Type-Based Automated Verification of Authenticity in Asymmetric Cryptographic Protocols

Morten Dahl[2], Naoki Kobayashi[1], Yunde Sun[1], and Hans Hüttel[2]

[1] Tohoku University
[2] Aalborg University

Abstract. Gordon and Jeffrey developed a type system for verification of asymmetric and symmetric cryptographic protocols. We propose a modified version of Gordon and Jeffrey's type system and develop a type inference algorithm for it, so that protocols can be verified automatically as they are, without any type annotations or explicit type casts. We have implemented a protocol verifier SPICA2 based on the algorithm, and confirmed its effectiveness.

1 Introduction

Security protocols play a crucial role in today's Internet technologies including electronic commerce and voting. Formal verification of security protocols is thus an important, active research topic, and a variety of approaches to (semi-)automated verification have been proposed [8, 5, 15]. Among others, type-based approaches [1, 14, 15] have advantages that protocols can be verified in a modular manner, and that it is relatively easy to extend them to verify protocols at the source code level [4]. They have however a disadvantage that users have to provide complex type annotations, which require expertise in both security protocols and type theories. Kikuchi and Kobayashi [18] developed a type inference algorithm but it works only for symmetric cryptographic protocols.

To overcome the limitation of the type-based approaches and enable fully automated protocol verification, we integrate and extend the two lines of work – Gordon and Jeffrey's work [15] for verifying protocols using both symmetric and asymmetric cryptographic protocols, and Kikuchi and Kobayashi's work. The outcome is an algorithm for automated verification of authenticity in symmetric and asymmetric cryptographic protocols. The key technical novelty lies in the symmetric notion of *obligations* and *capabilities* attached to name types, which allows us to reason about causalities between actions of protocol participants in a general and uniform manner in the type system. It not only enables automated type inference, but also brings a more expressive power, enabling, e.g., verification of multi-party cryptographic protocols. We have developed a type inference algorithm for the new type system, and implemented a protocol verification tool SPICA2 based on the algorithm. According to experiments, SPICA2 is very fast; it could successfully verify a number of protocols in less than a second.

The rest of this paper is structured as follows. Section 2 introduces spi-calculus [2] extended with correspondence assertions as a protocol description language. Sections 3 and 4 present our type system and sketches a type inference algorithm. Section 5 reports implementation and experiments. Sections 6 and 7 discuss extensions and related work respectively. Proofs are found in the full version of this paper [10].

2 Processes

This section defines the syntax and operational semantics of the spi-calculus extended with correspondence assertions, which we call spi$_{CA}$. The calculus is essentially the same as that of Gordon and Jeffrey [15], except (i) there are no type annotations or casts (as they can be automatically inferred by our type inference algorithm), and (ii) there are no primitives for witness and trust; supporting them is left for future work.

We assume that there is a countable set of *names*, ranged over by $m, n, k, x, y, z, \ldots$. By convention, we often use $k, m, n, \ldots$ for free names and $x, y, z, \ldots$ for bound names.

The set of messages, ranged over by M, is given by:

$$M ::= x \mid (M_1, M_2) \mid \{M_1\}_{M_2} \mid \{\!|M_1|\!\}_{M_2}$$

(M_1, M_2) is a pair consisting of M_1 and M_2. The message $\{M_1\}_{M_2}$ ($\{\!|M_1|\!\}_{M_2}$, resp.) represents the ciphertext obtained by encrypting M_1 with the symmetric (asymmetric, resp.) key M_2. For the asymmetric encryption, we do not distinguish between encryption and signing; $\{\!|M_1|\!\}_{M_2}$ denotes an encryption if M_2 is a public key, while it denotes signing if M_2 is a private key.

The set of processes, ranged over by P, is given by:

$$\begin{array}{l} P ::= \mathbf{0} \mid M_1!M_2 \mid M?x.P \mid (P_1 \mid P_2) \mid *P \mid (\nu x)P \mid (\nu_{sym} x)P \mid (\nu_{asym} x, y)P \\ \quad \mid \mathbf{check}\ M_1\ \mathbf{is}\ M_2.P \mid \mathbf{split}\ M\ \mathbf{is}\ (x, y).P \mid \mathbf{match}\ M_1\ \mathbf{is}\ (M_2, y).P \\ \quad \mid \mathbf{decrypt}\ M_1\ \mathbf{is}\ \{x\}_{M_2}.P \mid \mathbf{decrypt}\ M_1\ \mathbf{is}\ \{\!|x|\!\}_{M_2^{-1}}.P \\ \quad \mid \mathbf{begin}\, M.P \mid \mathbf{end}\, M \end{array}$$

The names denoted by x, y are *bound* in P. We write $[M_1/x_1, \ldots, M_n/x_n]P$ for the process obtained by replacing every free occurrence of $x_1, \ldots, x_n$ in P with $M_1, \ldots,$ M_n. We write $\mathbf{FN}(P)$ for the set of free (i.e. non-bounded) names in P.

Process $\mathbf{0}$ does nothing, $M_1!M_2$ sends M_2 over the channel M_1, and $M_1?x.P$ waits to receive a message on channel M_1, and then binds x to it and behaves like P. $P_1 \mid P_2$ executes P_1 and P_2 in parallel, and $*P$ executes infinitely many copies of P in parallel.

We have three kinds of name generation primitives: (νx) for ordinary names, $(\nu_{sym} x)$ for symmetric keys, and $(\nu_{asym} x_1, x_2,)$ for asymmetric keys. $(\nu_{asym} x_1, x_2, P)$ creates a fresh key pair (k_1, k_2) (where k_1 and k_2 are encryption and decryption keys respectively), and behaves like $[k_1/x_1, k_2/x_2]P$. The process $\mathbf{check}\ M_1\ \mathbf{is}\ M_2.P$ behaves like P if M_1 and M_2 are the same name, and otherwise behaves like $\mathbf{0}$. The process $\mathbf{split}\ M\ \mathbf{is}\ (x, y).P$ behaves like $[M_1/x, M_2/y]P$ if M is a pair (M_1, M_2); otherwise it behaves like $\mathbf{0}$. $\mathbf{match}\ M_1\ \mathbf{is}\ (M_2, y).P$ behaves like $[M_3/y]P$ if M_1 is a pair of the form (M_2, M_3); otherwise it behaves like $\mathbf{0}$. Process $\mathbf{decrypt}\ M_1\ \mathbf{is}\ \{x\}_{M_2}.P$ ($\mathbf{decrypt}\ M_1\ \mathbf{is}\ \{\!|x|\!\}_{M_2^{-1}}.P$, resp.) decrypts ciphertext M_1 with symmetric (asymmetric, resp.) key M_2, binds x to the result and behaves like P; if M_1 is not an encryption, or an encryption with a key not matching M_2, then it behaves like $\mathbf{0}$. The process $\mathbf{begin}\, M.P$ raise an event $\mathbf{begin}\, M$ and behaves like P, while $\mathbf{end}\, M$ just raises an event $\mathbf{end}\, M$; they are used to express expected authenticity properties.

Example 1. We use the three protocols in Figure 1, taken from [15], as running examples. POSH and SOSH protocols aim to pass a new message `msg` from B to A, so

POSH:	**SOPH**	**SOSH**
`A->B: n`	`A->B:` $\{\vert(\mathtt{msg},\mathtt{n})\vert\}_{pk_B}$	`A->B:` $\{\vert\mathtt{n}\vert\}_{pk_B}$
`B begins msg`	`B begins msg`	`B begins msg`
`B->A:` $\{\vert(\mathtt{msg},\mathtt{n})\vert\}_{sk_B}$	`B->A: n`	`B->A:` $\{\vert\mathtt{msg},\mathtt{n}\vert\}_{pk_A}$
`A ends msg`	`A ends msg`	`A ends msg`

Fig. 1. Informal Description of Three Protocols

$(\nu_{asym} sk_B, pk_B)(net!pk_B \mid$	(* create asymmetric keys for B and make pk_B public *)
$(\nu non)(net!non \mid$	(* A creates a nonce and sends it *)
$net?ctext.\mathbf{decrypt}\ ctext\ \mathbf{is}\ \{\vert x\vert\}_{pk_B^{-1}}.$	(* receive a cypertext and decrypt it*)
$\mathbf{split}\ x\ \mathbf{is}\ (m, non').\mathbf{check}\ non\ \mathbf{is}\ non'.$	(* decompose pair x and check nonce *)
$\mathbf{end}\ m) \mid$	(* believe that m came from B *)
$net?n.$	(* B receives a nonce *)
$(\nu msg)\mathbf{begin}\ msg.$	(* create a message and declare that it is going to be sent*)
$net!\{\vert(msg, n)\vert\}_{sk_B})$	(* encrypt and send (msg, n) *)

Fig. 2. Public-Out-Secret-Home (POSH) protocol in spi$_{CA}$

that A can confirm that `msg` indeed comes from B, while SOPH protocol aims to pass `msg` from A to B, so that A can confirm that `msg` has been received by B. The second and fourth lines of each protocol expresses the required authenticity by using Woo and Lam's correspondence assertions [20]. "`B begins msg`" on the second line of POSH means "B is going to send `msg`", and "`A ends msg`" on the fourth line means "A believes that B has sent `msg`". The required authenticity is then expressed as a correspondence between begin- and end-events: whenever an end-event ("`A ends msg`" in this example) occurs, the corresponding begin-event ("`B begins msg`") must have occurred.[3] In the three protocols, the correspondence between begin- and end-events is guaranteed in different ways. In POSH, the correspondence is guaranteed by the signing of the second message with B's secret key, so that A can verify that B has created the pair (msg, n). In SOPH, it is guaranteed by encrypting the first message with B's public key, so that the nonce n, used as an acknowledgment, cannot be forged by an attacker. SOSH is similar to POSH, but keeps `n` secret by using A and B's public keys.

Figure 2 gives a formal description of POSH protocol, represented as a process in spi$_{CA}$. The first line is an initial set-up for the protocol. An asymmetric key pair for B is created and the decryption key pk_B is sent on a public channel *net*, on which an attacker can send and receive messages. The next four lines describe the behavior of A. On the second line, a nonce *non* is created and sent along *net*. On the third line, a ciphertext *ctext* is received and decrypted (or verified) with B's public key. On the fourth line, the pair is decomposed and it is checked that the second component coincides with the nonce sent before. On the fifth line, an end-event is raised, meaning that A believes that

[3] There are two types of correspondence assertions in the literature: non-injective (or one-to-many) and injective (or one-to-one) correspondence. Throughout the paper we consider the latter.

msg came from B. The last three lines describe the behavior of B. On the sixth line, a nonce n is received from *net*. On the seventh line, a new message *msg* is created and a begin-event is raised, meaning that B is going to send *msg*. On the last line, the pair (msg, n) is encrypted (or signed) with B's secret key and sent on *net*. $\square$

Following Gordon and Jeffrey, we call a process *safe* if it satisfies correspondence assertions (i.e. for each end-event, a corresponding begin-event has occurred before), and *robustly safe* if a process is safe in the presence of arbitrary attackers (representable in spi$_{CA}$). Proving robust safety automatically is the goal of protocol verification in the present paper. To formalize the robust safety, we use the operational semantics shown in Figure 3. A runtime state is a quadruple $\langle \Psi, E, N, \mathcal{K}\rangle$, where Ψ is a multiset of processes, and E is the set of messages on which begin-events have occurred but the matching end-events have not. N is the set of names (including keys) created so far, and $\mathcal{K}$ is the set of key pairs. The special runtime state **Error** denotes that correspondence assertions have been violated. Note that a reduction gets stuck when a process does not match a rule. For example, **split** M **is** $(x, y).P$ is reducible only if M is of the form (M_1, M_2). Using the operational semantics, the robust safety is defined as follows.

$$\langle \Psi \uplus \{n?y.P, n!M\}, E, N, \mathcal{K}\rangle \longrightarrow \langle \Psi \uplus \{[M/y]P\}, E, N, \mathcal{K}\rangle \qquad \text{(R-COM)}$$

$$\langle \Psi \uplus \{P \mid Q\}, E, N, \mathcal{K}\rangle \longrightarrow \langle \Psi \uplus \{P, Q\}, E, N, \mathcal{K}\rangle \qquad \text{(R-PAR)}$$

$$\langle \Psi \uplus \{*P\}, E, N, \mathcal{K}\rangle \longrightarrow \langle \Psi \uplus \{*P, P\}, E, N, \mathcal{K}\rangle \qquad \text{(R-REP)}$$

$$\langle \Psi \uplus \{(\nu x)P\}, E, N, \mathcal{K}\rangle \longrightarrow \langle \Psi \uplus \{[n/x]P\}, E, N \cup \{n\}, \mathcal{K}\rangle \ (n \notin N) \qquad \text{(R-NEW)}$$

$$\langle \Psi \uplus \{(\nu_{sym} x)P\}, E, N, \mathcal{K}\rangle \longrightarrow \langle \Psi \uplus \{[k/x]P\}, E, N \cup \{k\}, \mathcal{K}\rangle \ (k \notin N) \qquad \text{(R-NEWSK)}$$

$$\begin{array}{l}\langle \Psi \uplus \{(\nu_{asym} x, y)P\}, E, N, \mathcal{K}\rangle \\ \longrightarrow \langle \Psi \uplus \{[k_1/x, k_2/y]P\}, E, N \cup \{k_1, k_2\}, \mathcal{K} \cup \{(k_1, k_2)\}\rangle \ (k_1, k_2 \notin N)\end{array} \qquad \text{(R-NEWAK)}$$

$$\langle \Psi \uplus \{\mathbf{check}\ n\ \mathbf{is}\ n.P\}, E, N, \mathcal{K}\rangle \longrightarrow \langle \Psi \uplus \{P\}, E, N, \mathcal{K}\rangle \qquad \text{(R-CHK)}$$

$$\langle \Psi \uplus \{\mathbf{split}\ (M, N)\ \mathbf{is}\ (x, y).P\}, E, N, \mathcal{K}\rangle \longrightarrow \langle \Psi \uplus \{[M/x, N/y]P\}, E, N, \mathcal{K}\rangle \qquad \text{(R-SPLT)}$$

$$\langle \Psi \uplus \{\mathbf{match}\ (M, N)\ \mathbf{is}\ (M, z).P\}, E, N, \mathcal{K}\rangle \longrightarrow \langle \Psi \uplus \{[N/z]P\}, E, N, \mathcal{K}\rangle \qquad \text{(R-MTCH)}$$

$$\langle \Psi \uplus \{\mathbf{decrypt}\ \{M\}_k\ \mathbf{is}\ \{x\}_k.P\}, E, N, \mathcal{K}\rangle \longrightarrow \langle \Psi \uplus \{[M/x]P\}, E, N, \mathcal{K}\rangle \qquad \text{(R-DECS)}$$

$$\begin{array}{l}\langle \Psi \uplus \{\mathbf{decrypt}\ \{\!|M|\!\}_{k_1}\ \mathbf{is}\ \{\!|x|\!\}_{k_2^{-1}}.P\}, E, N, \mathcal{K}\rangle \\ \longrightarrow \langle \Psi \uplus \{[M/x]P\}, E, N, \mathcal{K}\rangle \ (\text{if } (k_1, k_2) \in \mathcal{K})\end{array} \qquad \text{(R-DECA)}$$

$$\langle \Psi \uplus \{\mathbf{begin}\, M.P\}, E, N, \mathcal{K}\rangle \longrightarrow \langle \Psi \uplus \{P\}, E \uplus \{M\}, N, \mathcal{K}\rangle \qquad \text{(R-BGN)}$$

$$\langle \Psi \uplus \{\mathbf{end}\, M\}, E \uplus \{M\}, N, \mathcal{K}\rangle \longrightarrow \langle \Psi, E, N, \mathcal{K}\rangle \qquad \text{(R-END)}$$

$$\langle \Psi \uplus \{\mathbf{end}\, M\}, E, N, \mathcal{K}\rangle \longrightarrow \mathbf{Error} \quad (\text{if } M \notin E) \qquad \text{(R-ERR)}$$

Fig. 3. Operational Semantics

Definition 21 (safety, robust safety) *A process P is* safe *if* $\langle\{P\}, \emptyset, \mathbf{FN}(P), \emptyset\rangle \not\longrightarrow^* \mathbf{Error}$. *A process P is* robustly safe *if $P|O$ is safe for every spi$_{CA}$ process O that contains no begin/end/check operations.*[4]

3 Type System

This section presents a type system such that well-typed processes are robustly safe. This allows us to reduce protocol verification to type inference.

3.1 Basic Ideas

Following the previous work [14, 15, 18], we use the notion of *capabilities* (called effects in [14, 15]) in order to statically guarantee that end-events can be raised only after the corresponding begin-events. A capability φ is a multiset of *atomic capabilities* of the form $\mathbf{end}(M)$, which expresses a permission to raise "end M" event. The robust safety of processes is guaranteed by enforcing the following conditions on capabilities: (i) to raise an "end M" event, a process must possess and consume an atomic $\mathbf{end}(M)$ capability; and (ii) an atomic $\mathbf{end}(M)$ capability is generated only by raising a "begin M" event. Those conditions can be statically enforced by using a type judgment of the form: $\Gamma; \varphi \vdash P$, which means that P can be safely executed under the type environment Γ and the capabilities described by φ. For example, $x\!:\!T; \{\mathbf{end}(x)\} \vdash \mathbf{end}\, x$ is a valid judgment, but $x\!:\!T; \emptyset \vdash \mathbf{end}\, x$ is not. The two conditions above can be locally enforced by the following typing rules for begin and end events:

$$\frac{\Gamma; \varphi + \{\mathbf{end}(M)\} \vdash P}{\Gamma; \varphi \vdash \mathbf{begin}\, M.P} \qquad\qquad \frac{}{\Gamma; \varphi + \{\mathbf{end}(M)\} \vdash \mathbf{end}\, M}$$

The left rule ensures that the new capability $\mathbf{end}(M)$ is available after the begin-event, and the right rule for end ensures that the capability $\mathbf{end}(M)$ must be present.

The main difficulty lies in how to pass capabilities between processes. For example, recall the POSH protocol in Figure 2, where begin- and end-events are raised by different protocol participants. The safety of this protocol can be understood as follows: B obtains the capability $\mathbf{end}(\mathit{msg})$ by raising the begin event, and then passes the capability to A by attaching it to the nonce n. A then extracts the capability and safely executes the end event. As n is signed with B's private key, there is no way for an attacker to forge the capability. For another example, consider the SOPH protocol in the middle of Figure 1. In this case, the nonce n is sent in clear text, so that B cannot pass the capability to A through the second message. Instead, the safety of the SOPH protocol is understood as follows: A attaches to n (in the first message) an *obligation* to raise the begin-event. B then discharges the obligation by raising the begin-event, and notifies of it by sending back n. Here, note that an attacker cannot forge n, as it is encrypted by B's public key in the first message.

[4] Having no check operations is not a limitation, as an attacker process can check the equality of n_1 and n_2 by $\mathbf{match}\ (n_1, n_1)\ \mathbf{is}\ (n_2, x).P$.

To capture the above reasoning by using types, we introduce types of the form $\mathbf{N}(\varphi_1, \varphi_2)$, which describes names carrying an obligation φ_1 and a capability φ_2. In the examples above, n is given the type $\mathbf{N}(\emptyset, \{\mathbf{end}(\mathit{msg})\})$ in the second message of POSH protocol, and the type $\mathbf{N}(\{\mathbf{end}(\mathit{msg})\}, \emptyset)$ in the first message of SOPH protocol.

The above types $\mathbf{N}(\emptyset, \{\mathbf{end}(\mathit{msg})\})$ and $\mathbf{N}(\{\mathbf{end}(\mathit{msg})\}, \emptyset)$ respectively correspond to *response* and *challenge types* in Gordon and Jeffrey's type system [15]. Thanks to the uniform treatment of name types, type inference for our type system reduces to a problem of solving constraints on capabilities and obligations, which can further be reduced to linear programming problems by using the technique of [18]. The uniform treatment also allows us to express a wider range of protocols (such as multi-party cryptographic protocols). Note that neither obligations nor asymmetric cryptography are supported by the previous type system for automated verification [18]; handling them requires non-trivial extensions of the type system and the inference algorithm.

3.2 Types

Definition 31 *The syntax of types, ranged over by τ, is given by:*

$$\begin{array}{lr}
\tau ::= \mathbf{N}_\ell(\varphi_1, \varphi_2) \mid \mathbf{SKey}(\tau) \mid \mathbf{DKey}(\tau) \mid \mathbf{EKey}(\tau) \mid \tau_1 \times \tau_2 & \\
\varphi ::= \{A_1 \mapsto r_1, \ldots, A_m \mapsto r_m\} & \textit{capabilities} \\
A ::= \mathbf{end}(M) \mid \mathbf{chk}_\ell(M, \varphi) & \textit{atomic cap.} \\
\iota ::= x \mid 0 \mid 1 \mid 2 \mid \cdots & \textit{extended names} \\
\ell ::= \mathbf{Pub} \mid \mathbf{Pr} & \textit{name qualifiers}
\end{array}$$

Here, r_i ranges over non-negative rational numbers.

The type $\mathbf{N}_\ell(\varphi_1, \varphi_2)$ is assigned to names carrying obligations φ_1 and capabilities φ_2. Here, obligations and capabilities are mappings from atomic capabilities to rational numbers. For example, $\mathbf{N}_\ell(\{\mathbf{end}(a) \mapsto 1.0\}, \{\mathbf{end}(b) \mapsto 2.0\})$ describes a name that carries the obligation to raise $\mathbf{begin}\, a$ once, and the capability to raise $\mathbf{end}\, b$ twice. Fractional values are possible: $\mathbf{N}_\ell(\emptyset, \{\mathbf{end}(b) \mapsto 0.5\})$ means that the name carries a half of the capability to raise $\mathbf{end}\, b$, so that if combined with another half of the capability, it is allowed to raise $\mathbf{end}\, b$. The introduction of fractions slightly increases the expressive power of the type system, but the main motivation for it is rather to enable efficient type inference as in [18]. When the ranges of obligations and capabilities are integers, we often use multiset notations; for example, we write $\{\mathbf{end}(a), \mathbf{end}(a), \mathbf{end}(b)\}$ for $\{\mathbf{end}(a) \mapsto 2, \mathbf{end}(b) \mapsto 1\}$. The atomic capability $\mathbf{chk}_\ell(M, \varphi)$ expresses the capability to check equality on M by $\mathbf{check}\ M\ \mathbf{is}\ M'.P$: since nonce checking releases capabilities this atomic effect is used to ensure that each nonce can only be checked once. The component φ expresses the capability that can be extracted by the check operation (see the typing rule for check operations given later).

Qualifier ℓ attached to name types are essentially the same as the **Public**/**Private** qualifiers in Gordon and Jeffrey's type system and express whether a name can be made public or not. We often write $\mathbf{Un}$ for $\mathbf{N}_{\mathbf{Pub}}(\emptyset, \emptyset)$.

The type $\mathbf{SKey}(\tau)$ describes symmetric keys used for decrypting and encrypting values of type τ. The type $\mathbf{EKey}(\tau)$ ($\mathbf{DKey}(\tau)$, resp.) describes asymmetric keys used for encrypting (decrypting, resp.) values of type τ. The type $\tau_1 \times \tau_2$ describes pairs of values of types τ_1 and τ_2. As in [18], we express the dependency of types on names by using indices. For example, the type $\mathbf{Un} \times \mathbf{N}_\ell(\emptyset, \{\mathbf{end}(0)\})$ denotes a pair (M_1, M_2) where M_1 has type $\mathbf{Un}$ and M_2 has type $\mathbf{N}_\ell(\emptyset, \{\mathbf{end}(M_1)\})$. The type $\mathbf{Un}\times(\mathbf{Un}\times\mathbf{N}_{\mathbf{Pub}}(\emptyset, \{\mathbf{end}(0,1) \mapsto r\})$ describes triples of the form $(M_1, (M_2, M_3))$, where M_1 and M_2 have type $\mathbf{Un}$, and M_3 has type $\mathbf{N}_{\mathbf{Pub}}(\emptyset, \{\mathbf{end}(M_2, M_1) \mapsto r\})$. In general, an index i is a natural number referring to the i-th closest first component of pairs. In the syntax of atomic capabilities $\mathbf{end}(M)$, M is an extended message that may contain indices. We use the same metavariable M for the sake of simplicity.

Predicates on types Following Gordon and Jeffrey, we introduce two predicates $\mathbf{Pub}$ and $\mathbf{Taint}$ on types, inductively defined by the rules in Figure 4. $\mathbf{Pub}(\tau)$ means that a value of type τ can safely be made public by e.g. sending it through a public channel. $\mathbf{Taint}(\tau)$ means that a value of type τ may have come from an untrusted principal and hence cannot be trusted. It may for instance have been received through a public channel or have been extracted from a ciphertext encrypted with a public key.

The first rule says that for $\mathbf{N}_\ell(\varphi_1, \varphi_2)$ to be public, the obligation φ_1 must be empty, as there is no guarantee that an attacker fulfills the obligation. Contrary, for $\mathbf{N}_\ell(\varphi_1, \varphi_2)$ to be tainted, the capability φ_2 must be empty if $\ell = \mathbf{Pub}$, as the name may come from an attacker and the capability cannot be trusted.[5]

$\mathbf{Pub}$ and $\mathbf{Taint}$ are a sort of dual, flipped by the type constructor $\mathbf{EKey}$. In terms of subtyping, $\mathbf{Pub}(\tau)$ and $\mathbf{Taint}(\tau)$ may be understood as $\tau \leq \mathbf{Un}$ and $\mathbf{Un} \leq \tau$ respectively, where $\mathbf{Un}$ is the type of untrusted, non-secret data. Note that $\mathbf{DKey}$ is co-variant, $\mathbf{EKey}$ is contra-variant, and $\mathbf{SKey}$ is invariant; this is analogous to Pierce and Sangiorgi's IO types with subtyping [19].

$$\frac{\ell = \mathbf{Pub} \qquad \varphi_1 = \emptyset}{\mathbf{Pub}(\mathbf{N}_\ell(\varphi_1, \varphi_2))} \qquad \frac{\ell = \mathbf{Pub} \Rightarrow \varphi_2 = \emptyset}{\mathbf{Taint}(\mathbf{N}_\ell(\varphi_1, \varphi_2))} \qquad \frac{\mathbf{Pub}(\tau_1) \qquad \mathbf{Pub}(\tau_2)}{\mathbf{Pub}(\tau_1 \times \tau_2)}$$

$$\frac{\mathbf{Taint}(\tau_1) \qquad \mathbf{Taint}(\tau_2)}{\mathbf{Taint}(\tau_1 \times \tau_2)} \qquad \frac{\mathbf{Pub}(\tau) \qquad \mathbf{Taint}(\tau)}{\mathbf{Pub}(\mathbf{SKey}(\tau))} \qquad \frac{\mathbf{Pub}(\tau) \qquad \mathbf{Taint}(\tau)}{\mathbf{Taint}(\mathbf{SKey}(\tau))}$$

$$\frac{\mathbf{Taint}(\tau)}{\mathbf{Pub}(\mathbf{EKey}(\tau))} \qquad \frac{\mathbf{Pub}(\tau)}{\mathbf{Taint}(\mathbf{EKey}(\tau))} \qquad \frac{\mathbf{Pub}(\tau)}{\mathbf{Pub}(\mathbf{DKey}(\tau))} \qquad \frac{\mathbf{Taint}(\tau)}{\mathbf{Taint}(\mathbf{DKey}(\tau))}$$

Fig. 4. Predicates $\mathbf{Pub}$ and $\mathbf{Taint}$

[5] These conditions are more liberal than the corresponding conditions in Gordon and Jeffrey's type system. In their type system, for Public Challenge φ_1 (which corresponds to $\mathbf{N}_{\mathbf{Pub}}(\varphi_1, \emptyset)$ in our type system) to be tainted, φ_1 must also be empty.

Operations and relations on capabilities and types We write $dom(\varphi)$ for the set $\{A \mid \varphi(A) > 0\}$. We identify capabilities up to the following equality $\approx$:

$$\varphi_1 \approx \varphi_2 \iff (dom(\varphi_1) = dom(\varphi_2) \wedge \forall A \in dom(\varphi_1).\varphi_1(A) = \varphi_2(A)).$$

We write $\varphi \leq \varphi'$ if $\varphi(A) \leq \varphi'(A)$ holds for every $A \in dom(\varphi)$ and we define the summation of two capabilities by: $(\varphi_1 + \varphi_2)(A) = \varphi_1(A) + \varphi_2(A)$. This is a natural extension of the multiset union. We write $\varphi_1 - \varphi_2$ for the least φ such that $\varphi_1 \leq \varphi + \varphi_2$.

As we use indices to express dependent types, messages may be substituted in types. Let i be an index and M a message. The substitution $[M/i]\tau$ is defined inductively in the straight-forward manner, except for pair types where

$$[M/i](\tau_1 \times \tau_2) = ([M/i]\tau_1) \times ([M/(i+1)]\tau),$$

such that the index is shifted for the second component.

3.3 Typing

We introduce two forms of type judgments: $\Gamma; \varphi \vdash M : \tau$ for messages, and $\Gamma; \varphi \vdash P$ for processes, where Γ, called a type environment, is a sequence of type bindings of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$. Judgment $\Gamma; \varphi \vdash M : \tau$ means that M evaluates to a value of type τ under the assumption that each name has the type described by Γ and that capability φ is available. $\Gamma; \varphi \vdash P$ means that P can be safely executed (i.e. without violation of correspondence assertions) if each free name has the type described by Γ and the capability φ is available. For example, $x : \mathbf{Un}; \{\mathbf{end}(x)\} \vdash \mathbf{end}\, x$ is valid but $x : \mathbf{Un}; \emptyset \vdash \mathbf{end}\, x$ is not.

We consider only the judgements that are *well-formed* in the sense that (i) φ refers to only the names bound in Γ, and (ii) Γ must be well-formed, i.e., if Γ is of the form $\Gamma_1, x : \tau, \Gamma_2$ then τ only refers to the names bound in Γ_1 and x is not bound in neither Γ_1 nor Γ_2. See [10] for the formal definition of the well-formedness of type environments and judgments. We freely permute bindings in type environments as long as they are well-formed; for example, we do not distinguish between $x : \mathbf{Un}, y : \mathbf{Un}$ and $y : \mathbf{Un}, x : \mathbf{Un}$.

Typing The typing rules are shown in Figure 5. The rule T-CAST says that the current capability can be used for discharging obligations and increasing capabilities of the name. T-CAST plays a role similar to the typing rule for cast processes in Gordon and Jeffrey's type system, but our cast is implicit and changes only the capabilities and obligations, not the shape of types. This difference is important for automated type inference. The other rules for messages are standard; T-PAIR is the standard rule for dependent sum types (except for the use of indices).

In the rules for processes, the capabilities shown by _ can be any capabilities. The rules are also similar to those of Gordon and Jeffrey, except for the rules T-OUT, T-IN, T-NEWN, and T-CHK. In rule T-OUT, we require that the type of message M_2 is public as it can be received by any process, including the attacker. Similarly, in rule T-IN we require that the type of the received value x is tainted, as it may come from any process.

$$\frac{}{\Gamma, x:\tau;\varphi \vdash x:\tau}\ \text{(T-VAR)} \qquad \frac{\Gamma;\varphi_1 \vdash M_1 : \tau_1 \qquad \Gamma;\varphi_2 \vdash M_2 : [M_1/0]\tau_2}{\Gamma;\varphi_1+\varphi_2 \vdash (M_1, M_2) : \tau_1 \times \tau_2}\ \text{(T-PAIR)}$$

$$\frac{\Gamma;\varphi_1 \vdash M_1 : \tau_1 \quad \Gamma;\varphi_2 \vdash M_2 : \mathbf{SKey}(\tau_1)}{\Gamma;\varphi_1+\varphi_2 \vdash \{M_1\}_{M_2} : \mathbf{N}_\ell(\emptyset,\emptyset)}\ \text{(T-SENC)} \qquad \frac{\Gamma;\varphi_1 \vdash M_1 : \tau \quad \Gamma;\varphi_2 \vdash M_2 : \mathbf{EKey}(\tau)}{\Gamma;\varphi_1+\varphi_2 \vdash \{\!|M_1|\!\}_{M_2} : \mathbf{N}_\ell(\emptyset,\emptyset)}\ \text{(T-AENC)}$$

$$\frac{\Gamma;\varphi_1 \vdash M : \mathbf{N}_\ell(\varphi_2,\varphi_3)}{\Gamma;\varphi_1+\varphi_2'+\varphi_3' \vdash M : \mathbf{N}_\ell(\varphi_2-\varphi_2', \varphi_3+\varphi_3')}\ \text{(T-CAST)}$$

$$\frac{}{\Gamma;\emptyset \vdash \mathbf{0}}\ \text{(T-ZERO)} \qquad \frac{\Gamma;\varphi_1 \vdash P_1 \qquad \Gamma;\varphi_2 \vdash P_2}{\Gamma;\varphi_1+\varphi_2 \vdash P_1 \mid P_2}\ \text{(T-PAR)} \qquad \frac{\Gamma;\emptyset \vdash P}{\Gamma;\emptyset \vdash *P}\ \text{(T-REP)} \qquad \frac{\Gamma;\varphi' \vdash P \qquad \varphi' \leq \varphi}{\Gamma;\varphi \vdash P}\ \text{(T-CSUB)}$$

$$\frac{\begin{array}{c}\Gamma;\varphi_1 \vdash M_1 : \mathbf{N}_\ell(\emptyset,\emptyset)\\ \Gamma;\varphi_2 \vdash M_2 : \tau \qquad \mathbf{Pub}(\tau)\end{array}}{\Gamma;\varphi_1+\varphi_2 \vdash M_1!M_2}\ \text{(T-OUT)} \qquad \frac{\begin{array}{c}\Gamma;\varphi_1 \vdash M : \mathbf{N}_\ell(\emptyset,\emptyset)\\ \Gamma, x:\tau;\varphi_2 \vdash P \qquad \mathbf{Taint}(\tau)\end{array}}{\Gamma;\varphi_1+\varphi_2 \vdash M?x.P}\ \text{(T-IN)} \qquad \frac{\Gamma, x:\mathbf{SKey}(\tau);\varphi \vdash P}{\Gamma;\varphi \vdash (\nu_{sym} x)P}\ \text{(T-NEWSK)}$$

$$\frac{\Gamma, x:\mathbf{N}_\ell(\varphi_1,\emptyset), \varphi + \{\mathbf{chk}_\ell(x,\varphi_1)\} \vdash P}{\Gamma;\varphi \vdash (\nu x)P}\ \text{(T-NEWN)} \qquad \frac{\Gamma, k_1:\mathbf{EKey}(\tau), k_2:\mathbf{DKey}(\tau);\varphi \vdash P}{\Gamma;\varphi \vdash (\nu_{asym} k_1, k_2)P}\ \text{(T-NEWAK)}$$

$$\frac{\Gamma;\varphi_1 \vdash M_1 : \mathbf{N}_\ell(\_,\_) \qquad \Gamma;\varphi_2 \vdash M_2 : \mathbf{SKey}(\tau) \qquad \Gamma, x:\tau;\varphi_3 \vdash P}{\Gamma;\varphi_1+\varphi_2+\varphi_3 \vdash \mathbf{decrypt}\ M_1\ \mathbf{is}\ \{x\}_{M_2}.P}\ \text{(T-SDEC)}$$

$$\frac{\Gamma;\varphi_1 \vdash M_1 : \mathbf{N}_\ell(\_,\_) \qquad \Gamma;\varphi_2 \vdash M_2 : \mathbf{DKey}(\tau) \qquad \Gamma, x:\tau;\varphi_3 \vdash P}{\Gamma;\varphi_1+\varphi_2+\varphi_3 \vdash \mathbf{decrypt}\ M_1\ \mathbf{is}\ \{\!|x|\!\}_{M_2^{-1}}.P}\ \text{(T-ADEC)}$$

$$\frac{\Gamma;\varphi_1 \vdash M_1 : \mathbf{N}_\ell(\_,\_) \qquad \Gamma;\varphi_2 \vdash M_2 : \mathbf{N}_\ell(\emptyset,\varphi_5) \qquad \Gamma;\varphi_3+\varphi_4+\varphi_5 \vdash P}{\Gamma;\varphi_1+\varphi_2+\varphi_3+\{\mathbf{chk}_\ell(M_1,\varphi_4)\} \vdash \mathbf{check}\ M_1\ \mathbf{is}\ M_2.P}\ \text{(T-CHK)}$$

$$\frac{\Gamma;\varphi_1 \vdash M : \tau_1 \times \tau_2 \qquad \Gamma, y:\tau_1, z:[y/0]\tau_2;\varphi_2 \vdash P}{\Gamma;\varphi_1+\varphi_2 \vdash \mathbf{split}\ M\ \mathbf{is}\ (y,z).P}\ \text{(T-SPLIT)}$$

$$\frac{\Gamma;\varphi_1 \vdash M_1 : \tau_1 \times \tau_2 \qquad \Gamma;\varphi_2 \vdash M_2 : \tau_1 \qquad \Gamma, z:[M_2/0]\tau_2;\varphi_3 \vdash P}{\Gamma;\varphi_1+\varphi_2+\varphi_3 \vdash \mathbf{match}\ M_1\ \mathbf{is}\ (M_2,z).P}\ \text{(T-MATCH)}$$

$$\frac{\Gamma;\varphi + \{\mathbf{end}(M)\} \vdash P}{\Gamma;\varphi \vdash \mathbf{begin}\, M.P}\ \text{(T-BEGIN)} \qquad \frac{}{\Gamma;\varphi + \{\mathbf{end}(M)\} \vdash \mathbf{end}\, M}\ \text{(T-END)}$$

Fig. 5. Typing Rules

This is different from Gordon and Jeffrey's type system where the type of messages sent to or received from public channels must be $\mathbf{Un}$, and a subsumption rule allows any value of a public type to be typed as $\mathbf{Un}$ and a value of type $\mathbf{Un}$ to be typed as any tainted type. In effect, our type system can be considered a restriction of Gordon and Jeffrey's such that the subsumption rule is only allowed for messages sent or received via public channels. This point is important for automated type inference.

In rule T-NEWN, the obligation φ_1 is attached to the fresh name x and recorded in the atomic check capability. Capabilities corresponding to φ_1 can then later be extracted by a check operation if the obligation has been fulfilled. In rule T-CHK, $\mathbf{chk}_\ell(M_1, \varphi_4)$ in the conclusion means that the capability to check M_1 must be present. If the check succeeds, the capability φ_5 attached to M_2 can be extracted and used in P. In addition, the obligations attached to M_2 must be empty, i.e. all obligations initially attached to the name must have been fulfilled, and hence the capability φ_4 can be extracted and used in P. The above mechanism for extracting capabilities through obligations is different from Gordon and Jeffrey's type system in a subtle but important way, and provides more expressive power: see [10]. The remaining rules should be self-explanatory.

The following theorem guarantees the soundness of the type system. The proof is given in the full version [10].

Theorem 1 (soundness). *If $x_1 : \mathbf{Un}, \dots, x_m : \mathbf{Un}; \emptyset \vdash P$, then P is robustly safe.*

Example 2. Recall the POSH protocol in Figure 2. Let τ be $\mathbf{Un} \times \mathbf{N}_{\mathbf{Pub}}(\emptyset, \{\mathbf{end}(0)\})$. Then the process describing the behavior of B ($net?n.\cdots$ in the last five lines) is typed as the upper part of Figure 6. Here, $\Gamma = \mathit{net} : \mathbf{Un}, \mathit{sk}_B : \mathbf{EKey}(\tau), n : \mathbf{Un}, \mathit{msg} : \mathbf{Un}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; \emptyset \vdash \mathit{msg} : \mathbf{Un} \quad \dfrac{\Gamma; \emptyset \vdash n : \mathbf{N}_{\mathbf{Pub}}(\emptyset, \emptyset)}{\Gamma; \{\mathbf{end}(\mathit{msg})\} \vdash n : \mathbf{N}_{\mathbf{Pub}}(\emptyset, \{\mathbf{end}(\mathit{msg})\})}}{\Gamma; \{\mathbf{end}(\mathit{msg})\} \vdash (\mathit{msg}, n) : \tau}}{\cdots}}{\Gamma; \{\mathbf{end}(\mathit{msg}), \mathbf{chk}_{\mathbf{Pub}}(\mathit{msg}, \emptyset)\} \vdash \mathit{net}!\{\!|(\mathit{msg}, n)|\!\}_{\mathit{sk}_B}}}{\Gamma; \{\mathbf{chk}_{\mathbf{Pub}}(\mathit{msg}, \emptyset)\} \vdash \mathbf{begin}\, \mathit{msg}.\cdots}}{\mathit{net} : \mathbf{Un}, \mathit{sk}_B : \mathbf{EKey}(\tau), n : \mathbf{Un}; \emptyset \vdash (\nu \mathit{msg}) \cdots}}{\mathit{net} : \mathbf{Un}, \mathit{sk}_B : \mathbf{EKey}(\tau); \emptyset \vdash \mathit{net}?n.\cdots}$$

$$\dfrac{\dfrac{\dfrac{\Gamma_3; \{\mathbf{end}(m)\} \vdash \mathbf{end}\, m}{\Gamma_3; \{\mathbf{chk}_{\mathbf{Pub}}(\mathit{non}, \emptyset)\} \vdash \mathbf{check}\ \mathit{non}\ \mathbf{is}\ \mathit{non}'.\cdots}}{\Gamma_2, x : \tau; \{\mathbf{chk}_{\mathbf{Pub}}(\mathit{non}, \emptyset)\} \vdash \mathbf{split}\ x\ \mathbf{is}\ (m, \mathit{non}).\cdots}}{\Gamma_2; \{\mathbf{chk}_{\mathbf{Pub}}(\mathit{non}, \emptyset)\} \vdash \mathbf{decrypt}\ \mathit{ctext}\ \mathbf{is}\ \{\!|x|\!\}_{\mathit{pk}_B^{-1}}.\cdots}$$

Fig. 6. Partial Typing of the POSH Protocol

Similarly, the part $\mathbf{decrypt}\ \mathit{ctext}\ \mathbf{is}\ \{\!|x|\!\}_{\mathit{pk}_B^{-1}}.\cdots$ of process A is typed as the lower part of Figure 6. Here, $\Gamma_2 = \mathit{net} : \mathbf{Un}, \mathit{pk}_B : \mathbf{DKey}(\tau), \mathit{non} : \mathbf{Un}, \mathit{ctext} : \mathbf{Un}$ and $\Gamma_3 = \Gamma_2, x : \tau, m : \mathbf{Un}, \mathit{non}' : \mathbf{N}_{\mathbf{Pub}}(\emptyset, \{\mathbf{end}(m)\})$. Let P_1 be the entire process of the POSH protocol. It is typed by $\mathit{net} : \mathbf{Un}; \emptyset \vdash P_1$.

The SOPH and SOSH protocols in Figure 1 are typed in a similar manner. We show here only key types:

SOPH
$pk_B : \mathbf{EKey}(\mathbf{Un} \times \mathbf{N}_{\mathbf{Pub}}(\{\mathbf{end}(0)\}, \emptyset)), sk_B : \mathbf{DKey}(\mathbf{Un} \times \mathbf{N}_{\mathbf{Pub}}(\{\mathbf{end}(0)\}, \emptyset))$

SOSH
$pk_A : \mathbf{EKey}(\mathbf{Un} \times \mathbf{N}_{\mathbf{Pr}}(\emptyset, \{\mathbf{end}(0)\})), sk_A : \mathbf{DKey}(\mathbf{Un} \times \mathbf{N}_{\mathbf{Pr}}(\emptyset, \{\mathbf{end}(0)\}))$
$pk_B : \mathbf{EKey}(\mathbf{Un} \times \mathbf{N}_{\mathbf{Pr}}(\emptyset, \emptyset)), \quad sk_B : \mathbf{DKey}(\mathbf{Un} \times \mathbf{N}_{\mathbf{Pr}}(\emptyset, \emptyset))$

Note that for POSH and SOPH the name qualifier must be $\mathbf{Pub}$, and only for the SOSH protocol may it be $\mathbf{Pr}$. $\square$

4 Type Inference

We now briefly discuss type inference. For this we impose a minor restriction to the type system, namely that in rule T-PAIR, if M_1 is not a name then the indice 0 cannot occur in τ_2. Similarly, in rule T-MATCH we require that index 0 does not occur unless M_2 is a name. These restrictions prevent the size of types and capabilities from blowing up. Given as input a process P with free names $x_1, \ldots, x_n$, the algorithm to decide $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; \emptyset \vdash P$ proceeds as follows:

1. Determine the *shape of the type* (or simple type) of each term via a standard unification algorithm, and construct a template of a type derivation tree by introducing qualifier and capability variables.
2. Generate a set C of constraints on qualifier and capability variables based on the typing rules such that C is satisfiable if and only if $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; \emptyset \vdash P$.
3. Solve the qualifier constraints.
4. Transform the capability constraints to linear inequalities over the rational numbers.
5. Use linear programming to determine if the linear inequalities are satisfiable.

In step 1, we can assume that there are no consecutive applications of T-CAST and T-CSUB. Thus, the template of a type derivation tree can be uniquely determined: for each process and message constructor there is an application of the rule matching the constructor followed by at most one application of T-CAST or T-CSUB.

At step 3 we have a set of constraints C of the form:

$$\{\ell_i = \ell'_i \mid i \in I\} \cup \{(\ell''_j = \mathbf{Pub}) \Rightarrow (\varphi_j = \emptyset) \mid j \in J\} \cup C_1$$

where I and J are finite sets, $\ell_i, \ell'_i, \ell''_j$ are qualifier variables or constants, and C_1 is a set of effect constraints (like $\varphi_1 \leq \varphi_2$). Here, constraints on qualifiers come from equality constraints on types and conditions $\mathbf{Pub}(\tau)$ and $\mathbf{Taint}(\tau)$. In particular, $(\ell''_j = \mathbf{Pub}) \Rightarrow (\varphi_j = \emptyset)$ comes from the rule for $\mathbf{Taint}(\mathbf{N}_{\ell''_j}(\varphi, \varphi_j))$. By obtaining the most general unifier θ of the first set of constraints $\{\ell_i = \ell'_i \mid i \in I\}$ we obtain the constraint set $C' \equiv \{(\theta\ell''_j = \mathbf{Pub}) \Rightarrow (\theta\varphi_j = \emptyset) \mid j \in J\} \cup \theta C_1$. Let $\gamma_1, \ldots, \gamma_k$ be the remaining qualifier variables, and let $\theta' = [\mathbf{Pr}/\gamma_1, \ldots, \mathbf{Pr}/\gamma_k]$. Then C is satisfiable if and only if $\theta' C'$ is satisfiable. Thus, we obtain the set $\theta' C'$ of effect constraints that is satisfiable if and only if $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; \emptyset \vdash P$ holds.

Except for step 3, the above algorithm is almost the same as our previous work and we refer the interested reader to [17, 18]. By a similar argument to that given in [18] we can show that under the assumptions that the size of each begin/end assertion occurring in the protocol is bounded by a constant and that the size of simple types is polynomial in the size of the protocol, the type inference algorithm runs in polynomial time.

Example 3. Recall the POSH protocol in Figure 2. By the simple type inference in step 1 we get the following types for names:

$$\mathit{non}, \mathit{non}' : \mathbf{N}, \mathit{pk}_B : \mathbf{DKey}(\mathbf{N} \times \mathbf{N}), \ldots$$

By preparing qualifier and capability variables we get the following elaborated types and constraints on those variables:

$$\begin{array}{l}\mathit{non} : \mathbf{N}_{\gamma_1}(\xi_{0,o}, \xi_{0,c}), \mathit{non}' : \mathbf{N}_{\gamma_1'}(\xi'_{0,o}, \xi'_{0,c}), \ldots \\ \mathbf{Pub}(\mathbf{N}_{\gamma_1}(\xi_{0,o}, \xi_{0,c})) \quad \gamma_1 = \gamma_1' \quad \xi_6 \leq \xi_3 + \xi_4 + \xi_5 \\ \xi_2 \geq \xi'_{0,o} + (\xi_5 - \xi'_{0,c}) \quad \xi_7 \geq \xi_1 + \xi_2 + \xi_3 + \{\mathbf{chk}_{\gamma_1}(\mathit{non}, \xi_4)\} \quad \cdots\end{array}$$

Here, the constraint $\mathbf{Pub}(\mathbf{N}_{\gamma_1}(\xi_{0,o}, \xi_{0,c}))$ comes from *net*!*non*, and the other constraints from $\mathbf{check}$ *non* $\mathbf{is}$ *non*. $\cdots$. By solving the qualifier constraints, we get $\gamma_1 = \gamma_1' = \mathbf{Pub}, \ldots$, and are left with constraints on capability variables. By computing (an over-approximation of) the domain of each capability, we can reduce it to constraints on linear inequalities. For example, by letting $\xi_i = \{\mathbf{chk}_{\mathbf{Pub}}(\mathit{non}, \xi_4) \mapsto x_i, \mathbf{end}(m) \mapsto y_i, \ldots\}$, the last constraint is reduced to:

$$x_7 \geq x_1 + x_2 + x_3 + 1 \quad y_7 \geq y_1 + y_2 + y_3 + 0 \quad \cdots$$

5 Implementation and Experiments

We have implemented a protocol verifier SpiCA2 based on the type system and inference algorithm discussed above. The implementation is mostly based on the formalization in the paper, except for a few extensions such as sum types and private channels to securely distribute initial keys. The implementation can be tested at `http://www.kb.ecei.tohoku.ac.jp/~koba/spica2/`.

We have tested SpiCA2 on several protocols with the results of the experiments shown in Table 5. Experiments were conducted using a machine with a 3GHz CPU and 2GB of memory.

The descriptions of the protocols used in the experiments are available at the above URL. `POSH`, `SOPH`, and `SOSH` are (spi$_{CA}$-notations of) the protocols given in Figure 1. `GNSL` is the generalized Needham-Schroeder-Lowe protocol [9]: see [10] for details. `Otway-Ree` is Otway-Ree protocol using symmetric keys. `Iso-two-pass` is from [15], and the remaining protocols are the Needham-Schroeder-Lowe protocol and its variants, taken from the sample programs of Cryptyc [16] (but with type annotations and casts removed). `ns-flawed` is the original flawed version, `nsl-3` and `nsl-7` are 3- and 7-message versions of Lowe's fix, respectively. See [16] for the other three. As the table shows, all the protocols have been correctly verified or rejected. Furthermore, verification succeeded in less than a second except for `GNSL`. For `GNSL`, the slow-down is caused by the explosion of the number of atomic capabilities to be considered, which blows up the number of linear inequalities obtained from capability constraints.

Protocols	Typing	Time (sec.)	Protocols	Typing	Time (sec.)
POSH	yes	0.001	ns-flawed	no	0.007
SOPH	yes	0.001	nsl-3	yes	0.015
SOSH	yes	0.001	nsl-7	yes	0.049
GNSL	yes	7.40	nsl-optimized	yes	0.012
Otway-Ree	yes	0.019	nsl-with-secret	yes	0.023
Iso-two-pass	yes	0.004	nsl-with-secret-optimized	yes	0.016

Table 1. Experimental results

6 Extensions

In this section, we hint on how to modify our type system and type inference algorithm to deal with other features. Formalization and implementation of the extensions are left for future work.

Our type system can be easily adopted to deal with non-injective correspondence [13], which allows multiple end-events to be matched by a single begin-event. It suffices to relax the typing rules, for example, by changing the rules for begin- and end-events to:

$$\frac{\Gamma;\varphi + \{\mathbf{end}(M) \mapsto r\} \vdash P \qquad r > 0}{\Gamma;\varphi \vdash \mathbf{begin}\, M.P} \qquad \frac{r > 0}{\Gamma;\varphi + \{\mathbf{end}(M) \mapsto r\} \vdash \mathbf{end}\, M}$$

Fournet et al. [12] generalized begin- and end-events by allowing predicates to be defined by Datalog programs. For example, the process:

assume *employee*(a); **expect** *canRead(a, handbook)*

is safe in the presence of the clause "canRead(X,handbook) :- employee(X)". Here, the primitives **assume** and **expect** are like non-injective versions of **begin** and **end**. A similar type system can be obtained by extending our capabilities to mappings from ground atomic formulas to rational numbers (where $\varphi(L) > 0$ means L holds), and introducing rules for assume and expect similar to the rules above for begin and end-events. To handle clauses like "canRead(X,handbook) :- employee(X)", we can add the following rule:

$$\frac{\begin{array}{c}\Gamma;\varphi + \{L \mapsto r\} \vdash P \qquad \text{There is an (instance of) clause } L :- L_1, \ldots, L_k \\ r \leq \varphi(L_i) \text{ for each } i \in \{1, \ldots, k\}\end{array}}{\Gamma;\varphi \vdash P}$$

This allows us to derive a capability for L whenever there are capabilities for $L_1, \ldots, L_k$. To reduce capability constraints to linear programming problems, it suffices to extend the algorithm to obtain the domain of each effect [18], taking clauses into account (more precisely, if there is a clause $L :- L_1, \dots, L_k$ and $\theta L_1, \dots, \theta L_k$ are in the domain of φ, we add θL to the domain of φ).

To deal with trust and witness in [15], we need to mix type environments and capabilities, so that type environments can also be attached to names and passed around. The resulting type system is rather complex, so that we leave the details to another paper.

7 Related Work

The present work extends two lines of previous work: Gordon and Jeffrey's type systems for authenticity [14, 15], and Kikuchi and Kobayashi's work to enable type inference for symmetric cryptographic protocols [18]. In our opinion the extension is non-trivial, requiring the generalization of name types and a redesign of the type system. This has yielded a fully-automated and efficient protocol verifier. As for the expressive power, the fragment of Gordon and Jeffrey's type system (subject to minor restrictions) without trust and witness can be easily embedded into our type system. On the other hand, thanks to the uniform treatment of name types in terms of capabilities and obligations, our type system can express protocols that are not typable in Gordon and Jeffrey's type system, like the GNSL multi-party protocol [9]. See [10] for more details.

Gordon et al. [3, 4] extended their work to verify source code-level implementation of cryptographic protocols by using refinement types. Their type systems still require refinement type annotations. We plan to extend the ideas of the present work to enable partial type inference for their type system. Bugliesi, Focardi, and Maffei [6, 11, 7] have proposed a protocol verification method that is closely related to Gordon and Jeffrey's type systems. They [11] developed an algorithm for automatically inferring *tags* (which roughly correspond to Gordon and Jeffrey's types in [14, 15]). Their inference algorithm is based on exhaustive search of taggings by backtracking, hence our type inference would be more efficient. As in Gordon and Jeffrey type system, their tagging and typing system is specialized for the typical usage of nonces in two-party protocols, and appears to be inapplicable to multi-party protocols like GNSL.

There are automated protocol verification tools based on other approaches, such as ProVerif [5] and Scyther [8]. Advantages of our type-based approach are: (i) it allows modular verification of protocols[6]; (ii) it sets up a basis for studies of partial or full type inference for more advanced type systems for protocol verification [4] (for an evidence, recall Section 6); and (iii) upon successful verification, it generates types as a certificate, which explains why the protocol is safe, and can be independently checked by other type-based verifiers [15, 4]. On the other hand, ProVerif [5] and Scyther [8] have an advantage that they can generate an attack scenario given a flawed protocol. Thus, we think that our type-based tool is complementary to existing tools.

[6] Although the current implementation of SPICA2 only supports whole protocol analysis, it is easy to extend it to support partial type annotations to enable modular verification. For that purpose, it suffices to allow bound variables to be annotated with types, and generate the corresponding constraints during type inference. For example, for a type-annotated input $M?(x : \tau_1).P$, we just need to add the subtype constraint $\tau_1 \leq \tau$ to rule T-IN.

Acknowledgment This work was partially supported by the Mitsubishi Foundation.

References

1. Abadi, M.: Secrecy by typing in security protocols. JACM 46(5), 749–786 (1999)
2. Abadi, M., Gordon, A.D.: A Calculus for Cryptographic Protocols: The Spi Calculus. Information and Computation 148(1), 1–70 (January 1999)
3. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008). pp. 17–32 (2008)
4. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular verification of security protocol code by typing. In: Proceedings of POPL 2010. pp. 445–456 (2010)
5. Blanchet, B.: From Secrecy to Authenticity in Security Protocols. In: 9th International Static Analysis Symposium (SAS'02). LNCS, vol. 2477, pp. 342–359. Springer-Verlag (2002)
6. Bugliesi, M., Focardi, R., Maffei, M.: Analysis of typed analyses of authentication protocols. In: 18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005). pp. 112–125 (2005)
7. Bugliesi, M., Focardi, R., Maffei, M.: Dynamic types for authentication. Journal of Computer Security 15(6), 563–617 (2007)
8. Cremers, C.J.F.: Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In: Proceedings of ACM Conference on Computer and Communications Security (CCS 2008). pp. 119–128 (2008)
9. Cremers, C.J.F., Mauw, S.: A family of multi-party authentication protocols - extended abstract. In: Proceedings of WISSEC'06 (2006)
10. Dahl, M., Kobayashi, N., Sun, Y., Hüttel, H.: Type-based automated verification of authenticity in asymmetric cryptographic protocols. Full version, available at `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/protocol-full.pdf` (2011)
11. Focardi, R., Maffei, M., Placella, F.: Inferring authentication tags. In: Proceedings of the Workshop on Issues in the Theory of Security (WITS 2005). pp. 41–49 (2005)
12. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization policies. ACM Trans. Prog. Lang. Syst. 29(5) (2007)
13. Gordon, A.D., Jeffrey, A.: Typing one-to-one and one-to-many correspondences in security protocols. In: Software Security – Theories and Systems, Mext-NSF-JSPS International Symposium (ISSS 2002). LNCS, vol. 2609, pp. 263–282. Springer-Verlag (2002)
14. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. Journal of Computer Security 11(4), 451–520 (2003)
15. Gordon, A.D., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. Journal of Computer Security 12(3-4), 435–483 (2004)
16. Haack, C., Jeffrey, A.: Cryptyc. `http://www.cryptyc.org/` (2004)
17. Kikuchi, D., Kobayashi, N.: Type-based verification of correspondence assertions for communication protocols. In: Proceedings of APLAS 2007. LNCS, vol. 4807, pp. 191–205. Springer-Verlag (2007)
18. Kikuchi, D., Kobayashi, N.: Type-based automated verification of authenticity in cryptographic protocols. In: Proceedings of ESOP 2009. LNCS, vol. 5502, pp. 222–236. Springer-Verlag (2009)
19. Pierce, B., Sangiorgi, D.: Typing and subtyping for mobile processes. Mathematical Structures in Computer Science 6(5), 409–454 (1996)
20. Woo, T.Y., Lam, S.S.: A semantic model for authentication protocols. In: RSP: IEEE Computer Society Symposium on Research in Security and Privacy. pp. 178–193 (1993)