.
24. Basin, D., Keller, M., Radomirovic, S., Sasse, R.: Alice and Bob meet equational theories. In Martí-Oliet, N., Ölveczky, P.C., Talcott, C., eds.: Logic, Rewriting, and Concurrency. Volume 9200 of Lecture Notes in Computer Science. Springer International Publishing (2015) 160–180
25. Schmidt, B., Meier, S., Cremers, C., Basin, D.: Automated analysis of Diffie-Hellman protocols and advanced security properties. In: Computer Security Foundations Symposium (CSF), 2012 IEEE 25th, IEEE (2012) 78–94
26. Eclipse Foundation: Eclipse IDE `http://www.eclipse.org`.