Secrecy Types for a Simulatable Cryptographic Library

Peeter Laud
Dept. of Mathematics and Computer Science, Tartu University
J. Liivi 2, 50409 Tartu, Estonia
peeter.laud@ut.ee

ABSTRACT

We present a type system for checking secrecy of messages handled by protocols that use the Backes-Pfitzmann-Waidner library for cryptographic operations. A secure realization of this library exists, therefore we obtain for the first time a cryptographically sound analysis for a full language for expressing protocols, particularly handling symmetric encryption and unbounded number of sessions. The language is similar to the spi-calculus, but has a completely deterministic semantics. The type system is similar to the Abadi-Blanchet type system for asymmetric communication.

Categories and Subject Descriptors: C.2.2 [Network protocols]: Protocol verification; D.3.3 [Language Constructs and Features]: Concurrent programming structures, Coroutines

General Terms: Security, Verification.

Keywords: Simulatability, type systems.

1. INTRODUCTION

We consider the problem of analyzing the security of cryptographic protocols. An ideal methodology for protocol analysis is preferably amenable to automation, and its outcome should also be easily verifiable by a third party.

Over the years, two fruitful approaches have evolved to model the behavior of the protocols. One of them is the Dolev-Yao model [22] where the cryptographic messages are interpreted as terms in a certain free algebra; the possible operations on these messages are defined by term rewriting. For such kinds of semantics a rich body of work on automatic protocol analysis exists, see [39] for a survey. The other kind of semantics is based on computational complexity theory. There the messages are modeled as bit-strings and the attacker is allowed to perform arbitrary operations on them, as long as its consumption of resources, particularly the running time remains reasonable (polynomial in the length of secrets); Yao [47] was one of the first to explore this direction. It is closer to the real-world execution of the protocol and a proof in that model is a better guarantee for security than a proof in the Dolev-Yao model. Unfortunately, the proofs of protocol properties in the computational model tend to be more complex and there has not been much work on automating these proofs. Recently, the situation has begun to improve, though.

In this paper we define a language for cryptographic protocols, similar to the spi-calculus [4] and give it a semantics using the computational model. We then propose a type system for our language and show that if a protocol types then it preserves the secrecy of messages given to it by its users. Our semantics relies on the simulatable cryptographic library [12, 11, 9] by Backes, Pfitzmann and Waidner. The library comes with two semantics, one of them resembles, although is not quite the same as the Dolev-Yao model. The other semantics — the "real implementation" of its functionality" — is in the computational model. A theorem states that any security property satisfied by the formal semantics is also satisfied by the computational one. Hence the proofs may use the Dolev-Yao-like semantics. As far as we know, this type system is the first case where an analysis method amenable to automation is used to analyse the BPW library.

The expressiveness of our calculus is similar to the spi-calculus; it contains nonce and key generations, symmetric and asymmetric encryptions and decryptions, equality checks, communication primitives, unbounded parallelism. Communication occurs between protocol *participants* and also between a participant and its *user*. Here the participant denotes a Turing machine that executes the functionality directly related to the protocol. A protocol user is a Turing machine or a set of Turing machines that uses the protocol functionality, most often through API calls, to achieve its own ends. In existing protocol analyses, where some process calculus is involved, there are usually no specific methods for communicating with the user. Instead, it is stated that some variables have been assigned values before the protocol is started, and giving received values back to the user is not handled at all. We have chosen to not adopt this convention here (although it would have been possible). It would not have fit well into the simulatability framework — we would not have obtained the full benefit from composability.

The semantics of our calculus is entirely deterministic, as prescribed by the simulatability framework. In contrast with the spi-calculus, the scheduling of threads of participants is explicit, and it uniquely determines the thread that handles each incoming messages. See Sec. 4 for details. Our type system is inspired by the type system of Abadi and Blanchet [2] (they work in the Dolev-Yao model). In this type system, the type of a variable reflects its intended use. Values with a 'public' type may be passed to the attacker. Encryption keys have a 'key' type parameterized by the type of plaintexts that may be encrypted. Communication channels similarly have types parameterized by the type of their messages. The type system can also indicate that two variables cannot be equal, when they have nonintersecting types.

CCS'05, November 7–11, 2005, Alexandria, Virginia, USA.

In this paper we first review the related work in Sec. 2 and give an overview of the BPW library in Sec. 3. As next we present our process calculus and give its semantics and a definition for confidentiality of inputs in Sec. 4. The security definition is not new, it has been proposed by Backes and Pfitzmann [10]. In Sec. 5 we present the type system and in Sec. 6 give a sketch of the correctness proof. The subject reduction theorem that we employ may be of independent interest.

The full paper [34] gives extra details for the material presented in Sections 4–6.

2. RELATED WORK

Reconciling the two main approaches for modeling and analyzing cryptographic protocols has received quite a lot of attention recently. This work can be characterized as more ambitious than mere provably correct analysis of cryptographic protocols in the computational model. Instead, one aims to show that the properties (from some large and interesting enough class) of the protocol in the Dolev-Yao model carry directly over to the computational model, at least if the protocol satisfies some extra constraints.

This line of work started with the relating of formal and computational symmetric encryption by Abadi and Rogaway [6]; their work considered only passive adversaries. The relationship of formal and computational encryption under attacks from passive adversaries was further investigated by Abadi and Jürjens [5] and by Micciancio and Warinschi [41]. Micciancio and Panjwani [40] considered the case where the adversary adaptively chose the formal messages to be computationally interpreted, the necessary security of the encryption scheme still corresponded to security against passive adversaries. In contrast, we have given a static program analysis [32] and a type system [35] that work directly in the computational model, handling programs containing symmetric encryption.

Reconciliation approaches taking into account also active adversaries have mostly considered asymmetric primitives and/or integrity properties (i.e. properties of execution traces). Guttman et al. [28] were one of the first to consider authentication in the presence of active adversaries in two models. Their approach was different from the later ones in that the security definitions in the computational model were not complexity-theoretical, but information-theoretical, so the obtained security guarantee was stronger than usual. The cost for this added strength was the length of the shared secrets. They also pioneered the technique of translating a protocol run in the computational model, after it had finished, to a run in the formal model and showing that if that run would not have been possible in the formal model then something which should happen only with a negligible probability must have happened in the run in the computational model. The approach was developed further by Micciancio and Warinschi [42] who related the formal and computational traces for protocols using symmetric encryption. Cortier and Warinschi [21] show that there exist automatic analyses for the formal model that carry directly over to the computational model. Janvier et al. [30] extend [42] by allowing secret keys to be transferred (in a limited way) and in [31] by considering more cryptographic primitives.

Translating confidentiality properties (i.e. equivalences over execution traces) between formal and computational models has received seemingly less attention. Secrecy in the computational model is usually defined as a confidentiality property while in the formal model it may also be a confidentiality property (process equivalence [4, 15]), but more commonly is an integrity property (the adversary is unable to output the message that we want to protect). Herzog et al. [29] show that if the asymmetric encryption is plaintext-aware [14] then the computational adversary cannot construct the interpretation of any formal message that the formal adversary cannot construct. Note that here the property in the computational model is still an integrity property so the adversary may learn *partial* information about the secret messages. Cortier and Warinschi [21] show that a certain integrity property in the formal model implies the secrecy of nonces generated by protocol parties in the computational model; this result is also applicable to [30, 31]. The confidentiality property is similar to the find-then-guess secrecy property [24, 13] but seems to be weaker in that the adversary can choose less parameters. Also, it only applies to nonces generated during the protocol run. A yet different approach, capable of verifying the independence of secret payloads but limited to a bounded number of protocol runs, is ours [33] where a protocol is transformed in an automated way, such that the view of the adversary does not change distinguishably. The transformation is based on the security definitions of the cryptographic primitives which demand the indistinguishability of certain two oracles — parts of the protocol that behave as the "real" oracle may be replaced by the "ideal" oracle. If one can transform out all syntactic accesses to the secret payloads then the payloads are secure — the view of the adversary is independent of them.

We already mentioned a different approach in the Introduction — the faithful abstraction of cryptographic primitives (in the computational model) — and the simulatable library [12, 11, 9]. The realizations of some protocols have been proved correct with the help of that library [8, 7], these hand-crafted proofs argue in terms of the model presented by the library. Another line of work in the faithful abstraction is the work on universal composability by Canetti [16]. Faithful abstractions are given for several cryptographic primitives in this model [17, 19, 20], including asymmetric encryption and digital signatures. This time they are not parts of a monolithic library, but each primitive is abstracted by a different machine; such design should be considered superior. The price of the better design are the increased capabilities of the adversary for attacking these abstractions, compared to the BPW library. Still, Canetti and Herzog [18] have defined the abstract functionality for certified public key encryption which allows them to relate the integrity properties satisfied by protocols (with bounded number of runs) using only asymmetric encryption in formal and computational models.

Yet another approach is that of Lincoln, Mitchell et al. [36, 37]; they have given a "computational" semantics for a variant of π-calculus where probabilistic choice replaces non-determinism everywhere. The definitions of confidentiality and integrity properties from the computational model easily carry over to this setting. They have also devised a formal proof system [46] for this calculus, but it does not seem to be amenable for automatic deduction. Somewhat related is the result by Zunino and Degano [48]. They have given the Dolev-Yao adversary the ability to break the encryption (find the key from a ciphertext), but only with negligible probability. They show that this ability does not strengthen the adversary.

Let us also mention some of the work in the area of type systems for cryptographic protocol analysis. The first type system of this kind was proposed by Abadi [1], it could be used for verifying the secrecy of payloads or nonces in the protocols using only symmetric encryption. This type system, as well as all the others that we describe work in the Dolev-Yao model. The type system was extended to cope with asymmetric encryption by Abadi and Blanchet [2]; this type system is the closest to the type system presented in current paper. Abadi and Blanchet [3] further generalized this type system to handle *generic* cryptographic primitives. The type system of Abadi has also been extended by Gordon and Jeffrey [25, 26, 27] to check for integrity properties.

3. OVERVIEW OF THE CRYPTOGRAPHIC LIBRARY

Let us give an overview of the simulatable cryptographic library [12], as well as simulatability [44] itself. A *system* is a set of *structures*. A structure *Str* is a collection of probabilistic interactive Turing machines. These machines have named input and output ports; an input and output port with the same name make up a channel between corresponding machines. These channels are secure; authentic or insecure channels can be modeled by two secure channels with the adversary having a port on one or both of them. Some of the ports of the structure have no complementary ports in the structure; a certain subset S of them are called the *free ports*, these ports are used to access the functionality of the structure. The rest of unconnected ports are meant for the adversary, they model the necessary imperfections of the system. A *configuration* consists of a structure, a machine H connecting only to the free ports of the structure and modeling the intended user of the system, and an adversarial machine A that must connect to all remaining unconnected ports. There may also be connections between H and A. The *view* of the user H is the distribution of the sequence of messages on the ports of H.

Let Str and Str' be two structures with the same set of free ports S. We say that Str is *at least as secure as* Str' (denoted $Str \geq Str'$; that relation is called *simulatability*) if for each possible H and A there exists an adversary S, such that the view of H in the configuration with Str and A is computationally indistinguishable [23] from its view in the configuration with Str' and S. In other words, anything that can happen to H using the functionality provided by Str can also happen to it if it uses the functionality of Str'. A system Sys is *at least as secure as* a system Sys' if each structure of Sys is at least as secure as some structure of Sys'.

Often we call Sys the *real system* and Sys' the *ideal system*. In this case the structures in Sys' have to consist of a single, deterministic Turing machine that specifies the intended functionality. The structures in Sys must have a Turing machine for each possible location (if the functionality is for **n** parties then there must be **n** machines, one for each party) and they should use mostly insecure channels for communication (the secure or authentic channels may be used for initial exchange of secrets). A particular ideal system is the simulatable cryptographic library. It contains a structure for each possible number **n** of honest parties, this structure is made up of a single machine $\mathcal{TH}_{\mathbf{n}}$ that has ports $\mathsf{in}_{\mathsf{u}_i}?$ and $\mathsf{out}_{\mathsf{u}_i}!$ for communicating with the i-th party and ports $\mathsf{in}_{\mathsf{a}}?$ and $\mathsf{out}_{\mathsf{a}}!$ for communicating with the adversary. The i-th party sends API calls to $\mathcal{TH}_{\mathbf{n}}$ on the channel $\mathsf{in}_{\mathsf{u}_i}$ and receives answers on $\mathsf{out}_{\mathsf{u}_i}$. Also, when a message is sent to the i-th party through $\mathcal{TH}_{\mathbf{n}}$ then it also reaches the party through the channel $\mathsf{out}_{\mathsf{u}_i}$. In a corresponding real structure, there is a machine M_i for each party, these machines use real cryptographic primitives [24, 45] for securing the messages.

The main component of $\mathcal{TH}_{\mathbf{n}}$ is the database of terms. The database records for each term its subterms and the parties that "know" that term. The parties and the adversary access the terms through handles that by itself are just consecutive integers; hence they contain no information about the contents of the database. The sending of messages from one party to another therefore has to be done with the help of $\mathcal{TH}_{\mathbf{n}}$ in translating the handles. The machine $\mathcal{TH}_{\mathbf{n}}$ offers to the honest parties and the adversary the functionality to store (command store) "raw data" (i.e. arbitrary bit-strings) inside $\mathcal{TH}_{\mathbf{n}}$ and to retrieve (command retrieve) them (by its handle); to build up lists of terms (command list) and to take components of lists (command list_proj); to construct new nonces (gen_nonce), public encryption and private decryption keys (gen_enc_keypair), and symmetric encryption/decryption keys (gen_symenc_key); to encrypt (encrypt and sym_encrypt) lists and to decrypt (decrypt and sym_decrypt) the resulting ciphertexts; to find either the type (get_type) or the length (get_len) of a term. Two terms may be compared for equality, this is done by comparing their handles. A party can also send a term, which must be a list, to another party either over a secure, authentic or insecure channel. A term sent over authentic channel is also sent to the adversary; a term sent over insecure channel is sent only to the adversary. The adversary can send messages to honest parties over insecure channels. The adversary can also insert "garbage" terms (command adv_garbage) to the database. As a departure from the Dolev-Yao model, one can determine the public key from an asymmetric ciphertext (pk_of_enc) or the identity of the key (not the key itself) from a symmetric ciphertext, so the encryption, as modeled by $\mathcal{TH}_{\mathbf{n}}$, is key-revealing (type-3 in terms of [6]). Also, the adversary may create invalid ciphertexts (both symmetric (adv_unknown_symenc) and asymmetric (adv_invalid_ciph)). For the symmetric encryption, the adversary may later also fix (adv_fix_symenc_content) the encryption key and the plaintext, as long as it has a handle to both of them.

The cryptographic library also contains digital signatures and symmetric authenticators, but as we are not going to handle them in the current paper (although the necessary extensions should not be difficult), we will not describe them here.

Let us also describe the scheduling of machines. Only one machine runs at a time, and it processes a single input from a single input port. Besides input and output ports the channels also have *clock ports*, denoted $c^{\triangleleft}!$ for a channel c. The messages currently on a channel c are stored in a buffer, messages written to $c!$ are appended to this buffer. When a machine M runs, it may write an integer to exactly one of the clock ports it has. If it does so (writes i to $c^{\triangleleft}!$) and the buffer of c has at least i messages waiting then the i-th message is removed from the buffer and passed to the machine M' having the port $c?$, the machine M' will run next. Otherwise the adversary will run next. The machine $\mathcal{TH}_{\mathbf{n}}$ has clock ports for channels $\mathsf{out}_{\mathsf{u}_i}$ and $\mathsf{out}_{\mathsf{a}}$. Whenever it writes something to these channels, it follows it by 1 on the corresponding clock port. The parties are supposed to have the ports $\mathsf{in}_{\mathsf{u}_i}{}^{\triangleleft}!$, this allows $\mathcal{TH}_{\mathbf{n}}$ to be used as a subroutine when operating with terms. Note, however, that when a party commands $\mathcal{TH}_{\mathbf{n}}$ to send a message to someone then the control is not returned to that party.

There is a caveat related to the simulatability of $\mathcal{TH}_{\mathbf{n}}$ by the real structure $(\mathsf{M}_1, \ldots, \mathsf{M}_{\mathbf{n}})$ — the user H may not be completely arbitrary. For the simulatability to hold, a condition regarding the usage of symmetric keys must be true. In [9] the condition, called NoComm, states that H must ensure that a symmetric key k either becomes known to the adversary before it is used, or that whenever a term containing k is sent over a channel that the adversary can read, the subterm containing k must be encrypted by some other key k' for which the adversary does not know the decryption key. Moreover, to enable the use of hybrid argument [23, Chap. 3] in the simulatability proof, it must be known statically which symmetric key encrypts which one. In [9] this is formalized by ordering the keys by their first use as encryption keys and requiring that a key may only be encrypted under keys whose first use was earlier. We formalize this by fixing the "order" — a positive integer — of a symmetric key when it is created and require that a key may only be encrypted by keys of higher order. The order is fixed by giving it as an argument to the command gen_symmetric_key of $\mathcal{TH}_{\mathbf{n}}$. The machine $\mathcal{TH}_{\mathbf{n}}$ is supposed to just discard that argument, but its presence in the view of $\mathcal{TH}_{\mathbf{n}}$ is necessary for the simulatability proof. More precisely, the order given to symmetric keys allows us to give an order to each term in the database — the order of lists is the maximum order of their components, and the order of terms that are neither symmetric keys nor lists is 0. We then require the order of the symmetric key to be higher than the order of the plaintext.

4. THE PROCESS CALCULUS

The functionality of the protocol is provided to the users of the protocol in the form of a structure $C_{\mathbf{n}}$ (for $\mathbf{n}$ users). The structure C consists of the machines M_i ($1 \leq i \leq n$) realizing the cryptographic primitives for the i-th party, and of the machines P_i that execute the instructions (of the i-th party) that make up the actual protocol. The process calculus is used to program the participants P_i of the protocol. The machine P_i must have the ports $\mathsf{in}_{\mathsf{u}_i}!$, $\mathsf{out}_{\mathsf{u}_i}?$ and $\mathsf{in}_{\mathsf{u}_i}{}^{\triangleleft}!$ to communicate with the machine M_i. We let the communication with the i-th user to go over the channels $\mathsf{pin}_{\mathsf{u}_i}$ and $\mathsf{pout}_{\mathsf{u}_i}$ in the form of API calls, i.e. these channels are secure. The machine P_i has the ports $\mathsf{pin}_{\mathsf{u}_i}?$, $\mathsf{pout}_{\mathsf{u}_i}!$ and $\mathsf{pout}_{\mathsf{u}_i}{}^{\triangleleft}!$, all messages from P_i to the user are scheduled immediately.

The simulatability result for the cryptographic library [12, 9] states that $C_{\mathbf{n}}$ is at least as secure as the structure $C'_{\mathbf{n}}$ consisting of $\mathcal{TH}_{\mathbf{n}}$ and $\mathsf{P}_1, \ldots, \mathsf{P}_{\mathbf{n}}$. Hence we analyse the structure $C'_{\mathbf{n}}$ in the rest of the paper.

The process calculus is designed to be quite similar to the spi-calculus [4]. As mentioned in the introduction, we make explicit the selection of a thread for each incoming message. In some approaches where complexity-theoretic security definitions are used and therefore nondeterminism cannot be employed, the adversary is allowed to choose which thread handles the received message. We think that this is not the right design decision because the information that is available to the honest party, but not to the adversary, should be able to influence which thread handles the incoming message. I.e. the decision should be made by the honest party itself. As a minimal addition supporting that decision-making we add the "invalid input" command $\mathcal{II}$ to the calculus. When a thread executes the command $\mathcal{II}$, its computations since the last input are discarded and the received message is passed to the next thread.

The task of the semantics of the process calculus is to describe how the state of $C'_{\mathbf{n}}$ evolves in response to the inputs it may receive from the users (over the ports $\mathsf{pin}_{\mathsf{u}_i}?$) and from the adversary (over the port $\mathsf{in}_{\mathsf{a}}?$). A state $\mathbf{C}$ of $C'_{\mathbf{n}}$ consists of the following elements:

- The contents $\mathcal{O}$ of the database of $\mathcal{TH}_{\mathbf{n}}$.
- The states $\mathbf{S}_i$ of the machines P_i.
- The multisets of messages $\mathbf{L}_c^{i\rightarrow j}$ on secure and authentic channels that have not yet been delivered to their recipients (the adversary schedules these channels as well). Here $1 \leq i, j \leq n$ and $c \in \{\mathsf{s}, \mathsf{a}\}$.

The possible actions are various inputs from the adversary and the users (external actions), and the internal actions. The state of each P_i contains a bit showing whether P_i is currently *active* or not; at most one P_i can be active at any time. If there is an active P_i then the next executed action is internal, otherwise it is external. Of course, we have to make sure that the semantics, as we define it, is actually executable by $C'_{\mathbf{n}}$.

In the following we define the expressions and the processes and explain how an expression is evaluated and how a process evolves. Afterwards we explain how the states $\mathbf{S}_i$, which consists mainly of a list of processes, and the state $\mathbf{C}$ evolve. The omitted details can be found in [34].

Let $\mathbf{Var}$ be the set of variables; it is a countable set. The values of the variables will be integers, they may be interpreted either as bit-strings (i.e. the "raw data" exchanged with the protocol user) or term handles (the database of $\mathcal{TH}_{\mathbf{n}}$ maps them to terms). In the following we denote the elements of $\mathbf{Var}$ by x (with subscripts). The *values* v, *expressions* e, *processes* P (corresponding to active threads) and *input processes* I (corresponding to inactive threads) are defined in Fig. 1 and Fig. 3. Here n and i are integers. The expressions that are not underlined directly correspond to the commands of $\mathcal{TH}_{\mathbf{n}}$ of the same name. The other expressions correspond to certain sequences of commands, given in Fig. 2. The reason for their introduction is to hide that $\mathcal{TH}_{\mathbf{n}}$ only allows lists to be plaintexts, and that $\mathsf{gen\_enc\_keypair}$ returns handles for both the secret and the public key. Hence $\underline{\mathsf{keypair}}$ actually returns the secret key.

$$v ::= n \mid \bot$$

$$
\begin{array}{lllllll}
e & ::= & n & \mid & \underline{\mathsf{keypair}} & \mid & \mathsf{store}(e) \\
 & \mid & x & \mid & \underline{\mathsf{pubkey}}(e) & \mid & \underline{\mathsf{pubenc}}(e_{\mathrm{k}}, e_{\mathrm{t}}) \\
 & \mid & \bot & \mid & \mathsf{gen\_symenc\_key}(i) & \mid & \underline{\mathsf{pubdec}}(e_{\mathrm{k}}, e_{\mathrm{t}}) \\
 & & & \mid & \mathsf{retrieve}(e) & \mid & \mathsf{list}(e_1, \ldots, e_k) \\
 & & & \mid & \underline{\mathsf{privenc}}(e_{\mathrm{k}}, e_{\mathrm{t}}) & \mid & \mathsf{list\_proj}(e, i) \\
 & & & \mid & \underline{\mathsf{privdec}}(e_{\mathrm{k}}, e_{\mathrm{t}}) & \mid & \mathsf{gen\_nonce}
\end{array}
$$

Figure 1: Values and arithmetic expressions

$$
\begin{aligned}
\underline{\mathsf{keypair}} &\equiv \pi_1(\mathsf{gen\_enc\_keypair}()) \\
\underline{\mathsf{pubkey}}(e) &\equiv e + 1 \\
\underline{\mathsf{pubenc}}(e_{\mathrm{k}}, e_{\mathrm{t}}) &\equiv \mathsf{encrypt}(e_{\mathrm{k}}, \mathsf{list}(e_{\mathrm{t}})) \\
\underline{\mathsf{pubdec}}(e_{\mathrm{k}}, e_{\mathrm{t}}) &\equiv \mathsf{list\_proj}(\mathsf{decrypt}(e_{\mathrm{k}}, e_{\mathrm{t}}), 1) \\
\underline{\mathsf{privenc}}(e_{\mathrm{k}}, e_{\mathrm{t}}) &\equiv \mathsf{sym\_encrypt}(e_{\mathrm{k}}, \mathsf{list}(e_{\mathrm{t}})) \\
\underline{\mathsf{privdec}}(e_{\mathrm{k}}, e_{\mathrm{t}}) &\equiv \mathsf{list\_proj}(\mathsf{sym\_decrypt}(e_{\mathrm{k}}, e_{\mathrm{t}}), 1)
\end{aligned}
$$

Figure 2: Derived expressions

$$
\begin{array}{rcl}
\mathit{SIP} & ::= & \mathbf{receive}_c(x_{\mathrm{p}}, x) \\
\mathit{IP} & ::= & \mathit{SIP} \quad \mid \quad !\mathit{SIP} \\
I & ::= & \mathit{IP}.P \\
I^* & ::= & \mathbf{0} \qquad \mid \quad I \mid I^* \\
P & ::= & I^* \qquad \mid \quad \mathcal{II} \qquad \mid \quad \mathbf{send}_c(e_{\mathrm{p}}, e).I^* \\
 & \mid & \mathbf{let}\ x := e\ \mathbf{in}\ P\ \mathbf{else}\ P' \\
 & \mid & \mathbf{if}\ e = e'\ \mathbf{then}\ P\ \mathbf{else}\ P'
\end{array}
$$

Figure 3: The process calculus

The participant P_i evaluates a closed (i.e. without free variables) expression e inductively over the expression structure, by sending the respective commands to $\mathcal{TH}_{\mathbf{n}}$ and getting back their values. If the evaluation of any subexpression fails (i.e. returns $\bot$; denoted $\downarrow$ in [12]) then the value of the whole expression is $\bot$, too. In the full paper we give a precise definition of the relation $e\ {}^{\mathcal{O}}\!\Downarrow^{i}_{\mathcal{O}'}\ v$ meaning that e, if evaluated by P_i with the contents $\mathcal{O}$ of the database of $\mathcal{TH}_{\mathbf{n}}$, results in v and the contents of the database becomes $\mathcal{O}'$.

As next, let us describe the execution of a process P by an active P_i. Both I^* and $\mathcal{II}$ denote deactivated processes, but they are handled differently by P_i. Let $\mathbf{Chan} \subseteq \mathbb{N}$ be the set of *abstract channels*. An abstract channel is used to group messages sent between protocol participants, as well as between the protocol user and participant (although the abstract channel does not alone determine the sender and the receiver of a message). The set $\mathbf{Chan}$ is partitioned into four parts, denoted $\mathbf{Chan}_c$, where $c \in \{\mathsf{s}, \mathsf{a}, \mathsf{i}, \mathsf{u}\}$. If a message is sent on an abstract channel from $\mathbf{Chan}_{\mathsf{s}}$ [resp. $\mathbf{Chan}_{\mathsf{a}}$,

$\mathbf{Chan}_\mathsf{i}$] then it means that the message travels between protocol participants over a secure (resp. authentic, insecure) channel. If a message is sent on an abstract channel from $\mathbf{Chan}_\mathsf{u}$ then it travels between the protocol user and the protocol participant (i.e. over one of the channels $\mathsf{pin}_{\mathsf{u}_i}$ or $\mathsf{pout}_{\mathsf{u}_i}$). Let us assume that $|\mathbf{Chan}_\mathsf{u}| = 1$, because the type system has to handle all inputs from the user identically anyway. The process $\mathbf{send}_c(e_\mathrm{p}, e).I^*$ evaluates e as v and e_p as v_p. If neither of them is $\perp$ then it sends v to the participant v_p "over the abstract channel c" (unless $c \in \mathbf{Chan}_\mathsf{u}$, in this case v is simply handed over to the user). This means that $\mathsf{list}(\mathsf{store}(c), v)$ is sent to v_p over the secure / authentic / insecure channel, depending on c. The process then becomes I^*. The process $\mathbf{let}\ x := e\ \mathbf{in}\ P\ \mathbf{else}\ P'$ evaluates e as v. If $v \neq \perp$ then it becomes $P_{x\leftarrow v}$ otherwise it becomes P'. I.e. here x is bound in P but not in P'. The process $\mathbf{if}\ e = e'\ \mathbf{then}\ P\ \mathbf{else}\ P'$ evaluates both e and e', compares them and becomes either P or P', depending on the result.

The state $\mathbf{S}_i$ of P_i can be inactive or active. An inactive $\mathbf{S}_i$ is just a list of closed input processes. It is activated by a message arriving at the port $\mathsf{out}_{\mathsf{u}_i}?$ or $\mathsf{pin}_{\mathsf{u}_i}?$. An active $\mathbf{S}_i$ contains that message; let c_v be the abstract channel over which it arrived, u its (apparent) sender, and m its contents. If the message read from $\mathsf{out}_{\mathsf{u}_i}?$ is not of the shape created by a $\mathbf{send}$-command then $\mathbf{S}_i$ is immediately deactivated again. An active state also contains the list of input processes I^*_{pre} that have already processed the message and have rejected it, the currently running process (may be missing) and the list of input processes I^*_{post} that have not yet been run. Upon activation, I^*_{pre} is empty and I^*_{post} equals the inactive state. If the currently running process is missing then the step of $\mathbf{S}_i$ consists of taking the first input process $(!)\mathbf{receive}_c(x_\mathrm{p}, x).P$ of I^*_{post} (if it is empty then $\mathbf{S}_i$ is deactivated with I^*_{pre} as its state) and comparing c to c_v. If $c = c_v$ then $P_{x\leftarrow m, x_\mathrm{p}\leftarrow u}$ becomes the currently running process, otherwise this input process is moved to the end of I^*_{pre}. If the currently running process P is present and active then a step of $\mathbf{S}_i$ is just a step of P. If P is $\mathcal{II}$ then it is discarded and the first input process in I^*_{post} is moved to the end of I^*_{pre}. If P is I^* then $\mathbf{S}_i$ is deactivated and its new state is the concatenation of (1) I^*_{pre}, (2) I^*, (3) if $head(I^*_{\mathsf{post}})$ is replicated then I^*_{post} else $tail(I^*_{\mathsf{post}})$.

The state $\mathbf{C}$ of $C'_\mathbf{n}$ may evolve in the following ways. If the state $\mathbf{S}_i$ of a P_i is active then a step of $\mathbf{C}$ is just a step of $\mathbf{S}_i$. If $\mathbf{S}_i$ is deactivated then the sent message is delivered to the adversary (if it was on an authentic or insecure channel), put to the right buffer $\mathbf{L}_c^{i\to j}$ (if it was on a secure or authentic channel) or sent to the user (if it was on an abstract channel from $\mathbf{Chan}_\mathsf{u}$). If the states of all P_i are inactive then $C'_\mathbf{n}$ is expecting a message from outside, there are two kinds of such messages. First are the basic or local adversary commands [12] executed by the adversary, these cause the contents $\mathcal{O}$ of the database of $\mathcal{TH}_\mathbf{n}$ to be changed and a handle and the control to be returned to the adversary; no participant is activated. Second are the incoming messages to protocol participants, initiated either by the adversary (who also schedules secure and authentic channels [12]) or the users. This causes one of the participants P_i to be activated with the incoming message, and the message removed from the right buffer $\mathbf{L}_c^{i\to j}$ (if it arrived over a secure or authentic channel).

The precise definition of the evolution of the states of $C'_\mathbf{n}$ is given in the full paper [34]. There we define a relation $^{\mathcal{O}}\xrightarrow{i}{}^{\mathcal{O}'}$ for processes and for participant states, meaning that a process [participant state] at the left hand side, when executed by P_i with the contents of the database of $\mathcal{TH}_\mathbf{n}$ as $\mathcal{O}$, becomes the process [participant state] at the right hand side and the contents of the database of $\mathcal{TH}_\mathbf{n}$ becomes $\mathcal{O}'$. We also define a relation $\to$ for the states of $C'_\mathbf{n}$.

Security

We now define our notion of secrecy. We consider as secret all data that the protocol user(s) pass to the protocol over the channels $\mathsf{pin}_{\mathsf{u}_i}$. The actual secrecy definition has to be stated for the *real* system, i.e. we cannot refer to the internal state of $\mathcal{TH}$ in that definition.

We use the definition of payload secrecy given by Backes and Pfitzmann [10]. To state that definition we first rename the ports $\mathsf{pin}_{\mathsf{u}_i}?$, $\mathsf{pout}_{\mathsf{u}_i}!$ and $\mathsf{pout}_{\mathsf{u}_i}{}^{\triangleleft}!$ of the machine P_i; let the renamed ports be $\overline{\mathsf{pin}_{\mathsf{u}_i}}?$, $\overline{\mathsf{pout}_{\mathsf{u}_i}}!$ and $\overline{\mathsf{pout}_{\mathsf{u}_i}}{}^{\triangleleft}!$. Let the machines R and F both have ports $\mathsf{pin}_{\mathsf{u}_i}?$, $\mathsf{pout}_{\mathsf{u}_i}!$, $\mathsf{pout}_{\mathsf{u}_i}{}^{\triangleleft}!$, $\overline{\mathsf{pin}_{\mathsf{u}_i}}!$, $\overline{\mathsf{pin}_{\mathsf{u}_i}}{}^{\triangleleft}!$ and $\overline{\mathsf{pout}_{\mathsf{u}_i}}?$ for all $i \in \{1, \ldots, \mathbf{n}\}$, i.e. either R or F can be placed between the user(s) H of the protocol and the machines P_i (with renamed ports). The machine F operates by just forwarding every message it receives on $\mathsf{pin}_{\mathsf{u}_i}?$ [resp. $\overline{\mathsf{pout}_{\mathsf{u}_i}}?$] to $\overline{\mathsf{pin}_{\mathsf{u}_i}}!$ [resp. $\mathsf{pout}_{\mathsf{u}_i}!$] and clocking the output channel. Hence the operation of the original system (with unrenamed ports of P_i) is identical to the operation of the modified system (with renamed ports of P_i) when F is placed between the user(s) and the machines P_i. The machine R also forwards the messages between the user(s) and the machines P_i, but it additionally scrambles them. It keeps a dictionary T — an initially empty set of pairs of bit-strings — for that reason. On input x on $\mathsf{pin}_{\mathsf{u}_i}?$, the machine R checks whether $(x, y) \in T$ for some y. If not, then it generates a random y of the same length as x and not yet occurring as the second component of the pairs in T; the pair (x, y) is then added to T. The bit-string y is then output on $\overline{\mathsf{pin}_{\mathsf{u}_i}}!$. On input y on $\overline{\mathsf{pout}_{\mathsf{u}_i}}?$, the machine R checks whether $(x, y) \in T$ for some x. If yes, x is output on $\mathsf{pout}_{\mathsf{u}_i}!$, otherwise y is output.

Backes and Pfitzmann [10] define the payload to be secure if for all possible adversaries the view of the user, where R is placed between the user and the machines P_i, is computationally indistinguishable from its view, where F is placed between the user and the machines P_i. I.e. the user and the adversary jointly are unable to determine whether the communication between the user and the machines P_i is scrambled or not. It is also shown that payload security is preserved under simulatability.

Backes and Pfitzmann [10] have shown that the following conditions suffice for payload secrecy for the configuration C'_n.

(I) the bit-strings that the machines P_i receive from the ports $\mathsf{pin}_{\mathsf{u}_i}?$ do not affect the control flow of P_i, i.e. this data is not used in the $\mathbf{if}$-statements;

(II) the machines P_i may pass the bit-strings received from the user to the cryptographic library only in store-commands;

(III) the terms resulting from these store-commands will not become available to the adversary, i.e. the adversary does not get handles for these terms.

So our type system must ensure these three properties. As we mentioned before, it must also ensure that

(IV) symmetric keys of order i only encrypt terms of order less than i (note that symmetric keys created by the adversary have no order and are thereby not restricted by this condition);

(V) if a symmetric key unknown to the adversary (i.e. the adversary does not have a handle to it) is used for encryption then this key will never become known to the adversary.

Example

We revisit the example of Needham-Schroeder-Lowe public-key protocol by Abadi and Blanchet [2]. In informal syntax, the protocol is the following:

$$\begin{array}{ll} A \longrightarrow B: & \{s_A, A\}_{k_B} \\ B \longrightarrow A: & \{s_A, s_B, B\}_{k_A} \\ A \longrightarrow B: & \{s_B\}_{k_B} \ . \end{array}$$

Here k_A and k_B are the public keys of A and B; and s_A and s_B are fresh nonces that become shared secrets as the result of the protocol. To demonstrate that s_B does not become known to the adversary, we add the fourth message $B \longrightarrow A : \{s\}_{s_B}$ where B sends to A a secret message s (received from the user of the protocol) *symmetrically* encrypted with s_B and let A return the received message to the user.

The protocol starts by A and B generating new asymmetric key pairs and sending the public part to each other over an authentic channel. The attack found by Lowe [38] against the original Needham-Schroeder protocol [43] is a man-in-the-middle attack where A is first tricked to execute the protocol with adversarially controlled C who then impersonates A in a run with B. To model that A can execute the protocol not only with B we let A also receive public keys other than k_B. As these keys come from the adversary they arrive over insecure channels but we do not let A differentiate between k_B and other keys.

Let $c_1, c_2 \in \mathbf{Chan}_\mathsf{a}$, $c_i \in \mathbf{Chan}_\mathsf{i}$ and $c_u \in \mathbf{Chan}_\mathsf{u}$. We use the channel c_1 to communicate the public key of A to B and the channel c_2 to communicate the public key of B to A. The need for two authentic channels is caused by our type-system's need to distinguish between these two keys. The processes for A and B are given in Fig. 4. Here we let $\mathbf{let}^{P'}\ e\ \mathbf{in}\ P$ denote the process $\mathbf{let}\ e\ \mathbf{in}\ P\ \mathbf{else}\ P'$ and let $\mathbf{if}^{P'}\ b\ \mathbf{then}\ P$ denote $\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ P'$.

In Fig. 4, A starts by generating a new asymmetric key pair and sending the public part to B. It then expects public keys of other parties to be sent to it over authentic (the key of B) or insecure channels. The process A' handles all sessions with a fixed other party X. First it expects the public key of X as the sign to start a new session with it. The rest of A' should be clear. At the end of the protocol run, A' returns the received message s to the user. The process B starts by receiving the secret message s from the user and storing it in the database of $\mathcal{TH}_\mathbf{n}$. It then generates a new asymmetric key pair, sends the public part to A and gets the public key of A. Afterwards it can participate in any number of protocol sessions. The process B assumes that its partner in these sessions is A as it encrypts the outgoing messages with k_A.

All public keys are sent to A in a similar way — both new public keys (of principals that A did not know before; these keys are received outside of the process A') and the public keys of parties with whom to start a new protocol run (received inside A'). The order of considering threads and inserting new threads to the list of threads in the state of P_A ensures that the first time when A receives a public key this is interpreted as the public key of a new party (and a new process A' is spawned) and when the same public key is received again then A attempts to start a new protocol run with this party.

5. THE TYPE SYSTEM

A *typing* Γ assigns types to the free variables of the processes and to the abstract channels used by the protocol. The types of channels reflect the types of messages passed over them. In the following we define the set of types and also state, when a process (or input process) *types* with respect to a typing. We say that a protocol *types* if there exists a typing such that all the input processes in the initial states of the machines P_i type with respect to it.

$A'(X, k_X)$ is

$$\begin{array}{l}
!\mathbf{receive}_{c_i}[\mathit{fc1}](mc).\mathbf{if}^{\mathcal{JJ}}\ mc = k_X\ \mathbf{then} \\
\quad \mathbf{let}^{\mathbf{0}}\ s_A := \mathsf{gen_nonce}\ \mathbf{in} \\
\quad \mathbf{send}_{c_i}(X, \underline{\mathsf{pubenc}}(k_X, \mathsf{list}(s_A, k_A))). \\
\mathbf{receive}_{c_i}(y_{X2}, m_2).\mathbf{let}^{\mathcal{JJ}}\ l_2 := \underline{\mathsf{pubdec}}(k_A^{-1}, m_2)\ \mathbf{in} \\
\quad \mathbf{let}^{\mathcal{JJ}}\ s_{A2} := \mathsf{list_proj}(l_2, 1)\ \mathbf{in}\ \mathbf{let}^{\mathcal{JJ}}\ k'_{AX} := \mathsf{list_proj}(l_2, 2)\ \mathbf{in} \\
\quad \mathbf{let}^{\mathcal{JJ}}\ k_{X2} := \mathsf{list_proj}(l_2, 3)\ \mathbf{in}\ \mathbf{if}^{\mathcal{JJ}}\ k_X = k_{X2}\ \mathbf{then} \\
\quad \mathbf{if}^{\mathcal{JJ}}\ s_{A2} = s_A\ \mathbf{then}\ \mathbf{send}_{c_i}(X, \underline{\mathsf{pubenc}}(k_{X2}, k'_{AX})). \\
\mathbf{receive}_{c_i}(y_{X4}, m_4).\mathbf{let}^{\mathcal{JJ}}\ s' := \underline{\mathsf{privdec}}(k'_{AX}, m_4)\ \mathbf{in} \\
\quad \mathbf{send}_{c_u}(_, \mathsf{retrieve}(s')).\mathbf{0}
\end{array}$$

A is

$$\begin{array}{l}
\mathbf{let}^{\mathbf{0}}\ k_A^{-1} := \underline{\mathsf{keypair}}\ \mathbf{in}\ \mathbf{send}_{c_1}(\text{“B”}, \underline{\mathsf{pubkey}}(k_A^{-1})). \\
\big(\mathbf{receive}_{c_2}(y_B, k_B). \\
\mathbf{if}^{\mathcal{JJ}}\ y_B = \text{“B”}\ \mathbf{then}\ A'(y_B, k_B) \mid \\
!\mathbf{receive}_{c_i}[y_{X1}](k_X).A'(y_{X1}, k_X)\big)
\end{array}$$

B is

$$\begin{array}{l}
\mathbf{receive}_{c_u}(\mathit{fc2}, s_I).\mathbf{let}^{\mathbf{0}}\ s := \mathsf{store}(s_I)\ \mathbf{in} \\
\mathbf{let}^{\mathbf{0}}\ k_B^{-1} := \underline{\mathsf{keypair}}\ \mathbf{in}\ \mathbf{let}^{\mathbf{0}}\ k^B := \underline{\mathsf{pubkey}}(k_B^{-1})\ \mathbf{in} \\
\quad \mathbf{send}_{c_2}(\text{“A”}, k_B). \\
\mathbf{receive}_{c_1}(y_A, k_A).\mathbf{if}^{\mathcal{JJ}}\ y_A = \text{“A”}\ \mathbf{then} \\
!\mathbf{receive}_{c_i}[y_1](m_1).\mathbf{let}^{\mathcal{JJ}}\ l_1 := \underline{\mathsf{pubdec}}(k_B^{-1}, m_1)\ \mathbf{in} \\
\quad \mathbf{let}^{\mathcal{JJ}}\ s'_A := \mathsf{list_proj}(l_1, 1)\ \mathbf{in}\ \mathbf{let}^{\mathcal{JJ}}\ k_{A2} := \mathsf{list_proj}(l_1, 2)\ \mathbf{in} \\
\quad \mathbf{if}^{\mathcal{JJ}}\ k_A = k_{A2}\ \mathbf{then}\ \mathbf{let}^{\mathbf{0}}\ k_{AB} := \mathsf{gen_symenc_key}(1)\ \mathbf{in} \\
\quad \mathbf{send}_{c_i}(\text{“A”}, \underline{\mathsf{pubenc}}(k_A, \mathsf{list}(s'_A, k_{AB}, k_B))). \\
\mathbf{receive}_{c_i}(y_3, m_3).\mathbf{let}^{\mathcal{JJ}}\ k_{AB2} := \underline{\mathsf{pubdec}}(k_B^{-1}, m_3)\ \mathbf{in} \\
\quad \mathbf{if}^{\mathcal{JJ}}\ k_{AB} = k_{AB2}\ \mathbf{then}\ \mathbf{send}_{c_i}(\text{“A”}, \underline{\mathsf{privenc}}(k_{AB2}, s)).\mathbf{0}
\end{array}$$

Figure 4: Example protocol

$$T ::= T^{(I)} \mid T^{(H)}$$

$$T^{(I)} ::= \mathrm{PubRD} \mid \mathrm{SecRD} \mid \mathrm{AllRD}$$

$$T^{(H)} ::= T^{(A)} \mid \mathrm{DK}(T^{(A)})$$

$$\begin{array}{lllll}
T^{(A)} & ::= & \mathrm{Public} & \mid & \mathrm{PubData} \\
& \mid & \mathrm{SNonce} & \mid & \mathrm{SecData} \\
& \mid & \mathrm{EK}(T^{(A)}) & \mid & \mathrm{SK}^i(T^{(A)}) \\
& \mid & \mathrm{List}(T_1^{(A)}, \ldots, T_n^{(A)}) & \mid & T_1^{(A)} + T_2^{(A)}
\end{array}$$

Figure 5: The types for variables and channels

The types T are defined in Fig. 5. The type system contains two main kinds of types — the types $T^{(I)}$ are intended for "raw data", i.e. the data received from the protocol users, as well as for the identities of participants. The types $T^{(H)}$ are intended for the terms in the database of $\mathcal{TH}_\mathbf{n}$ (both for variables containing handles, and for abstract channels). The subtyping and sameness relations are given in Fig. 6. If two types are subtypes of each other then we count them as being the same.

The meaning of types $T^{(I)}$ should be clear — the data received from the user of the protocol gets the type SecRD and the data representing names of the parties gets the type PubRD. As terms in the database these have types $\mathrm{SecData}$ and $\mathrm{PubData}$, respectively. The types $\mathrm{DK}(T)$, $\mathrm{EK}(T)$ and $\mathrm{SK}^i(T)$ are respectively for asymmetric decryption, asymmetric encryption and symmetric keys (of order i) where T is the type of plaintexts. The types $\mathrm{DK}(T)$ are

$$\mathrm{PubRD} \leq \mathrm{AllRD} \qquad \mathrm{SecRD} \leq \mathrm{AllRD}$$

$$T \leq T \qquad \mathrm{PubData} \leq \mathrm{Public} \qquad \mathrm{EK}(T) \leq \mathrm{Public}$$

$$\big(\forall i : T_i \leq \mathrm{Public}\big) \Rightarrow \mathrm{List}(T_1, \ldots, T_n) \leq \mathrm{Public}$$

$$T_i \leq T_1 + T_2 \qquad (T_1 \leq T \wedge T_2 \leq T) \Rightarrow T_1 + T_2 \leq T$$

$$\big(\forall i : T_i \leq T_i'\big) \Rightarrow \mathrm{List}(T_1, \ldots, T_n) \leq \mathrm{List}(T_1', \ldots, T_n')$$

$$T_1{+}(T_2{+}T_3) \equiv (T_1{+}T_2){+}T_3 \quad T_1{+}T_2 \equiv T_2{+}T_1 \quad T{+}T \equiv T$$

$$\mathrm{List}(T_1, \ldots, T_i + T_i', \ldots, T_n) \equiv \begin{array}{l} \mathrm{List}(T_1, \ldots, T_i, \ldots, T_n) + \\ \mathrm{List}(T_1, \ldots, T_i', \ldots, T_n) \end{array}$$

Figure 6: Subtyping

kept separate because the library $\mathcal{TH}_\mathbf{n}$ restricts the use of these keys purely to the decryption of ciphertexts. Finally, a variable of type $T_1 + T_2$ contains a handle to a term with either type T_1 or T_2; subsequent parsing may make the typing more precise. We say that a type is *public* if it is a subtype of Public. The type Public has another special meaning — a variable of type Public (and not some subtype of it) may point to a *tainted* term — a term that is constructed by the adversary.

Not all types $\mathrm{SK}^i(T)$ are valid types — the orders of keys have to be respected. Let us define the *order* for all types $T^{(A)}$. The order of all public types, as well as $\mathrm{SecData}$ and SNonce is 0. The order of $\mathrm{SK}^i(T)$ is i. The order of lists and sums is the maximum order of their components. A type $\mathrm{SK}^i(T)$ is *valid* if the order of T is strictly less than i.

Note that our type system does not allow making a symmetric encryption key available to the adversary. The property NoComm [9] does not exclude revealing a symmetric key, but this must be done before the key is used for encryption. We do not believe that real protocols generate symmetric keys only to make them immediately available to the attacker, so we simplify the type system by excluding this case. This immediately takes care of the property (V).

As we said before, Γ maps the free variables of a process to types, i.e. if we say that a process P *types* according to Γ (denoted $\Gamma \vdash P$) then $dom(\Gamma)$ must at least include all free variables of P. The typing Γ also has to map channels between principals to types (of the messages passed on these channels). The abstract channels have been introduced for this purpose. There are even two types for each abstract channel — one for sent and one for received messages. These two types can by safely taken as equal if both ends of a channel connect to protocol participants, but this is not the case for channels between the participant and the user. The typing Γ must define two types, $\Gamma(c, \mathsf{s})$ and $\Gamma(c, \mathsf{r})$ for each abstract channel c occurring in the protocol. These may not be completely arbitrary — if $c \in \mathbf{Chan}_\mathsf{a}$ then $\Gamma(c, \mathsf{s})$ and $\Gamma(c, \mathsf{r})$ must be public types. Also, if $c \in \mathbf{Chan}_\mathsf{i}$ then both $\Gamma(c, \mathsf{s})$ and $\Gamma(c, \mathsf{r})$ must be equal to Public. For abstract channels c in $\mathbf{Chan}_\mathsf{s}$, $\mathbf{Chan}_\mathsf{a}$ and $\mathbf{Chan}_\mathsf{i}$ the inequality $\Gamma(c, \mathsf{s}) \leq \Gamma(c, \mathsf{r})$ must hold. If $c \in \mathbf{Chan}_\mathsf{u}$ then $\Gamma(c, \mathsf{r}) = \mathrm{SecRD}$ and $\Gamma(c, \mathsf{s}) = \mathrm{AllRD}$ — the inputs from the user have to be treated as secret, whereas anything can be sent back to him.

Compared to the type system of Abadi and Blanchet [2] a category of types is absent — the public encryption keys that the adversary is not allowed to know. The reason for this omission is the operation of the cryptographic library $\mathcal{TH}_\mathbf{n}$ — it tags each asymmetric encryption with the public encryption key used to produce it.

Let $T \perp T'$ denote that T and T' have no common subtypes. Let $\mathbf{TA}$ (resp. $\mathbf{TH}$) be the set of all types $T^{(A)}$ (resp. $T^{(H)}$) in Fig. 5. The typing judgments $\Gamma \vdash e : T$ for expressions e are given in Fig. 7. Here Γ must give types to all variables occurring in e. Beside $e : T$ we may be able to derive $\mathbf{NF}(e)$ or $\mathbf{AF}(e)$, meaning that the computation of the value of the expression e either never fails (i.e. results in $\perp$) or always fails. This knowledge is useful when typing the processes of form **let** $x := e$ **in** P **else** Q where it allows to leave either P or Q untyped (see rules (LetP, LetF) in Fig. 9). The rules for inferring the failure of expressions are given in Fig. 8. Note that never failing only makes sense if the expression has a type. Most of the typing rules for expressions should be obvious. The computation that is done with public inputs may always produce public outputs (PNce, PEU, SEU, List, StoP, SDU, PrT, Const, RetP). The meaning of the types $\mathrm{EK}(T)$ and $\mathrm{SK}^i(T)$ is reflected in the rules (PET, SET) — the encryption with these keys is enough to turn data of type T to public data. Also, if the decryption of public data succeeds with the symmetric key of type $\mathrm{SK}^i(T)$ then the plaintext must have been of type T (SDT). If the decryption succeeds with the asymmetric decryption key of type $\mathrm{DK}(T)$, then the plaintext may also be of type T, but it may also have been generated by the adversary (who may have learned the encryption key). Hence the type of decryption is either T or Public (PD).

The typing rules for the processes are given in Fig. 9. We see that a message received from a channel gets the type of this channel (Recv) while the sender of that message is known to the adversary. The handles may be compared only if it is certain that the corresponding terms are not created by store-commands storing messages received from the user (IfH). Sometimes we can deduce that the compared terms cannot be equal (if their types are necessarily different), then we do not have to worry about (i.e. type) the $\mathbf{then}$-branch (IfH).

Example

There exists a typing Γ with $dom(\Gamma) = \{c_1, c_2, c_i, c_u\} \times \{\mathsf{s}, \mathsf{r}\}$, such that both processes A and B in Fig. 4 type with respect to it. Let us denote $T_1 = \mathrm{List}(\mathrm{SNonce}, \mathrm{SK}^1(\mathrm{SecData}), \mathrm{EK}(T_2))$ and $T_2 = \mathrm{List}(\mathrm{SNonce}, \mathrm{Public}) + \mathrm{SK}^1(\mathrm{SecData})$. If we take $\Gamma(c_i, \mathsf{s}) = \Gamma(c_i, \mathsf{r}) = \mathrm{Public}$, $\Gamma(c_u, \mathsf{r}) = \mathrm{SecRD}$, $\Gamma(c_u, \mathsf{s}) = \mathrm{AllRD}$, $\Gamma(c_1, \mathsf{s}) = \Gamma(c_1, \mathsf{r}) = \mathrm{EK}(T_1)$ and $\Gamma(c_2, \mathsf{s}) = \Gamma(c_2, \mathsf{r}) = \mathrm{EK}(T_2)$, then this satisfies the constraints put on the values of $\Gamma(c)$ for channels c in $\mathbf{Chan}_\mathsf{a}$, $\mathbf{Chan}_\mathsf{i}$ or $\mathbf{Chan}_\mathsf{u}$. To show that $\Gamma \vdash A$ and $\Gamma \vdash B$, let us mention the types that some variables get when typing A and B; the rest of the types should be easy to figure out. We mostly mention the type when the variable points to newly generated data.

In process A, variable k_A^{-1} gets the type $\mathrm{DK}(T_1)$. The process A' is included in A twice (first time for $k_X = k_B$ and second time for possibly tainted k_X-s), the types of variables are somewhat different there. The variable s_A gets the type SNonce in the first case and Public in the second. We need to give the type Public to $\underline{\mathsf{pubenc}}(k_X, \mathsf{list}(s_A, k_A))$, this is done with help of the rule (PET) in the first case and (PEU) in the second. The value l_2 gets a sum type, at this point we use the rule (Sum), giving us two subcases for both cases. The comparison **if** $s_{A2} = s_A$ effectively rules out $l_1 : \mathrm{Public}$ in the first case and $l_1 : T_1$ in the second case because of incompatible types (we find that the *else*-part is definitely taken). In the remaining subcases $\underline{\mathsf{pubenc}}(k_{X2}, k_{AX}')$ has the type Public

$$\frac{\Gamma \vdash e : T \quad T \leq T'}{\Gamma \vdash e : T'}\ (\text{Sub}) \qquad \overline{\overline{\Gamma \vdash n : \mathrm{PubRD}}}\ (\text{Const}) \qquad \overline{\overline{\Gamma \vdash x : \Gamma(x)}}\ (\text{Var}) \qquad \frac{\forall i : \Gamma \vdash e_i : T_i \quad \forall i : T_i \in \mathbf{TA}}{\Gamma \vdash \mathsf{list}(e_1, \dots, e_k) : \mathrm{List}(T_1, \dots, T_k)}\ (\text{List})$$

$$\frac{\Gamma \vdash e : \mathrm{PubRD}}{\Gamma \vdash \mathsf{store}(e) : \mathrm{PubData}}\ (\text{StoP}) \qquad \frac{\Gamma \vdash e : \mathrm{SecRD}}{\Gamma \vdash \mathsf{store}(e) : \mathrm{SecData}}\ (\text{StoS}) \qquad \overline{\overline{\Gamma \vdash \mathsf{gen\_symenc\_key}(i) : \mathrm{SK}^i(T)}}\ (\text{SK})$$

$$\overline{\overline{\Gamma \vdash \underline{\mathsf{keypair}} : \mathrm{DK}(T)}}\ (\text{KP}) \qquad \overline{\overline{\Gamma \vdash \mathsf{gen\_nonce} : \mathrm{SNonce}}}\ (\text{SNce}) \qquad \overline{\overline{\Gamma \vdash \mathsf{gen\_nonce} : \mathrm{Public}}}\ (\text{PNce})$$

$$\frac{\Gamma \vdash e_{\mathrm{k}} : \mathrm{Public} \quad \Gamma \vdash e_{\mathrm{t}} : \mathrm{Public}}{\Gamma \vdash \underline{\mathsf{pubenc}}(e_{\mathrm{k}}, e_{\mathrm{t}}) : \mathrm{Public}}\ (\text{PEU}) \qquad \frac{\Gamma \vdash e_{\mathrm{k}} : \mathrm{EK}(T) \quad \Gamma \vdash e_{\mathrm{t}} : T}{\Gamma \vdash \underline{\mathsf{pubenc}}(e_{\mathrm{k}}, e_{\mathrm{t}}) : \mathrm{Public}}\ (\text{PET}) \qquad \frac{\Gamma \vdash e_{\mathrm{k}} : \mathrm{Public} \quad \Gamma \vdash e_{\mathrm{t}} : \mathrm{Public}}{\Gamma \vdash \underline{\mathsf{privenc}}(e_{\mathrm{k}}, e_{\mathrm{t}}) : \mathrm{Public}}\ (\text{SEU})$$

$$\frac{\Gamma \vdash e_{\mathrm{k}} : \mathrm{SK}^i(T) \quad \Gamma \vdash e_{\mathrm{t}} : T}{\Gamma \vdash \underline{\mathsf{privenc}}(e_{\mathrm{k}}, e_{\mathrm{t}}) : \mathrm{Public}}\ (\text{SET}) \qquad \frac{\Gamma \vdash e : \mathrm{Public}}{\Gamma \vdash \mathsf{list\_proj}(e, i) : \mathrm{Public}}\ (\text{PrT}) \qquad \frac{\Gamma \vdash e : \mathrm{List}(T_1, \dots, T_k)}{\Gamma \vdash \mathsf{list\_proj}(e, i) : T_i}\ (\text{PrL})$$

$$\frac{\Gamma \vdash e_{\mathrm{k}} : \mathrm{Public} \quad \Gamma \vdash e_{\mathrm{t}} : \mathrm{Public}}{\Gamma \vdash \underline{\mathsf{privdec}}(e_{\mathrm{k}}, e_{\mathrm{t}}) : \mathrm{Public}}\ (\text{SDU}) \qquad \frac{\Gamma \vdash e_{\mathrm{k}} : \mathrm{SK}^i(T) \quad \Gamma \vdash e_{\mathrm{t}} : \mathrm{Public}}{\Gamma \vdash \underline{\mathsf{privdec}}(e_{\mathrm{k}}, e_{\mathrm{t}}) : T}\ (\text{SDT}) \qquad \frac{\Gamma \vdash e_{\mathrm{k}} : \mathrm{DK}(T) \quad \Gamma \vdash e_{\mathrm{t}} : T}{\Gamma \vdash \underline{\mathsf{pubdec}}(e_{\mathrm{k}}, e_{\mathrm{t}}) : T + \mathrm{Public}}\ (\text{PD})$$

$$\frac{\Gamma \vdash e : \mathrm{DK}(T)}{\Gamma \vdash \underline{\mathsf{pubkey}}(e) : \mathrm{EK}(T)}\ (\text{PK}) \qquad \frac{\Gamma \vdash e : \mathrm{Public}}{\Gamma \vdash \mathsf{retrieve}(e) : \mathrm{PubRD}}\ (\text{RetP}) \qquad \frac{\Gamma \vdash e : T \quad T \not\leq \mathrm{SecData}}{\Gamma \vdash \mathsf{retrieve}(e) : \mathrm{SecRD}}\ (\text{RetS})$$

Figure 7: Typing expressions

$$\frac{e' \text{ is a subexpression of } e \quad \Gamma \vdash \mathbf{AF}(e')}{\Gamma \vdash \mathbf{AF}(e)}\ (\text{fSub}) \qquad \frac{e \in \{n, x, \mathsf{gen\_symenc\_key}(i), \underline{\mathsf{keypair}}, \mathsf{gen\_nonce}\}}{\Gamma \vdash \mathbf{NF}(e)}\ (\text{fNoSub})$$

$$\frac{\Gamma \vdash e : T \quad \forall k \geq i \, \forall T_1, \dots, T_k : T \perp \mathrm{List}(T_1, \dots, T_k)}{\Gamma \vdash \mathbf{AF}(\mathsf{list\_proj}(e, i))}\ (\text{fPr}) \qquad \frac{\forall i : \Gamma \vdash \mathbf{NF}(e_i)}{\Gamma \vdash \mathbf{NF}(\mathsf{list}(e_1, \dots, e_k))}\ (\text{fList}) \qquad \frac{\exists i : \Gamma \vdash e_i : T \quad T \notin \mathbf{TA}}{\Gamma \vdash \mathbf{AF}(\mathsf{list}(e_1, \dots, e_k))}\ (\text{fList'})$$

$$\frac{\Gamma \vdash \mathbf{NF}(e)}{\Gamma \vdash \mathbf{NF}(\mathsf{store}(e))}\ (\text{fSto}) \qquad \frac{\Gamma \vdash e_{\mathrm{k}} : T \quad \forall i, T' : T \perp \mathrm{SK}^i(T') \quad T \perp \mathrm{Public}}{\Gamma \vdash \mathbf{AF}(\underline{\mathsf{privenc}}(e_{\mathrm{k}}, e_{\mathrm{t}}))}\ (\text{fSE}) \qquad \frac{\Gamma \vdash e_{\mathrm{k}} : T \quad \forall T' : T \perp \mathrm{DK}(T')}{\Gamma \vdash \mathbf{AF}(\underline{\mathsf{pubdec}}(e_{\mathrm{k}}, e_{\mathrm{t}}))}\ (\text{fPD})$$

$$\frac{\Gamma \vdash e : T \quad \forall T' : T \perp \mathrm{DK}(T')}{\Gamma \vdash \mathbf{AF}(\underline{\mathsf{pubkey}}(e))}\ (\text{fPK}) \qquad \frac{\Gamma \vdash e_{\mathrm{k}} : T \quad \forall T' : T \perp \mathrm{EK}(T')}{\Gamma \vdash \mathbf{AF}(\underline{\mathsf{pubenc}}(e_{\mathrm{k}}, e_{\mathrm{t}}))}\ (\text{fPE}) \qquad \frac{\Gamma \vdash e : T \quad T \in \mathbf{TA} \quad T \perp \mathrm{SecData} + \mathrm{PubData}}{\Gamma \vdash \mathbf{AF}(\mathsf{retrieve}(e))}\ (\text{fRet})$$

Figure 8: Failing of expressions

$$\overline{\overline{\Gamma \vdash \mathbf{0}}}\ (\text{Zero}) \qquad \overline{\overline{\Gamma \vdash \mathfrak{II}}}\ (\text{II}) \qquad \frac{x \in FV(P) \quad \Gamma[x \mapsto T_1] \vdash P \quad \Gamma[x \mapsto T_2] \vdash P}{\Gamma[x \mapsto T_1 + T_2] \vdash P}\ (\text{Sum}) \qquad \frac{\forall i : \Gamma \vdash I_i}{\Gamma \vdash I_1 \mid \cdots \mid I_n \mid \mathbf{0}}\ (\text{Par})$$

$$\frac{\Gamma \vdash I^* \quad \Gamma \vdash e_{\mathrm{p}} : \mathrm{PubRD} \quad \Gamma \vdash e : \Gamma(c, \mathsf{s})}{\Gamma \vdash \mathbf{send}_c(e_{\mathrm{p}}, e).I^*}\ (\text{Send}) \qquad \frac{\Gamma[x \mapsto \Gamma(c, \mathsf{r}), x_{\mathrm{p}} \mapsto \mathrm{PubRD}] \vdash P}{\Gamma \vdash (!)\mathbf{receive}_c(x_{\mathrm{p}}, x).P}\ (\text{Recv})$$

$$\frac{\Gamma \vdash e : T \quad \Gamma[x \mapsto T] \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathbf{let}\ x := e\ \mathbf{in}\ P\ \mathbf{else}\ Q}\ (\text{Let}) \qquad \frac{\Gamma \vdash e : T \quad \Gamma[x \mapsto T] \vdash P \quad \Gamma \vdash \mathbf{NF}(e)}{\Gamma \vdash \mathbf{let}\ x := e\ \mathbf{in}\ P\ \mathbf{else}\ Q}\ (\text{LetP}) \qquad \frac{\Gamma \vdash Q \quad \Gamma \vdash \mathbf{AF}(e)}{\Gamma \vdash \mathbf{let}\ x := e\ \mathbf{in}\ P\ \mathbf{else}\ Q}\ (\text{LetF})$$

$$\frac{\Gamma \vdash e : T \quad \Gamma \vdash e' : T' \quad T, T' \in \mathbf{TA} \quad \mathrm{SecData} \perp T + T' \quad \text{if } T \not\leq T' \text{ then } \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathbf{if}\ e = e'\ \mathbf{then}\ P\ \mathbf{else}\ Q}\ (\text{IfH})$$

$$\frac{\Gamma \vdash e : \mathrm{PubRD} \quad \Gamma \vdash e' : \mathrm{PubRD} \quad \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathbf{if}\ e = e'\ \mathbf{then}\ P\ \mathbf{else}\ Q}\ (\text{IfR})$$

Figure 9: Typing processes and input processes

either by (PET) or (PEU). The types of the rest of the variables are obvious.

In process B the variable k_B^{-1} gets the type $\mathrm{DK}(T_2)$. The value l_1 gets a sum type T_2. At this point we use the (Sum) rule. The case where $l_1 : \mathrm{SK}^1(\mathrm{SecData})$ is immediately ruled out by the following list_proj-operation. The variable k_{AB} obviously gets the type $\mathrm{SK}^1(\mathrm{SecData})$. The variable k_{AB2} also gets the sum type T_2. We use the (Sum) rule and the case $k_{AB2} : \mathrm{List}(\ldots)$ is ruled out by the following comparison **if** $k_{AB} = k_{AB2}$. It is straightforward to verify that all **send**- and **if**- statements type in both A and B.

6. CORRECTNESS OF THE TYPE SYSTEM

The type system is correct in the following sense.

THEOREM 1 (PRESERVATION OF SECRECY). *Let* $\mathbf{Chan}_P$ *be the set of abstract channels used by all processes in the initial states of all the machines* P_i. *If there exists a typing* Γ *with* $dom(\Gamma) = \mathbf{Chan}_P \times \{\mathsf{s},\mathsf{r}\}$, *such that* Γ *satisfies the constraints put on* $\Gamma(c,\mathsf{s})$ *and* $\Gamma(c,\mathsf{r})$, *as stated in Sec. 5, and* $\Gamma \vdash I$ *for all input processes* I *in the initial states of the machines* P_i *executing the protocols then the execution of the configurations of the structure* $C'_{\mathbf{n}}$ *satisfies the properties (I)–(V).*

From this result and from the preservation of secrecy of messages under simulatability we immediately get

COROLLARY 2. *If the conditions of Theorem 1 are met then the structure* $C_{\mathbf{n}}$ *preserves the secrecy of messages that it receives over the channels* $\mathsf{pin}_{\mathsf{u}_i}$.

Let us give a sketch of the correctness proof; full details can be found in [34]. The main tool in showing the correctness of the type system is a result similar to subject reduction. For this we have to keep track of the secrets given to participants of the protocol by the users, as well as the orders of symmetric keys. We extend the definition of values (Fig. 1) by $v ::= \cdots \mid n^{\mathrm{trk}}$ and of expressions by $e ::= \cdots \mid n^{\mathrm{trk}}$, where $n \in \mathbb{N}$; we call such values *tracked*. If the participant P_i has received a message n from the i-th user then this message is saved in the state of P_i as n^{trk}. We must ensure that the conditions (I)–(III) hold for these values. We also change the workings of some commands of $\mathcal{TH}_{\mathbf{n}}$, such that the "raw data" that has been stored in the database of $\mathcal{TH}_{\mathbf{n}}$ can be tracked, too. In particular the command store of $\mathcal{TH}_{\mathbf{n}}$ must be aware whether its argument is tracked and store it differently from untracked values in the database. The trackedness is also preserved in the return values of the command retrieve. To remain faithful to the observable behavior of the structure $C'_{\mathbf{n}}$ we define that in an **if**-statement of some process, the values n and n^{trk} are considered equal.

Another slight modification to the operation of $\mathcal{TH}_{\mathbf{n}}$ is necessary; the observable semantics remains unchanged by it. Namely, when P_i sends the command gen_symenc_key(i) to $\mathcal{TH}_{\mathbf{n}}$ for generating a new symmetric key of order i then $\mathcal{TH}_{\mathbf{n}}$ also stores i together with the newly created term denoting the key.

For defining when a state $\mathbf{C}$ types according to Γ we have to extend Γ to the terms T in the database of $\mathcal{TH}_{\mathbf{n}}$ and to the handles of terms that the machine $\mathcal{TH}_{\mathbf{n}}$ has returned to the participants. We denote the type of the handle $n \in \mathbb{N}$ given to the i-th participant by $\Gamma(n,i)$. We also have to extend the typing relation $\vdash$. The semantics of processes replaces variables with their values, but the typing rules in Fig. 7 do not allow an integer to represent a handle to a term. For each participant identity i we define a relation $\vdash_i$ for expressions and processes. All the rules given in Figures 7–9 are also defined to hold for $\vdash_i$. Additionally we introduce the axioms $\Gamma \vdash_i n^{\mathrm{trk}} : \mathrm{SecRD}$, $\Gamma \vdash_i n : \Gamma(n,i)$, and $\Gamma \vdash_i \mathbf{NF}(n^{\mathrm{trk}})$. We write $\Gamma \vdash \mathbf{C}$ if the following holds:

(A) $\Gamma \vdash_i P$ for all (input) processes P in the state $\mathbf{S}_i$, for all i. If some state $\mathbf{S}_i$ is active and the source of the message in that state is not some participant then the received message in that state is a handle to some term T in the database of $\mathcal{TH}_{\mathbf{n}}$ and $\Gamma(\mathsf{T}) \leq \Gamma(c,\mathsf{r})$ where c is the abstract channel name that is stored in $\mathbf{S}_i$. If there is a message that awaits sending (a process has just executed a **send**-command) then it is a pair where the first component indicates the abstract channel c, the second component is the actual message and the type of the second component, given by Γ is less or equal to $\Gamma(c,\mathsf{s})$. The same condition (the type of message is less or equal to the type of the abstract channel) holds for the messages in the buffers $\mathbf{L}_c^{i\to j}$ of secure and authentic channels. Additionally, the abstract channel recorded in the message must correspond to the security level of the concrete channel.

(B) If n has been given as a handle to the term T to some participant i and n is present in the state $\mathbf{S}_i$ (as part of an expression in a process, or as a handle to the received message) then $\Gamma(n,i) = \Gamma(\mathsf{T})$.

(C) Let τ be the type of the term T, as recorded by $\mathcal{TH}_{\mathbf{n}}$. Depending on τ, the type $\Gamma(\mathsf{T})$ must be one of the following:

- $\tau = \mathsf{data}$: $\Gamma(\mathsf{T})$ must be either $\mathrm{PubData}$ or $\mathrm{SecData}$. If the stored data is n^{trk} then $\Gamma(\mathsf{T})$ must be $\mathrm{SecData}$.
- $\tau = \mathsf{list}$: $\Gamma(\mathsf{T})$ must be $\mathrm{List}(\Gamma(\mathsf{T}_1),\ldots,\Gamma(\mathsf{T}_k))$ where $\mathsf{T}_1,\ldots,\mathsf{T}_k$ are the immediate subterms of T.
- $\tau = \mathsf{nonce}$: $\Gamma(\mathsf{T})$ must be either SNonce or Public.
- $\tau = \mathsf{skse}$: $\Gamma(\mathsf{T})$ must be either Public (if T was created by the adversary) or $\mathrm{SK}^i(T)$ for some T, where i is the order of T (if T was created by an honest party).
- $\tau = \mathsf{ske}$: $\Gamma(\mathsf{T})$ must be either Public or $\mathrm{DK}(T)$ for some T. If T is generated by a protocol party (i.e. a protocol party has a handle to it) then $\Gamma(\mathsf{T}) \neq \mathrm{Public}$.
- $\tau = \mathsf{pke}$: Let T' be the term representing the corresponding secret key. If $\Gamma(\mathsf{T}') = \mathrm{Public}$ then $\Gamma(\mathsf{T}) = \mathrm{Public}$. If $\Gamma(\mathsf{T}') = \mathrm{DK}(T)$ then $\Gamma(\mathsf{T}) = \mathrm{EK}(T)$.
- for other values of τ, $\Gamma(\mathsf{T})$ must be Public.

We call a term *public* if its type, according to Γ, is public. Otherwise we call a term secret.

(D) If the adversary has handle to a term T then T is public.

(E) If a term T is public and its immediate subterm T' is secret (the subterms of a ciphertext are the plaintext and the *public* key of type pke or pkse) then the type of T, as recorded by $\mathcal{TH}_{\mathbf{n}}$, is enc or symenc, T' is the corresponding plaintext, and the *decryption* key is secret.

(F) If the type of the term T, as recorded by $\mathcal{TH}_{\mathbf{n}}$, is symenc, and the type (by Γ) of the corresponding key is $\mathrm{SK}^i(T)$ for some type T, then the type (by Γ) of the plaintext must be a subtype of $\mathrm{List}(T)$. If the type of T is enc and the type of the corresponding decryption key is $\mathrm{DK}(T)$ then the type of the plaintext must be a subtype of $\mathrm{List}(T)$ or a public type.

With the definition of $\Gamma \vdash \mathbf{C}$ in place, the following lemmas and theorems are straightforward to prove:

LEMMA 3. *Let* $\mathbf{C}$ *be a state of the configuration* C, *let* $\Gamma \vdash \mathbf{C}$. *Let* e *be a closed expression,* v *be a value and* $e \;{}^{\mathbf{C}.\mathcal{O}}\!\Downarrow^{i}_{\mathcal{O}'}\; v$. *Let* $\mathbf{C}' = \mathbf{C}_{\mathcal{O}\leftarrow\mathcal{O}'}$. *Let* $\Gamma \vdash_i e : T$. *Then there exists a typing* Γ' *of* $\mathbf{C}'$ *that extends* Γ *(i.e.* $dom(\Gamma') \supseteq dom(\Gamma)$ *and* Γ' *agrees with* Γ *on all arguments from* $dom(\Gamma)$*), such that* $\Gamma' \vdash \mathbf{C}'$, *and if* $T \in \mathbf{TH}$ *then* v *is either* $\perp$ *or contains a handle to a term* T_v, *such that* $\Gamma'(\mathsf{T}_v) \leq T$.

LEMMA 4. *Let* $\mathbf{C}$ *be a state of the configuration* C*, let* $\Gamma \vdash \mathbf{C}$*. Let* e *be an expression and let* v *be the result of the* i*-th participant computing its value in the state* $\mathbf{C}$*. If* $\Gamma \vdash_i \mathbf{NF}(e)$ *and* e *has a type according to* Γ *and* $\vdash_i$ *then* $v \neq \perp$*. If* $\Gamma \vdash_i \mathbf{AF}(e)$ *then* $v = \perp$*.*

THEOREM 5 (SUBJECT REDUCTION). *Let* $\mathbf{C}$ *and* $\mathbf{C}'$ *be two states of the composition* C *and let* Γ *be a typing such that* $\Gamma \vdash \mathbf{C}$ *and* $\mathbf{C} \rightarrow \mathbf{C}'$*. Then there exists a typing* Γ' *extending* Γ*, such that* $\Gamma' \vdash \mathbf{C}'$*.*

7. CONCLUSIONS

We have presented a rather simple type-system for protocols that use the Backes-Pfitzmann-Waidner (BPW) cryptographic library. The type system can be used to show that the protocol preserves the secrecy of input messages. Our result shows that the existing methods of (semi)automatically checking the security of protocols (not only type systems, but also model-checking, program analysis, constraint solving, etc.) are applicable to that library. Indeed, Abadi and Blanchet [3] describe an automatic tool whose operation is equivalent to the type inference according to their type system; this tool should be readily modifiable for our type system. We are also quite confident that the more complex type systems by Gordon and Jeffrey [26, 25, 27] for checking integrity properties in protocols using Dolev-Yao model can be carried over to the BPW library.

8. ACKNOWLEDGMENTS

We are thankful to Cédric Fournet and to the anonymous referees of CCS 2005 for their comments. This research has been supported by Estonian Science Foundation, grant #6095.

9. REFERENCES

[1] M. Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM*, 46(5):749–786, Sept. 1999.

[2] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387–415, 2003.

[3] M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. *Journal of the ACM*, 52(1):102–146, Jan. 2005.

[4] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, Jan. 1999.

[5] M. Abadi and J. Jürjens. Formal Eavesdropping and Its Computational Interpretation. In proc. of *TACS 2001* (LNCS 2215), pages 82–94.

[6] M. Abadi and P. Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In proc. of *International Conference IFIP TCS 2000* (LNCS 1872) pages 3–22.

[7] M. Backes. A Cryptographically Sound Dolev-Yao Style Security Proof of the Otway-Rees Protocol. In proc. of *ESORICS 2004* (LNCS 3193) pages 89–108.

[8] M. Backes and B. Pfitzmann. A Cryptographically Sound Security Proof of the Needham-Schroeder-Lowe Public-Key Protocol. In proc. of *FST TCS 2003* (LNCS 2914), pages 1–12.

[9] M. Backes and B. Pfitzmann. Symmetric Encryption in a Simulatable Dolev-Yao Style Cryptographic Library. In proc. of *IEEE CSFW* 2004, pages 204–218.

[10] M. Backes and B. Pfitzmann. Relating Symbolic and Cryptographic Secrecy. In proc. of *IEEE S&P* 2005.

[11] M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. In proc. of *ESORICS* 2003 (LNCS 2808), pages 271–290.

[12] M. Backes, B. Pfitzmann, and M. Waidner. A Universally Composable Cryptographic Library. In proc. of *ACM CCS* 2003, pages 220-230.

[13] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption. In proc. of *FOCS* 1997, pages 394–403.

[14] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among Notions of Security for Public-Key Encryption Schemes. In proc of. *CRYPTO* ’98 (LNCS 1462), pages 26–45.

[15] B. Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In proc. of *IEEE S&P* 2004, pages 86–100.

[16] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In proc. of *FOCS* 2001, pages 136–145.

[17] R. Canetti and M. Fischlin. Universally Composable Commitments. In proc. of *CRYPTO* 2001 (LNCS 2139), pages 19–40.

[18] R. Canetti and J. Herzog. Universally Composable Symbolic Analysis of Cryptographic Protocols (The case of encryption-based mutual authentication and key exchange). Cryptology ePrint Archive: Report 2004/334, 22 Feb. 2005.

[19] R. Canetti and H. Krawczyk. Universally Composable Notions of Key Exchange and Secure Channels. In proc. of *EUROCRYPT* 2002 (LNCS 2332), pages 337–351.

[20] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In proc. of *STOC* 2002, pages 494–503.

[21] V. Cortier and B. Warinschi. Computationally Sound, Automated Proofs for Security Protocols. In proc. of *ESOP* 2005 (LNCS 3444), pages 157–171.

[22] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, Mar. 1983.

[23] O. Goldreich. *Foundations of Cryptography (Basic Tools)*. Cambridge University Press, 2001.

[24] S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Sciences*, 28(2):270–299, Apr. 1984.

[25] A. D. Gordon and A. Jeffrey. Authenticity by Typing for Security Protocols. *Journal of Computer Security*, 11(4):451–520, 2003.

[26] A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1-3):379–409.

[27] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3-4):435–483, 2004.

[28] J. Guttman, F. Thayer, and L. Zuck. The faithfulness of abstract protocol analysis: message authentication. In proc. of *ACM CCS* 2001, pages 186–195.

[29] J. Herzog, M. Liskov, and S. Micali. Plaintext Awareness via Key Registration. In proc. of *CRYPTO* 2003 (LNCS 2729), pages 548–564.

[30] R. Janvier, Y. Lakhnech, and L. Mazaré. Completing the Picture: Soundness of Formal Encryption in the Presence of Active Adversaries. In proc. of *ESOP* 2005 (LNCS 3444), pages 172–185.

[31] R. Janvier, Y. Lakhnech, and L. Mazaré. (De)Compositions of Cryptographic Schemes and their Applications to Protocols. Cryptology ePrint Archive, Report 2005/020, 1 Feb. 2005.

[32] P. Laud. Handling Encryption in Analyses for Secure Information Flow. In proc. of *ESOP* 2003 (LNCS 2618), pages 159–173.

[33] P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In proc. of *IEEE S&P* 2004, pages 71–85.

[34] P. Laud. Secrecy Types for a Simulatable Cryptographic Library. Research Report IT-LU-O-162-050823, Cybernetica, Aug. 2005.

[35] P. Laud and V. Vene. A Type System for Computationally Secure Information Flow. In proc. of *FCT* 2005 (LNCS 3623), pages 365–377.

[36] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A Probabilistic Poly-Time Framework for Protocol Analysis. In proc. of *ACM CCS* ’98, pages 112–121.

[37] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic Polynomial-Time Equivalence and Security Analysis. In proc. of *FM* ’99 (LNCS 1708), pages 776–793.

[38] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In proc. of *TACAS* ’96 (LNCS 1055), pages 147–166.

[39] C. Meadows. Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends. *IEEE Journal on Selected Areas in Communication*, 21(1):44–54, Jan. 2003.

[40] D. Micciancio and S. Panjwani. Adaptive Security of Symbolic Encryption. In proc. of *TCC* 2005 (LNCS 3378), pages 169–187.

[41] D. Micciancio and B. Warinschi. Completeness theorems for the Abadi-Rogaway logic of encrypted expressions. In *Workshop on Issues in the Theory of Security - WITS 2002*, Portland, Oregon, Jan. 2002.

[42] D. Micciancio and B. Warinschi. Soundness of Formal Encryption in the Presence of Active Adversaries. In proc. *TCC* 2004 (LNCS 2951), pages 133-151.

[43] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, Dec. 1978.

[44] B. Pfitzmann and M. Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In *IEEE S&P* 2001, pages 184–200.

[45] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In proc. of *CRYPTO* ’91 (LNCS 576), pages 433–444.

[46] A. Ramanathan, J. C. Mitchell, A. Scedrov, and V. Teague. Probabilistic Bisimulation and Equivalence for Security Analysis of Network Protocols. In proc. of *FOSSACS* 2004 (LNCS 2987), pages 468–483.

[47] A. Yao. Theory and applications of trapdoor functions (extended abstract). In proc. of *FOCS* ’82, pages 80–91.

[48] R. Zunino and P. Degano. A Note on the Perfect Encryption Assumption in a Process Calculus. In proc. of *FOSSACS* 2004 (LNCS 2987), pages 514–528.