Type-Based Automated Verification of Authenticity in Cryptographic Protocols

Daisuke Kikuchi and Naoki Kobayashi

Graduate School of Information Sciences, Tohoku University
{kikuchi,koba}@kb.ecei.tohoku.ac.jp

Abstract. Gordon and Jeffrey have proposed a type and effect system for checking authenticity in cryptographic protocols. The type system reduces the protocol verification problem to the type checking problem, but protocols must be manually annotated with non-trivial types and effects. To automate the verification of cryptographic protocols, we modify Gordon and Jeffrey's type system and develop a type inference algorithm. Key modifications for enabling automated type inference are introduction of fractional effects and replacement of typing rules with syntax-directed ones. We have implemented and tested a prototype protocol verifier based on our type system.

1 Introduction

Gordon and Jeffrey [1–3] developed a series of type systems for verifying authenticity in security protocols. The required authenticity properties are described by using Woo and Lam's correspondence assertions [4], and Gordon and Jeffrey's type systems guarantee that well-typed processes (describing security protocols) satisfy the correspondence assertions. The type systems reduce the problem of verifying authenticity properties in security protocols to the type checking problem. Based on the type systems, Haack and Jeffrey implemented a verifier for cryptographic protocols [5].

One of the main shortcomings of their type systems was that protocols must be explicitly annotated with types. Since the types may contain complex information about how communication channels, cryptographic keys, and nonces should be used in protocols, it seems difficult for non-expert users (especially those who are not familiar with the type systems) to supply such annotations.

In our previous work [6], we have extended Gordon and Jeffrey's type system for checking correspondence assertions [1] in the π-calculus (without cryptographic primitives), the first and simplest one in the series of their type systems, and developed a polynomial-time type inference algorithm for it. The key idea of the extension was to introduce *fractional* effects, which allowed us to reduce the type inference problem to linear programming over rational numbers, rather than integer linear programming.

In this paper, we extend our previous work [6], and show that a similar technique can be used to develop a type inference algorithm for a variant of

Gordon and Jeffrey's type system for checking authenticity in cryptographic protocols [2]. The key technique for enabling efficient type inference is to allow fractional effects, as in our previous work [6]. Some new challenges, however, arise in dealing with cryptographic primitives [2]. First, there are two rules for each message/process constructor in their type system: one for trusted data, and the other for untrusted data.[1] Second, there is an explicit cast operation for capturing the role of nonces in cryptographic protocols. These features make even the simple type inference (without effects) non-trivial. We modify Gordon and Jeffrey's type system [2] to remove those problems, so that there is only one rule for each message/process constructor, and no explicit cast operation is required. That modification allows us to develop a type inference algorithm in a manner similar to our previous work [6]. Although the expressive power of our type system is incomparable to that of Gordon and Jeffrey's type system [2] (there are processes typable in their type system but not in our type system, and vice versa), all the examples discussed in [2] are typable in our type system (modulo an extension with labeled variants).

The rest of this paper is structured as follows. Section 2 introduces `SpiCA`, an extension of the Spi-calculus with correspondence assertions, which is used for describing cryptographic protocols. Section 3 introduces our new type system for checking authenticity of cryptographic protocols. Section 4 describes a type inference algorithm, which serves as an algorithm for automatic verification of authenticity in cryptographic protocols. Section 5 reports preliminary experiments. Section 6 discusses related work and Section 7 concludes. A longer version of this paper is available from `http://www.kb.ecei.tohoku.ac.jp/~koba/esop09-long.pdf`.

2 `SpiCA`: Spi-Calculus with Correspondence Assertions

In this section, we introduce the language `SpiCA`, an extension of the Spi-calculus [8] with correspondence assertions. The language is similar to Gordon and Jeffrey's calculus [2]: our language is obtained from it by removing type annotations and cast operations.

2.1 Syntax

Definition 1 (messages, processes) The sets of *messages* and *processes*, ranged over by M and P respectively, are given by:

$$\begin{aligned} K, M, N &::= x \mid (M_1, M_2) \mid \mathbf{inl}(M) \mid \mathbf{inr}(M) \mid \{M\}_K \\ P, Q &::= \mathbf{0} \mid M!N \mid M?x.P \mid (P_1 \,|\, P_2) \mid *P \mid (\nu x)P \mid \mathbf{check}\ x\ \mathit{is}\ M.P \\ &\quad\ \mid \mathbf{decrypt}\ M\ \mathit{is}\ \{x\}_K.P \mid \mathbf{case}\ M\ \mathit{is}\ \mathbf{inl}(x).P\ \mathit{is}\ \mathbf{inr}(y).Q \\ &\quad\ \mid \mathbf{split}\ M\ \mathit{is}\ (x, y).P \mid \mathbf{begin}\ M.P \mid \mathbf{end}\ M.P \end{aligned}$$

Here, the meta-variables x and y range over the set $\mathcal{N}$ of variables.

[1] In a subsequent paper [7], Gordon and Jeffrey uses subtyping and merges the two rules into one. The subsumption rule still makes type inference difficult.

The variable x is bound in $M?x.P$, $(\nu x)P$ and **decrypt** M *is* $\{x\}_K.P$. x and y are bound in **case** M *is* $\mathbf{inl}(x).P$ *is* $\mathbf{inr}(y).Q$ and **split** M *is* $(x, y).P$. We write $[M/x]P$ for the process obtained by replacing x in P with M.

(M_1, M_2) is a pair consisting of M_1 and M_2. $\mathbf{inl}(M)$ and $\mathbf{inr}(M)$ are constructors for sums. $\{M\}_K$ is the message obtained by encrypting M with key K. Here, we assume perfect encryption; information about an encrypted message can be obtained only if the key is known.

The process $\mathbf{0}$ is an inaction. The process $M!N$ sends the message N on the channel M. The process $M?x.P$ waits to receive a message on channel M, binds x to it, and then behaves like P. $(P_1 \,|\, P_2)$ runs P_1 and P_2 in parallel, while $*P$ runs infinitely many copies of P in parallel. The process $(\nu x)P$ creates a fresh name (which may be used as a channel, a nonce, or a symmetric key), binds x to it, and behaves like P. The process **check** x *is* $M.P$ behaves like P if the values of x and M are the same; otherwise the process is aborted. The process **decrypt** M *is* $\{x\}_K.P$ decrypts the message M with the symmetric-key K. If the decryption succeeds, the process binds x to the decrypted message, and behaves like P; otherwise, the process is aborted. The process **case** M *is* $\mathbf{inl}(x).P$ *is* $\mathbf{inr}(y).Q$ behaves like $[N/x]P$ if M is of the form $\mathbf{inl}(N)$, and behaves like $[N/y]Q$ if M is of the form $\mathbf{inr}(N)$. The process **split** M *is* $(x, y).P$ splits the pair $M = (M_1, M_2)$, binds x to M_1 and y to M_2, and behaves like P.

The processes **begin** $M.P$ and **end** $M.P$ are special processes for declaring correspondence assertions; **begin** $M.P$ raises a "**begin** M" event and then behaves like P, while **end** $M.P$ raises an "**end** M" event and then behaves like P. It is expected (and will be guaranteed by our type system) that whenever an end-event occurs, a corresponding begin-event must have occurred before. Authenticity properties (like "if Alice receives a message m, then Bob must have sent the message") are reduced to such relations between begin- and end-events: See Example 1 below.

Example 1. Consider the following process *System*, taken from [2]:

$$\begin{aligned}
\mathit{System} &\triangleq (\nu \mathit{key})(*\mathit{Sender}(\mathit{ch}, \mathit{key}) \mid *\mathit{Receiver}(\mathit{ch}, \mathit{key}))\\
\mathit{Sender}(\mathit{ch}, \mathit{key}) &\triangleq \mathit{ch}?n.(\nu \mathit{msg})\mathbf{begin}\ \mathit{msg}.\mathit{ch}!\{(\mathit{msg}, n)\}_{\mathit{key}}\\
\mathit{Receiver}(\mathit{ch}, \mathit{key}) &\triangleq (\nu \mathit{non})(\mathit{ch}!\mathit{non} \mid \mathit{ch}?\mathit{ctext}.\mathbf{decrypt}\ \mathit{ctext}\ \mathit{is}\ \{x\}_{\mathit{key}}.\\
&\quad\ \mathbf{split}\ x\ \mathit{is}\ (m, \mathit{non}').\mathbf{check}\ \mathit{non}\ \mathit{is}\ \mathit{non}'.\mathbf{end}\ m)
\end{aligned}$$

System creates a shared key *key*, and runs infinitely many copies of $\mathit{Sender}(\mathit{ch}, \mathit{key})$ and $\mathit{Receiver}(\mathit{ch}, \mathit{key})$ in parallel. Here, *ch* is a public communication channel, on which an attacker may also send/receive messages. The process $\mathit{Receiver}(\mathit{ch}, \mathit{key})$ creates a fresh name *non* (which is often called a *nonce* in the terminology of security protocols) and sends it on *ch*. The process $\mathit{Sender}(\mathit{ch}, \mathit{key})$ then receives the nonce and creates a new message. It then raises a "**begin** *msg*"-event and sends the cyphertext $\{(\mathit{msg}, \mathit{non})\}_{\mathit{key}}$ on *ch*. Here, the event "**begin** *msg*" represents the fact that the process certainly sent the message *msg*. $\mathit{Receiver}(\mathit{ch}, \mathit{key})$ then receives the cyphertext, decrypts it (as a result, m and non' are bound to *msg* and *non*), and checks that the second element of the decrypted message matches the nonce it has sent before. The receiver then raises the event **end** *msg*, meaning that it has received the message *msg*. The correspondence between **end** *msg* and **begin** *msg*, (i.e., the property that whenever an "**end** *msg*"-event happens, a "**begin** *msg*"-event must have occurred before) assures that each time $\mathit{Receiver}(ch, key)$ executes **end** *msg*, the message *msg* has certainly been sent by $\mathit{Sender}(ch, key)$.

Note that the nonce check "**check** non *is* $non'.\cdots$" in the protocol above is essential; if there is no such check, then an attacker can confuse the receiver by duplicating a message $\{(msg, non)\}_{key}$ (by running $ch?x.(ch!x \mid ch!x)$, for instance). □

2.2 Semantics

The operational semantics is given in Figure 1. (The rules for **split**, **case** and replications are omitted.) Here, a state is represented as a triple $\langle \Psi, E, N\rangle$, where Ψ is a multiset of processes, N is a set of names, and E is a multiset of messages M such that the event **begin** M has been raised but **end** L has not. In other words, E describes capabilities (or, permissions) to raise end-events.

$$\begin{array}{c}
\langle \Psi \uplus \{x?y.P, x!M\}, E, N\rangle \longrightarrow \langle \Psi \uplus \{[M/y]P\}, E, N\rangle \\
\langle \Psi \uplus \{P \mid Q\}, E, N\rangle \longrightarrow \langle \Psi \uplus \{P, Q\}, E, N\rangle \\
\langle \Psi \uplus \{(\nu x)P\}, E, N\rangle \longrightarrow \langle \Psi \uplus \{[y/x]P\}, E, N \cup \{y\}\rangle \ (y \notin N) \\
\langle \Psi \uplus \{\mathbf{check}\ x\ \mathit{is}\ x.P\}, E, N\rangle \longrightarrow \langle \Psi \uplus \{P\}, E, N\rangle \\
\langle \Psi \uplus \{\mathbf{decrypt}\ \{M\}_K\ \mathit{is}\ \{x\}_K.P\}, E, N\rangle \longrightarrow \langle \Psi \uplus \{[M/x]P\}, E, N\rangle \\
\langle \Psi \uplus \{\mathbf{begin}\ L.P\}, E, N\rangle \longrightarrow \langle \Psi \uplus \{P\}, E \uplus \{L\}, N\rangle \\
\langle \Psi \uplus \{\mathbf{end}\ L.P\}, E \uplus \{L\}, N\rangle \longrightarrow \langle \Psi \uplus \{P\}, E, N\rangle
\end{array}$$

Fig. 1. Operational Semantics

We write $\langle \Psi, E, N\rangle \longrightarrow$ **Error** if **end** $L.P \in \Psi$ but $L \notin E$. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. The required correspondence between begin-events and end-events is stated as follows.

Definition 2 (safety) A process P is *safe* if $\langle \{P\}, \emptyset, N\rangle \not\longrightarrow^*$ **Error**, where N is the set of free names in P.

For security protocols, the safety of the process running protocols alone is not sufficient; the robust safety defined below means that the process is safe in the presence of attackers running in parallel.

Definition 3 (robust safety) A process P is *robustly safe* if $(P \mid O)$ is safe for any process O that contains no begin/end/check-assertions.

Remark 1. In Gordon and Jeffrey's definition [2], attackers may execute check operations. We removed them, as the check operations do not increase the power of attackers. **check** M *is* $N.P$ can be simulated by **decrypt** $\{x\}_M$ *is* $\{y\}_N.P$.

3 Type System

3.1 Types and Effects

Definition 4 (effects) The sets of *types* and *effects*, ranged over by T and e, are given by:

$$\begin{array}{rl} T \text{ (types)} & ::= \mathbf{N}(e) \mid \mathbf{Key}(T) \mid T_1 \times T_2 \mid T_1 + T_2 \\ e \text{ (effects)} & ::= [A_1 \mapsto r_1, \ldots, A_n \mapsto r_n] \\ A \text{ (atomic effects)} & ::= \mathbf{end}\langle M\rangle \mid \mathbf{chk}\langle \alpha\rangle \\ \alpha \text{ (extended names)} & ::= x \mid i \\ i \text{ (indices)} & ::= 0 \mid 1 \mid 2 \mid \cdots \end{array}$$

Here, $r_1, \ldots, r_n$ ranges over the set of non-negative rational numbers.

The type $\mathbf{N}(e)$ describes names used as channels, nonces, or cyphertexts. When the type describes a nonce, the effect e describes a capability to raise end-events carried by the nonce. For example, *non* passed through *ch* in Example 1 carries a capability to raise one "**end** *msg*"-event, so that its type is $\mathbf{N}([\mathbf{end}\langle \mathit{msg}\rangle \mapsto 1])$. We often write **Un** for $\mathbf{N}([\,])$. The type $\mathbf{Key}(T)$ describes keys used for decrypting messages of type T.

The type $T_1 \times T_2$ describes pairs consisting of messages of types T_1 and T_2. Indices are used to express dependencies of the second element on the first element: For example, $\mathbf{Un} \times \mathbf{N}([\mathbf{end}\langle 0\rangle \mapsto 1])$ describes a pair (a, b), where b's type is $\mathbf{N}([\mathbf{end}\langle a\rangle \mapsto 1])$. The type corresponds to $(x{:}\mathbf{Un}, \mathbf{N}([\mathbf{end}\langle x\rangle \mapsto 1])$ in Gordon and Jeffrey's notation [2]. The type $\mathbf{Un} \times (\mathbf{Un} \times \mathbf{N}([\mathbf{end}\langle 0\rangle \mapsto 1, \mathbf{chk}\langle 1\rangle \mapsto 1]))$ describes a message of the form $(a, (b, c))$ where a and b have type **Un**, and c has type $\mathbf{N}([\mathbf{end}\langle b\rangle \mapsto 1, \mathbf{chk}\langle a\rangle \mapsto 1])$. We use the nameless representation of dependent types just for technical convenience for formalizing type inference; in terms of the expressiveness of the type system, the nameless dependent types are equivalent to Gordon and Jeffrey's name dependent types [2].

The type $T_1 + T_2$ describes sums of the form $\mathbf{inl}(M)$ (where M is a message of type T_1) or $\mathbf{inr}(M)$ (where M is a message of type T_2).

An effect $[A_1 \mapsto r_1, \ldots, A_n \mapsto r_n]$ denotes the mapping f from the set of atomic events to the set of rational numbers such that $f(A_i) = r_i$ for $i \in \{1, \ldots, n\}$ and $f(M) = 0$ for $M \notin \{A_1, \ldots, A_n\}$. The atomic effect $\mathbf{end}\langle M\rangle$ denotes a capability to execute "**end** M," while the atomic effect $\mathbf{chk}\langle \alpha\rangle$ denotes a capability to execute **check** α *is* $x.P$. The latter kind of effect is used to guarantee that each nonce can be checked at most once. In the rest of this paper, the words "effects" and "capabilities" are used interchangeably.

Example 2. Names in Example 1 have the following types.

$$\mathit{ch} : \mathbf{Un} \quad x : \mathbf{N}([\,]) \times \mathbf{N}([\mathbf{end}\langle 0\rangle \mapsto 1]) \quad \mathit{key} : \mathbf{Key}(\mathbf{N}([\,]) \times \mathbf{N}([\mathbf{end}\langle 0\rangle \mapsto 1]))$$

A substitution $[x_1/i_1, \ldots, x_k/i_k]$, denoted by meta-variable θ, is a mapping from indices to names. The substitution, summation, and binary relation $\leq$ on effects are defined by:

$$(\theta e)(A) = \Sigma\{e(A') \mid \theta A' = A\} \qquad (e_1 + e_2)(A) = e_1(A) + e_2(A)$$
$$e \leq e' \Leftrightarrow \forall A.e(A) \leq e'(A)$$

The substitution θT on types is defined by:

$$[x/i]\mathbf{N}(e) = \mathbf{N}([x/i]e) \qquad [x/i]\mathbf{Key}(T) = \mathbf{Key}([x/i]T)$$
$$[x/i](T_1 \times T_2) = [x/i]T_1 \times [x/(i+1)]T_2 \qquad [x/i](T_1 + T_2) = [x/i]T_1 + [x/i]T_2$$

3.2 Typing Rules

We introduce two type judgment forms: $\Gamma; e \vdash M : T$ for messages and $\Gamma; e \vdash P$ for processes. Here, Γ, called a type environment, is a finite sequence of bindings of names to types. $\Gamma; e \vdash M : T$ means that given names described by Γ and capabilities described by e, one can construct a message M of type T. For example, we have

$$x : \mathbf{Un}, y : \mathbf{Un}; [\mathbf{end}\langle x\rangle \mapsto 1] \vdash (x, y) : \mathbf{Un} \times \mathbf{N}([\mathbf{end}\langle 0\rangle \mapsto 1]).$$

In this manner, capabilities (to raise end-events or check nonces) can be attached to a name, and passed to other processes.

$\Gamma; e \vdash P$ means that given names described by Γ and capabilities described by e, the process P can be safely executed. For example, $x{:}\mathbf{Un}, y{:}\mathbf{Un}; [\mathbf{end}\langle x\rangle \mapsto 1] \vdash \mathbf{end}\ x$ is a valid judgment, but $x{:}\mathbf{Un}, y{:}\mathbf{Un}; [\mathbf{end}\langle x\rangle \mapsto 1] \vdash \mathbf{end}\ y$ is invalid since there is no capability to execute $\mathbf{end}\ y$. When we write $\Gamma; e \vdash M : T$ or $\Gamma; e \vdash P$, we implicitly assume that Γ, e, and T are well-formed, in the sense that they do not contain undefined names. For example, when we write "$\Gamma, x{:}T$," only the names bound in Γ may occur in T.

The typing rules for messages are given in Figure 2. In rule MT-Var, $T + e$ is defined as $\mathbf{N}(e' + e)$ if T is of the form $\mathbf{N}(e')$; otherwise $T + e$ is T. The capabilities e are transferred from the environment to x if x has type $\mathbf{N}(e')$. The role of the rule is similar to that of Gordon and Jeffrey's typing rule for cast-operations [2]. Unlike in Gordon and Jeffrey's type system, however, the transfer of capabilities from the environment to a name is implicitly performed by MT-Var. The capabilities attached to a name can be extracted at most once by a check operation: see the rule T-Check given later.

In rule MT-Pair, the index 0 in T_2 refers to the first element, so that N must have type $[M/0]T_2$. The other rules are standard.

The typing rules for processes are given in Figure 3. Note that we have only one rule for each process constructor (except T-SubEf, which can be easily eliminated), while Gordon and Jeffrey's type system [2] had two rules for each process constructor: one for trusted data and the other for untrusted data.

In the figure, $FN(e)$ denotes the set $\bigcup\{FN(A) \mid e(A) > 0\}$, where $FN(A)$ is the set of extended names occurring in A. For example, $FN([\mathbf{end}\langle(x, y)\rangle \mapsto 0.5, \mathbf{end}\langle(y, z)\rangle \mapsto 0]) = \{x, y\}$.

The predicate $\mathbf{pub}(T)$ used in the figure is defined inductively by:

$$\frac{}{\mathbf{pub}(\mathbf{N}([\,]))} \qquad \frac{\mathbf{pub}(T)}{\mathbf{pub}(\mathbf{Key}(T))}$$

$$\frac{\mathbf{pub}(T_1) \qquad \mathbf{pub}(T_2)}{\mathbf{pub}(T_1 \times T_2)} \qquad \frac{\mathbf{pub}(T_1) \qquad \mathbf{pub}(T_2)}{\mathbf{pub}(T_1 + T_2)}$$

In other words, $\mathbf{pub}(T)$ holds if T does not carry any effects. The predicate $\mathbf{gen}(T)$ means that T is of the form $\mathbf{N}([\,])$ or $\mathbf{Key}(T')$.

We explain some of the key typing rules below. A communication channel in our calculus is an untrusted communication device, on which attackers may intercept, duplicate messages, etc. In rules T-Out and T-In, therefore, the type of messages sent on a channel must be public, meaning that they must not contain effects. To send a name carrying effects, one must encrypt it; otherwise, an attacker may abuse the effects (or, capabilities) carried by the name. Besides the requirement that it must be public, there is no restriction on the type of messages; thus, well-typed processes may suffer from type mismatch errors at run-time (when executing split and case expressions).

The rule T-Par splits the capabilities $e_1 + e_2$ into e_1 and e_2 for P_1 and P_2 respectively. In rule T-Res, x is a fresh name, so that a capability to use x as a nonce and check x is added to P.

The rule T-Check says that the check-expression extracts the capability e' carried by N, by consuming the capability to check x; the consumption of the capability $\mathbf{chk}\langle x\rangle$ ensures that the capability e' can no longer be extracted. The rules T-Begin and T-End say that the begin-expression adds the capability to raise an end-event, while the end-expression consumes the capability to raise an end-event. The rule T-SubEf allows some capabilities not to be used (so that for a begin-event, there may be no corresponding end-event).

$$\frac{}{\Gamma, x : T\,;\, e \vdash x : T + e} \qquad \text{(MT-Var)}$$

$$\frac{\Gamma\,;\, e_1 \vdash M : T_1 \qquad \Gamma\,;\, e_2 \vdash N : [M/0]T_2}{\Gamma\,;\, e_1 + e_2 \vdash (M, N) : T_1 \times T_2} \qquad \text{(MT-Pair)}$$

$$\frac{\Gamma\,;\, e \vdash M : T_1}{\Gamma\,;\, e \vdash \mathbf{inl}(M) : T_1 + T_2} \qquad \text{(MT-Inl)}$$

$$\frac{\Gamma\,;\, e \vdash M : T_2}{\Gamma\,;\, e \vdash \mathbf{inr}(M) : T_1 + T_2} \qquad \text{(MT-Inr)}$$

$$\frac{\Gamma\,;\, e \vdash M : T \qquad \Gamma\,;\, [\,] \vdash K : \mathbf{Key}(T)}{\Gamma\,;\, e \vdash \{M\}_K : \mathbf{Un}} \qquad \text{(MT-Encrypt)}$$

Fig. 2. Typing for Messages

$$\frac{}{\Gamma\,;\,[\,]\vdash \mathbf{0}}\quad\text{(T-Zero)}$$

$$\frac{\Gamma\,;\,[\,]\vdash x:\mathbf{Un}\qquad \Gamma\,;\,e\vdash N:T\qquad \mathbf{pub}(T)}{\Gamma\,;\,e\vdash x!N}\quad\text{(T-Out)}$$

$$\frac{\Gamma\,;\,[\,]\vdash x:\mathbf{Un}\qquad \Gamma,y:T\,;\,e\vdash P\qquad \mathbf{pub}(T)\qquad y\notin FN(e)}{\Gamma\,;\,e\vdash x?y.P}\quad\text{(T-In)}$$

$$\frac{\Gamma\,;\,e_1\vdash P_1\qquad \Gamma\,;\,e_2\vdash P_2}{\Gamma\,;\,e_1+e_2\vdash P_1\mid P_2}\quad\text{(T-Par)}$$

$$\frac{\Gamma\,;\,[\,]\vdash P}{\Gamma\,;\,[\,]\vdash *P}\quad\text{(T-Rep)}$$

$$\frac{\Gamma,x:T\,;\,e+[\mathbf{chk}\langle x\rangle\mapsto 1]\vdash P\qquad x\notin FN(e)\qquad \mathbf{gen}(T)}{\Gamma\,;\,e\vdash(\nu x)P}\quad\text{(T-Res)}$$

$$\frac{\Gamma\,;\,[\,]\vdash x:\mathbf{Un}\qquad \Gamma\,;\,[\,]\vdash N:\mathbf{N}(e')\qquad \Gamma\,;\,e+e'\vdash P}{\Gamma\,;\,e+[\mathbf{chk}\langle x\rangle\mapsto 1]\vdash \mathbf{check}\ x\ \mathit{is}\ N.P}\quad\text{(T-Check)}$$

$$\frac{\Gamma\,;\,[\,]\vdash M:\mathbf{Un}\qquad \Gamma\,;\,[\,]\vdash K:\mathbf{Key}(T)\qquad \Gamma,y:T\,;\,e\vdash P\qquad y\notin FN(e)}{\Gamma\,;\,e\vdash \mathbf{decrypt}\ M\ \mathit{is}\ \{y\}_K.P}$$
(T-Decrypt)

$$\frac{\Gamma\,;\,[\,]\vdash M:T_1+T_2\qquad \Gamma,y:T_1\,;\,e\vdash P_1\qquad \Gamma,z:T_2\,;\,e\vdash P_2\qquad y,z\notin FN(e)}{\Gamma\,;\,e\vdash \mathbf{case}\ M\ \mathit{is}\ \mathbf{inl}(y).P_1\ \mathit{is}\ \mathbf{inr}(z).P_2}$$
(T-Case)

$$\frac{\Gamma\,;\,[\,]\vdash M:T_1\times T_2\qquad \Gamma,y:T_1,z:[y/0]T_2\,;\,e\vdash P\qquad y,z\notin FN(e)}{\Gamma\,;\,e\vdash \mathbf{split}\ M\ \mathit{is}\ (y,z).P}$$
(T-Split)

$$\frac{\Gamma\,;\,e+[\mathbf{end}\langle M\rangle\mapsto 1]\vdash P\qquad FN(M)\subseteq\mathbf{dom}(\Gamma)}{\Gamma\,;\,e\vdash \mathbf{begin}\ M.P}\quad\text{(T-Begin)}$$

$$\frac{\Gamma\,;\,e\vdash P\qquad FN(M)\subseteq\mathbf{dom}(\Gamma)}{\Gamma\,;\,e+[\mathbf{end}\langle M\rangle\mapsto 1]\vdash \mathbf{end}\ M.P}\quad\text{(T-End)}$$

$$\frac{\Gamma\,;\,e'\vdash P\qquad e'\le e}{\Gamma\,;\,e\vdash P}\quad\text{(T-SubEf)}$$

Fig. 3. Typing for Processes

Example 3. Recall Example 1. $\mathit{Sender}(\mathit{ch}, \mathit{key})$ is typed as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1\,;\,[\,] \vdash \mathit{ch} : \mathbf{Un} \quad \dfrac{\Gamma_1\,;\,[\,] \vdash \mathit{key} : T_2 \quad \dfrac{\Gamma_1\,;\,[\,] \vdash \mathit{msg} : \mathbf{Un} \quad \Gamma_1\,;\,e \vdash n : \mathbf{N}(e)}{\Gamma_1\,;\,e \vdash (\mathit{msg}, n) : T_1}}{\Gamma_1\,;\,[\mathbf{end}\langle \mathit{msg}\rangle \mapsto 1] \vdash \{(\mathit{msg}, n)\}_{\mathit{key}} : \mathbf{Un}}}{\Gamma_1\,;\,[\mathbf{end}\langle \mathit{msg}\rangle \mapsto 1] \vdash \mathit{ch}!\{(\mathit{msg}, n)\}_{\mathit{key}}}}{\Gamma_1\,;\,[\mathbf{chk}\langle \mathit{msg}\rangle \mapsto 1, \mathbf{end}\langle \mathit{msg}\rangle \mapsto 1] \vdash \mathit{ch}!\{\mathit{msg}, n\}_{\mathit{key}}}}{\Gamma_1\,;\,[\mathbf{chk}\langle \mathit{msg}\rangle \mapsto 1] \vdash \mathbf{begin}\ \mathit{msg}. \cdots}}{\mathit{ch} : \mathbf{Un}, \mathit{key} : T, n : \mathbf{Un}\,;\,[\,] \vdash (\nu \mathit{msg}) \cdots}}{\mathit{ch} : \mathbf{Un}, \mathit{key} : T\,;\,[\,] \vdash \mathit{Sender}(\mathit{ch}, \mathit{key})}$$

Here, $T_1 = \mathbf{Un} \times \mathbf{N}([\mathbf{end}\langle 0\rangle \mapsto 1])$, $T_2 = \mathbf{Key}(T_1)$, $e = [\mathbf{end}\langle \mathit{msg}\rangle \mapsto 1]$ and $\Gamma_1 = \mathit{ch} : \mathbf{Un}, \mathit{key} : T, n : \mathbf{Un}, \mathit{msg} : \mathbf{Un}$.

The sub-process $\mathit{ch}?\mathit{ctext}. \cdots$ of $\mathit{Receiver}(\mathit{ch}, \mathit{key})$ is typed as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_3\,;\,[\,] \vdash \mathit{non} : \mathbf{Un} \quad \Gamma_3\,;\,[\,] \vdash \mathit{non}' : \mathbf{N}(e') \quad \Gamma_3\,;\,e' \vdash \mathbf{end}\ m}{\Gamma_3\,;\,[\mathbf{chk}\langle \mathit{non}\rangle \mapsto 1] \vdash \mathbf{check}\ \mathit{non}\ \mathit{is}\ \mathit{non}'. \cdots}}{\Gamma_2, x : \mathbf{Un} \times \mathbf{N}([\mathbf{end}\langle 0\rangle \mapsto 1])\,;\,[\mathbf{chk}\langle \mathit{non}\rangle \mapsto 1] \vdash \mathbf{split}\ x\ \mathit{is}\ (m, \mathit{non}'). \cdots}}{\Gamma_2\,;\,[\mathbf{chk}\langle \mathit{non}\rangle \mapsto 1] \vdash \mathbf{decrypt}\ \mathit{ctext}\ \mathit{is}\ \{x\}_{\mathit{key}}. \cdots}}{\mathit{ch} : \mathbf{Un}, \mathit{key} : T_2, \mathit{non} : \mathbf{Un}\,;\,[\mathbf{chk}\langle \mathit{non}\rangle \mapsto 1] \vdash \mathit{ch}?\mathit{ctext}. \cdots}$$

Here, $e' = [\mathbf{end}\langle m\rangle \mapsto 1]$, $\Gamma_2 = \mathit{ch} : \mathbf{Un}, \mathit{key} : T, \mathit{non} : \mathbf{Un}, \mathit{ctext} : \mathbf{Un}$ and $\Gamma_3 = \Gamma_2, m : \mathbf{Un}, \mathit{non}' : \mathbf{N}([\mathbf{end}\langle m\rangle \mapsto 1])$. From this, we can get $\mathit{ch} : \mathbf{Un}, \mathit{key} : T\,;\,[\,] \vdash \mathit{Receiver}(\mathit{ch}, \mathit{key})$.

The entire system *System* is typed as $\mathit{ch} : \mathbf{Un}\,;\,[\,] \vdash \mathit{System}$. □

3.3 Type Soundness

The soundness of our type system is stated as follows.

Theorem 1 (robust safety). *If* $x_1 : \mathbf{Un}, \ldots\ x_n : \mathbf{Un}\,;\,[\,] \vdash P$*, then* P *is robustly safe.*

The theorem says that if x_1:$\mathbf{Un}, \ldots\ x_n$:$\mathbf{Un}$:$[\,] \vdash P$ holds, then the correspondence assertions in P hold even in the presence of attackers.

The rest of this subsection sketches the proof of the above theorem. Gordon and Jeffrey [2] proved the robust safety by showing (i) any well-typed process is safe, and (ii) any attacker (an opponent process) is well-typed. Our proof is similar, but a few modifications are required, because of the following points:

– An attacker process is not necessarily typed in our type system.
– The safety of a well-typed process usually follows from the fact that typing is preserved by reductions. Our type system does not, however, satisfy the type preservation property (recall that the rules T-In and T-Out imposes no restriction on the type of messages, except the condition $\mathbf{pub}(T)$).

To remedy the problems above, we first extend the type system. We add the following rules for subtyping and subsumption to the type system presented so far.

$$\frac{\mathbf{pub}(T) \qquad \mathbf{pub}(T')}{T \leq T'} \qquad\qquad \frac{\Gamma\,;\, e \vdash M : T' \qquad T' \leq T}{\Gamma\,;\, e \vdash M : T}$$

Let us write $\Gamma; e \vdash_{\mathtt{EX}} M : T$ if $\Gamma; e \vdash M : T$ is derivable in the extended type system. Then, we can prove the following lemmas in a manner similar to [2]:

Lemma 1. *If $\Gamma; [\,] \vdash_{\mathtt{EX}} P$, then P is safe.*

Lemma 2. *If O contains no begin/end/check-expressions and $FN(O) \subseteq \{x_1, \ldots, x_n\}$, then $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; [\,] \vdash_{\mathtt{EX}} O$ holds.*

We can now prove Theorem 1.

Proof of Theorem 1 Suppose that $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; e \vdash P$ holds. Let O be a process such that $FN(O) \subseteq \{x_1, \ldots, x_n\}$ and O contains no begin/end/check-expressions. It suffices to show that $P \,|\, O$ is safe.

By the definition of the extended type system, we have $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; [\,] \vdash_{\mathtt{EX}} P$. By Lemma 2, we have $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; [\,] \vdash_{\mathtt{EX}} O$. By rule T-Par, we obtain $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; [\,] \vdash_{\mathtt{EX}} P \,|\, O$. By Lemma 1, $P \,|\, O$ is safe. $\square$

3.4 On the Expressive Power of the Type System

The expressive power of our type system is incomparable to that of Gordon and Jeffrey's type system [2]. On one hand, the following process, which uses the name x both as a pair and a sum, is typed under $x : \mathbf{Un}$ in Gordon and Jeffrey's type system, but not in our type system (without the extension).

$$\mathbf{split}\ x\ \mathit{is}\ (y, z).\mathbf{case}\ x\ \mathit{is}\ \mathbf{inl}(y).\mathbf{0}\ \mathit{is}\ \mathbf{inr}(z).\mathbf{0}$$

On the other hand, consider the following process *HalfCap*.

$$\begin{aligned}&(\nu y)(\nu z)(\mathbf{begin}\ x.(c!\{y\}_k \,|\, d!\{z\}_k)\\ &\qquad\qquad |\ c?u.d?v.\mathbf{decrypt}\ u\ \mathit{is}\ \{y'\}_k.\mathbf{decrypt}\ v\ \mathit{is}\ \{z'\}_k.\\ &\qquad\qquad\qquad \mathbf{check}\ y\ \mathit{is}\ y'.\mathbf{check}\ z\ \mathit{is}\ z'.\mathbf{end}\ x.)\end{aligned}$$

The first process raises a begin-event, and passes the capability to raise an end-event through the names y and z. The above process *HalfCap* is typed as follows in our type system:

$$x : \mathbf{Un}, c : \mathbf{Un}, d : \mathbf{Un}, k : \mathbf{Key}(\mathbf{N}([\mathbf{end}\langle x\rangle \mapsto 0.5]))\,;\, [\,] \vdash \mathit{HalfCap}.$$

P is not, however, typable in Gordon and Jeffrey's type system.

Despite the difference of the expressive power, however, we expect that both the type systems are equally effective for realistic protocols. First, with the extension discussed in Section 3.3, our type system is strictly more expressive than Gordon and Jeffrey's type system: If P is well-typed in their type system, then the process obtained from P by removing type annotations and casts is well-typed in our type system. Second, *HalfCap* given above is an artificial example, and we are not aware of realistic protocols that use fractional capabilities.

4 Type Inference Algorithm

A type inference algorithm can be obtained in the same manner as in our previous work [6]. The algorithm consists of the following steps.

- Step 1: Generate constraints on effects based on the typing rules.
- Step 2: Reduce the constraints on effects into linear inequalities on rational numbers.
- Step 3: Check whether the linear inequalities have a solution.

The algorithm is sound and complete: Given a process P, the algorithm always terminates, and it outputs a type-annotated process if and only if $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; [\,] \vdash P$.

For the first step, we first eliminate the rule T-SubEf by combining it with other rules. For example, the rule T-End can be replaced by:

$$\frac{\Gamma; e' \vdash P \qquad FN(M) \subseteq \mathbf{dom}(\Gamma) \qquad e' + [\mathbf{end}\langle M\rangle \mapsto 1] \leq e}{\Gamma; e \vdash \mathbf{end}\ M.P} \qquad \text{(T-End')}$$

The resulting typing rules are syntax-directed: there is exactly one rule for each message/process constructor. Based on the typing rules, we can easily generate constraints on type and effect variables, and then reduce them to constraints on effect variables of the following forms:

$$e \leq e' \qquad FN(e) \subseteq \{\alpha_1, \ldots, \alpha_n\} \qquad \alpha \notin FN(e)$$

Here, e is an expression constructed from effects, effect variables, $+$, and substitutions.

The second step is also straightforward. We first obtain a set of atomic effects $\{A_1, \ldots, A_m\}$ that may occur in effects. We then replace each effect variable ρ with $[A_1 \mapsto \eta_{\rho,1}, \ldots, A_m \mapsto \eta_{\rho,m}]$ by preparing variables $\eta_{\rho,1}, \ldots, \eta_{\rho,m}$ ranging over rational numbers. We can then reduce each effect to linear inequalities. For example, $\rho_1 \leq \rho_2$ is reduced to the set of constraints $\{\eta_{\rho_1,1} \leq \eta_{\rho_2,1}, \ldots, \eta_{\rho_1,m} \leq \eta_{\rho_2,m}\}$. $\alpha \notin FN(e)$ is replaced by: $\{\eta_{\rho,i} = 0 \mid \alpha \text{ occurs in } A_i\}$.

As in [6], the type inference algorithm runs in time polynomial in the size of a process under the following assumptions:

1. The simple type of each message occurring in the process is bound by a constant.
2. The arguments of begin/end-events cannot contain encrypted messages (of the form $\{M\}_K$).

Note that the first assumption ensures that the size of the effect constraints in step 1 is polynomial in the size of the given process. The first and second conditions ensure that the size of the set of relevant atomic effects is also polynomial, hence so is the size of the linear inequalities.

Example 4. Recall Example 1. In step 1, we first prepare the following template of type derivation for $Sender(ch, key)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_2; \rho_4 \vdash ch : \mathbf{Un} \qquad \dfrac{\Gamma_2; \rho_6 \vdash key : T \qquad \dfrac{\Gamma_2; \rho_8 \vdash msg : \mathbf{N}(\rho_{14}) \quad \Gamma_1; \rho_9 \vdash n : \mathbf{N}(\rho_{13})}{\Gamma_2; \rho_7 \vdash (msg, n) : \mathbf{N}(\rho_{11}) \times \mathbf{N}(\rho_{12})}}{\Gamma_2; \rho_5 \vdash \{(msg, n)\}_{key} : \mathbf{N}(\rho_{10})}}{\Gamma_2; \rho_3 \vdash ch!\{msg, n\}_{key}}}{\Gamma_2; \rho_2 \vdash \mathbf{begin}\ msg. \cdots}}{\Gamma_1; \rho_1 \vdash (\nu msg) \cdots}}{\Gamma_0; \rho_0 \vdash Sender(ch, key)}$$

Here, T, Γ_0, Γ_1, and Γ_2 are given by:

$$\begin{array}{ll} T = \mathbf{Key}(\mathbf{N}(\rho_{15}) \times \mathbf{N}(\rho_{16})) & \Gamma_0 = ch : \mathbf{Un}, key : T \\ \Gamma_1 = \Gamma_0, n : \mathbf{N}(\rho_n) & \Gamma_2 = \Gamma_1, msg : \mathbf{N}(\rho_{msg}) \end{array}$$

From the derivation tree, we obtain the following constraints:

$$\begin{array}{l} \rho_1 \leq \rho_0 \qquad \rho_2 \leq \rho_1 + [\mathbf{chk}\langle msg \rangle \mapsto 1] \qquad \rho_3 \leq \rho_2 + [\mathbf{end}\langle msg \rangle \mapsto 1] \\ \rho_4 = \rho_6 = [\,] \qquad \rho_7 \leq \rho_5 \leq \rho_3 \qquad \rho_8 + \rho_9 \leq \rho_7 \\ \mathbf{pub}(\mathbf{N}(\rho_n)) \qquad \mathbf{gen}(\mathbf{N}(\rho_{msg})) \qquad \mathbf{N}(\rho_{15}) \times \mathbf{N}(\rho_{16}) = \mathbf{N}(\rho_{11}) \times \mathbf{N}(\rho_{12}) \\ \mathbf{N}(\rho_{11}) = \mathbf{N}(\rho_{14}) \qquad \mathbf{N}(\rho_{13}) = [msg/0]\mathbf{N}(\rho_{12}) \qquad \rho_n + \rho_9 = \rho_{13} \\ FN(\rho_{15}) \subseteq \{ch\} \qquad FN(\rho_{16}) \subseteq \{ch, 0\} \qquad FN(\rho_n) \subseteq \{ch, key\} \qquad \cdots \end{array}$$

(The constraints on the last line come from the well-formedness conditions of type judgments.) The constraints on types can be easily reduced to those on effects: for example, $\mathbf{pub}(\mathbf{N}(\rho_n))$ is replaced by $\rho_n = [\,]$.

By analyzing the effect constraints generated from the whole process *System*, we can infer that the relevant atomic effects are $S = \{\mathbf{end}\langle \alpha \rangle, \mathbf{chk}\langle \alpha \rangle \mid \alpha \in \{0, msg, non, non', m\}\}$. Let $\rho_i(A) = \eta_{i,A}$ for $A \in S$. Then, we can generate linear inequalities from the effect constraints. For example, from $\rho_2 \leq \rho_1 + [\mathbf{chk}\langle msg \rangle \mapsto 1]$, we obtain the following linear inequalities:

$$\eta_{2,\mathbf{chk}\langle msg \rangle} \leq \eta_{1,\mathbf{chk}\langle msg \rangle} + 1 \qquad \forall A \in S \setminus \{\mathbf{chk}\langle msg \rangle\}. \eta_{2,A} \leq \eta_{1,A}$$

5 Experiments

We have implemented a prototype protocol verifier `SpiCA` based on our type system. The implementation is available from `http://www.kb.ecei.tohoku.ac.jp/~koba/spica/`. The system takes a protocol description without type annotations as an input. If the input is well-typed, the system annotates it with

Processes	Typing	#EC	#LC	Time (ms)
`nonce-handshake`	yes	49	13	20
`flawed-handshake`	no	45	0	20
`HalfCap`	yes	60	14	30
`woo-lam`	yes	273	311	50
`flawed-wide-mouth`	no	239	1208	90
`wide-mouth`	yes	349	1328	100
`otway-ree`	yes	462	2143	180

Table 1. Benchmark results

types and effects; otherwise, it just reports that the input is ill-typed. The current system uses simplex method routines of the GLPK library [9] (via ocaml-glpk, `http://ocaml-glpk.sourceforge.net/`) to solve linear inequalities; thus, the implementation may suffer from exponential time complexity in the worst-case.

Table 1 summarizes the results of preliminary experiments. The experiments are conducted on a machine with an Intel(R) Pentium(R) 1.2GHz processor and 500MB memory. The column "Typing" shows whether or not the processes were judged to be well-typed. The columns "#EC" and "#LC" respectively show the number of effect constraints and that of linear inequalities generated in Steps 1 and 2 of the algorithm. The column "Time" shows the running time. The process `nonce-handshake` is the system in Example 1, while `flawed-handshake` is a flawed version obtained from `nonce-handshake` by removing the check operation. The process `HalfCap` is the one discussed in Section 3.4. The other protocols were taken from Gordon and Jeffrey's paper [2]. The process `woo-lam` is a corrected version of Woo and Lam's protocol. The processes `flawed-wide-mouth` and `wide-mouth` are flawed and corrected versions of Abadi and Gordon's variant of wide mouth frog. The process `otway-ree` is Abadi and Needham's variant of Otway and Ree's key exchange protocol.

All the processes were correctly verified (or rejected as ill-typed in the case for the flawed protocols), and the inferred types and effects were as expected: for example, $\mathbf{Key}(\mathbf{N}([\mathbf{end}\langle x\rangle \mapsto 0.5]))$ was automatically inferred as the type of k in `HalfCap`.

In some cases, the number of linear constraints is smaller than that of effect constraints. That is because constraints (such as unification constraints) are simplified before the translation into linear constraints. In particular, for `flawed-handshake`, inconsistency is detected in the simplification phase for effect constraints.

6 Related Work

This paper combines Gordon and Jeffrey's work [2] on the type system for checking authenticity with our previous work [6] of using fractional effects to enable polynomial-time type inference for π-calculus with correspondence assertions.

The combination is non-trivial, however. Since Gordon and Jeffrey's type system has explicit type annotations and cast operations, non-trivial modifications of the type system were necessary to adapt our previous technique.

Gordon and Jeffrey later extended their type system to deal with asymmetric cryptographic protocols [7]. We expect that our approach can also be extended to deal with them.

Gordon, Hüttel, and Hansen [10] have also recently proposed a type inference algorithm for checking correspondence assertions in π-calculus. The algorithm checks one-to-many correspondence (in which there may be more than one end-events for each begin-event), rather than one-to-one correspondence considered in the present paper and our previous work [6]. Their algorithm is quite different from ours, and does not handle cryptographic primitives.

Bugliesi, Focardi, and Maffei [11, 12] have proposed type-based static analyses for authentication protocols that are closely related to Gordon and Jeffrey's type systems. They [13] later introduced an algorithm for automatically inferring *tags* (which roughly correspond to Gordon and Jeffrey's types [2, 7]). Their inference algorithm is based on exhaustive search of potential taggings by backtracking. Our type inference algorithm is therefore more efficient theoretically. The advantage of our polynomial-time type inference may not be so important in analyzing abstract descriptions of cryptographic protocols, which are usually very short. The advantage may be more significant for analyzing the source code of cryptographic protocols [14].

Blanchet [15] also proposed automated techniques for checking checking correspondence assertions in cryptographic protocols. An advantage of our type-based approach is that the result of type inference gives a better explanation of why the protocol is safe. Blanchet [16] has recently proposed a quite different technique for authenticity verification. His technique can guarantee soundness in the computational model, rather than in the formal model with the assumption of perfect encryption.

The idea of using rational numbers in type systems goes back to the work of Boyland [17], and has been extensively studied by Terauchi [18–20].

7 Conclusion

We have modified Gordon and Jeffrey's type system for checking correspondence assertions in cryptographic protocols, and obtained a type inference algorithm, which serves as an algorithm for automated verification of cryptographic protocols. Under certain reasonable assumptions, the algorithm runs in time polynomial in the size of an input process.

Acknowledgment We would like to thank Koki Nishizawa for comments and discussions on this work, and Kohei Suenaga for his help in using the GLPK library.

References

1. Gordon, A.D., Jeffrey, A.: Typing correspondence assertions for communication protocols. Theor. Comput. Sci. **300** (2003) 379–409
2. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. Journal of Computer Security **11**(4) (2003) 451–520
3. Gordon, A.D., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. In: 15th IEEE Computer Security Foundations Workshop (CSFW-15). (2002) 77–91
4. Woo, T.Y., Lam, S.S.: A semantic model for authentication protocols. In: RSP: IEEE Computer Society Symposium on Research in Security and Privacy. (1993) 178–193
5. Haack, C., Jeffrey, A.: Cryptyc. `http://www.cryptyc.org/` (2004)
6. Kikuchi, D., Kobayashi, N.: Type-based verification of correspondence assertions for communication protocols. In: Proceedings of APLAS 2007. Volume 4807 of LNCS., Springer-Verlag (2007) 191–205
7. Gordon, A.D., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. Journal of Computer Security **12**(3-4) (2004) 435–483
8. Abadi, M., Gordon, A.D.: A Calculus for Cryptographic Protocols: The Spi Calculus. Information and Computation **148**(1) (January 1999) 1–70
9. GNU Linear Programming Kit. `http://www.gnu.org/software/glpk`
10. Gordon, A.D., Hüttel, H., Hansen, R.R.: Type inference for correspondence types. In: 6th International Workshop on Security Issues in Concurrency (SecCo 2008). (2008)
11. Bugliesi, M., Focardi, R., Maffei, M.: Compositional analysis of authentication protocols. In: ESOP. Volume 2986 of LNCS., Springer-Verlag (2004) 140–154
12. Bugliesi, M., Focardi, R., Maffei, M.: Authenticity by tagging and typing. In: Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering (FMSE 2004). (2004) 1–12
13. Focardi, R., Maffei, M., Placella, F.: Inferring authentication tags. In: Proceedings of the Workshop on Issues in the Theory of Security (WITS 2005). (2005) 41–49
14. 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). (2008) 17–32
15. Blanchet, B.: From Secrecy to Authenticity in Security Protocols. In: 9th International Static Analysis Symposium (SAS'02). Volume 2477 of LNCS., Springer-Verlag (2002) 342–359
16. Blanchet, B.: Computationally sound mechanized proofs of correspondence assertions. In: 20th IEEE Computer Security Foundations Symposium (CSF'07). (2007) 97–111
17. Boyland, J.: Checking interference with fractional permissions. In: Proceedings of SAS 2003. Volume 2694 of LNCS., Springer-Verlag (2003) 55–72
18. Terauchi, T., Aiken, A.: Witnessing side-effects. In: Proc. of ICFP, ACM (2005) 105–115
19. Terauchi, T., Aiken, A.: A capability calculus for concurrency and determinism. ACM Trans. Prog. Lang. Syst. **30**(5) (2008)
20. Terauchi, T.: Checking race freedom via linear programming. In: Proc. of PLDI. (2008) 1–10