Affine Refinement Types for Authentication and Authorization

M. Bugliesi[1], S. Calzavara[1], F. Eigner[2], and M. Maffei[2]

[1] Università Ca' Foscari Venezia
{bugliesi,calzavara}@dais.unive.it
[2] Saarland University
{eigner,maffei}@cs.uni-saarland.de

Abstract. Refinement type systems have proved very effective for security policy verification in distributed authorization systems. In earlier work [12], we have proposed an extension of existing refinement typing techniques to exploit sub-structural logics and affine typing in the analysis of *resource aware* authorization, with policies predicating over access counts, usage bounds and resource consumption. In the present paper, we show that the invariants that we enforced by means of ad-hoc typing mechanisms in our initial proposal can be internalized, and expressed directly as proof obligations for the underlying affine logical system. The new characterization leads to a more general, modular design of the system, and is effective in the analysis of interesting classes of authentication protocols and authorization systems.

1 Introduction

Authorization policies constitute an effective device for security specification in distributed protocols and systems [3, 5]. In language-based security, such policies are specified by means of code annotations marking authorization-sensitive program points with logical formulas that serve as *assumptions* and *assertions*: the former express the credentials available at the clients, and track the clients' authorization requests; the latter are employed as resource guards at the server, and express the conditions required to accept the authorization requests. The annotations have no semantic import, and only serve the verification process: to show a system *safe*, i.e., to prove that it complies with a given policy, one must prove that all the active (unguarded) assertions at a given execution step are entailed by the active assumptions at that step, for every possible system run. Proving a system *robustly safe* amounts to prove that the same property holds in the presence of other, possibly malicious agents [6].

Safety (and robust safety) proofs for annotated specifications such as those of our present interests can be carried out statically, and effectively, with refinement typing systems [15, 4, 6, 7]. Refinement types [16] are dependent types of the form $\{x : T \mid F(x)\}$: a value M of this type is a value of type T such that the formula $F\{M/x\}$ holds. In type-based authorization systems, the refinement formulas are employed to capture the dynamic exchange of credentials required for authorization: this is accomplished by encoding such credentials as formulas that refine the payload types of the cryptographic keys involved in the authorization protocols.

Depending on the authorization properties expected, different logical frameworks may be appealed to for specification and verification. Our focus in the present paper is on *resource conscious* policies such as those governing large classes of modern authorization frameworks, based on consumable credentials, access counts and/or usage bounds. For such policies, and for the strong authentication protocols supporting them, one may resort to sub-structural (e.g., linear or affine) logics [17, 25] for specification. Correspondingly, typing systems with linear (or affine) refinements may be employed to achieve static accounts of the desired safety proofs.

In our earlier work in [12], we made a first step towards the design of a sound system for resource-sensitive authorization, drawing on techniques from typing systems for authentication and an affine extension of existing refinement typing systems. Here we make a step further, and show that the invariants that were enforced by means of ad-hoc mechanisms in our original proposal can be internalized into the underlying affine logical system, and expressed directly as proof obligations for the logic. Besides shedding new light on the logical foundations of the cryptographic patterns for authentication and distributed authorization, the new characterization is interesting, and promising, as it leads to a more modular and more powerful typing system.

Plan of the paper Section 2 reviews the background material. Section 3 gives an overview of our approach. Section 4 provides a detailed description of the type system and its main properties. Section 5 demonstrates the effectiveness of the type system with two small, yet significant, examples. Section 6 concludes.

2 Background

We give a brief review of the relevant components of our approach: affine logics for policy specification, applied pi-calculus for protocol description, and refinement typing systems for analysis and verification.

Affine logic. We focus on the following fragment of intuitionistic affine logic [25]:

$$F \ ::= \ A \ \mid \ F \otimes F \ \mid \ F \multimap F \ \mid \ \forall x.F \ \mid \ !F$$

This is the multiplicative fragment of affine logic, extended with the exponential modality to express persistent truths. We presuppose an underlying signature of predicate symbols which includes the binary equality predicate, and a countably infinite set of terms. Atomic formulas, noted A in the above productions, are built around predicates applied to terms, as in $p(M_1, \ldots, M_n)$; term equality uses infix notation, as in $M = N$. We assume familiarity with the resource interpretation of linear logic, by which each formula denotes a resource which is consumed once used in a derivation. In sequent calculus presentations of linear logic, that is achieved by dispensing with the structural rules of weakening and contraction, and by a careful manipulation of the environment, as exemplified in two representative rules, below:

$$\frac{\mathit{\Gamma}_1 \vdash F \qquad \mathit{\Gamma}_2 \vdash G}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash F \otimes G}\ (\otimes\text{-Right}) \qquad \frac{\mathit{\Gamma}_1 \vdash F \qquad \mathit{\Gamma}_2, G \vdash H}{\mathit{\Gamma}_1, F \multimap G, \mathit{\Gamma}_2 \vdash H}\ (\multimap\text{-Left})$$

Affine logic is a variant of linear logic which admits the weakening rule, whereby $\mathit{\Gamma}, F \vdash G$ is derivable when so is $\mathit{\Gamma} \vdash G$. As a result, proofs in affine logics must use each formula *at most* once (as to opposed to *exactly* once as in linear logic).

Applied pi-calculus. We specify protocols in a dialect of the applied pi-calculus [2], in which destructors are only used in let-expressions and may not occur in arbitrary terms [8]. We presuppose an underlying set of constructors and two countable sets of names (a, b, c, k, m, n) and variables (w, x, y, z), and let u range over names or variables uniformly. The syntax of terms M, N is as follows:

$$\begin{array}{rcl} M, N & ::= & a \mid x \mid \mathit{ek}(M) \mid \mathit{vk}(M) \mid \mathit{inl}(M) \mid \mathit{inr}(M) \mid (M, N) \\ & \mid & \mathit{enc}(M, N) \mid \mathit{sign}(M, N) \mid \mathit{senc}(M, N). \end{array}$$

Unary constructors include *ek* and *vk* to form encryption/verification keys from the corresponding decryption/signing keys, and *inl* and *inr* to construct tagged unions (see Section 4). Binary constructors comprise pairs and *senc*, *enc* and *sign* for symmetric, asymmetric encryption and digital signature, respectively. Destructors, ranged over by g, are partial functions to decompose terms. They include the unary *casel* and *caser* to deconstruct tagged unions, *sdec*, *dec* and *ver* for symmetric, asymmetric decryption and signature verification, respectively. For technical reasons, pairs are not decomposed by destructors, but with pattern matching within a specific process form (discussed below). Destructor evaluation may succeed and return a term, noted $g(\widetilde{M}) \Downarrow N$, or fail. The semantics of destructors is as expected, e.g., we have $\mathit{dec}(\mathit{enc}(M, \mathit{ek}(K)), K) \Downarrow M$ and $\mathit{ver}(\mathit{sign}(N, K'), \mathit{vk}(K')) \Downarrow N$.

The syntax of processes P, Q is defined as follows, in terms of two syntactic categories of *actions* A, and proper *processes* P, Q (the distinction is technically convenient in the definition of the typing rules):

$$\begin{array}{rcl} A & ::= & \mathbf{0} \mid \mathit{in}(M, x).P \mid *\mathit{in}(M, x).P \mid \mathit{out}(M, N).P \mid A|A \\ & \mid & \mathit{if}\ (M = N)\ \mathit{then}\ P\ \mathit{else}\ Q \mid \mathit{let}\ (x, y) = M\ \mathit{in}\ P\ \mathit{else}\ Q \\ & \mid & \mathit{let}\ x = g(\widetilde{M})\ \mathit{in}\ P\ \mathit{else}\ Q \mid \ll F. \\ \\ P, Q & ::= & A \mid P|Q \mid \mathit{new}\ a : T.P \mid \gg F. \end{array}$$

The scope of names and variables is delimited by restrictions, inputs and let expressions. The notions of free names and variables, denoted by *fn* and *fv* respectively, arise as expected. A process P is *closed* when $\mathit{fv}(P) = \emptyset$. Processes evolve according to the reduction relation $P \rightarrow Q$ ($\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$). The definition of reduction is standard: $\mathbf{0}$ is the stuck process; $new\ a : T.P$ creates a fresh name a and behaves as P; $in(M, x).P$ waits for a message N on channel M and then behaves as $P\{N/x\}$; $*in(M, x).P$ acts as an unbounded replication of $in(M, x).P$; $out(M, N).P$ outputs N on M, synchronously, and then behaves as P; $P|Q$ is the parallel composition of P and Q; $if\ (M = N)\ then\ P\ else\ Q$ reduces to P if M is syntactically equal to N, to Q otherwise; $let\ (x, y) = M\ in\ P\ else\ Q$ behaves as $P\{M_1/x\}\{M_2/y\}$ when M is (M_1, M_2), as Q otherwise; finally, $let\ x = g(\widetilde{M})\ in\ P\ else\ Q$ acts as $P\{N/x\}$ if $g(\widetilde{M}) \Downarrow N$, as Q otherwise. Assumptions ($\gg F$) and assertions ($\ll F$) are inert process forms, built around the formulas of our affine logic, that express policy annotations.

Definition 1 (Safety). *A closed process P is* safe *iff whenever $P \rightarrow^* new\ \widetilde{a} : \widetilde{T}.(\ll G_1 \mid \ldots \mid \ll G_n \mid Q)$ one has $Q \equiv\ \gg F_1 \mid \ldots \mid \gg F_m \mid A$, with A containing no top-level assertions, and $F_1, \ldots, F_m \vdash G_1 \otimes \ldots \otimes G_n$.*

Unlike most of the existing definitions of safety [15, 4, 6, 7], we take the tensor product of *all* the active assertions. That is required to remain faithful to the chosen logical framework, and enforce an affine use of each active assumption in our safety proofs. The definition extends readily to account for the presence of opponents. We define an opponent as a closed, *Un*-typed (cf. Section 4) process that does not contain any assertion.

Definition 2 (Robust Safety). *A closed process P is* robustly safe *iff $P \mid O$ is safe for every opponent O.*

The restriction to *Un*-typed processes is standard and does not involve any loss of generality; we ban assertions from opponent code, because otherwise an opponent could trivially break the safety property we target.

Refinement typing systems for distributed authorization. We review the main ideas and intuitions with a simple example, inspired by [15], of an on-line bookstore system governed by the following authorization policy:

$$\mathcal{A} \triangleq\ !\forall u, b.\,(Order(u, b) \multimap Clear(u, b))$$

The policy is stated as a persistent (reusable) formula: it establishes that an e-book order can be cleared for a user if that user has indeed ordered the e-book (note, in particular, the use of multiplicative implication to express the desired *injective* correspondence between order and clearance). The components are described by the annotated code below[3]

$$\begin{array}{ll} user & :: \gg Order(user, book) \mid out(net, (user, sign((user, book), k_{user}))) \\ bookstore & :: *in(net, (x_u, y)). \\ & \quad let\,(x_{vk}, x_{ek}) = keys(x_u)\,in\ let\,(x_u, x_b) = ver(y, x_{vk})\,in \\ & \quad \ll Clear(x_u, x_b) \mid out(net, enc(url(x_b), x_{ek})) \end{array}$$

[3] We assume that the bookstore keeps track of the public (encryption and verification) keys of each registered user, so that $keys(user) = (vk(k_{user}), ek(k_{user}))$. Also, for readability, we abuse the notation and use pattern-matching on input.

Consider now the system $\gg \mathcal{A} \mid \mathit{user} \mid \mathit{bookstore}$. In a system run, *user* authenticates her order of *book* to the *bookstore* by signing the request, and correspondingly *assumes* the formula $\mathit{Order}(\mathit{user}, \mathit{book})$ to declare her intention. The bookstore, in turn, receives the data from its input channel, verifies the signature and *asserts* the formula $\mathit{Clear}(\mathit{user}, \mathit{book})$ as a guard to clear the order. The system is safe, as the guard is entailed by the policy $\mathcal{A}$ and the assumption $\mathit{Order}(\mathit{user}, \mathit{book})$, which is available when the assertion $\mathit{Clear}(\mathit{user}, \mathit{book})$ is unleashed at top-level.

In a refinement type system, a safety proof can be derived by relying on the types of cryptographic keys. Key types have the general form $\mathit{Key}(\{x : T \mid C(x)\})$ representing keys with payload of (structural) type T for which the formula (credential) C may be assumed to hold. Given $k : \mathit{Key}(\{x : T \mid C(x)\})$, packaging a value m with k, as in $\mathit{enc}(m, k)$, typechecks provided that the formula $C(m)$ can be proved at the source site, using the assumptions available in the typing context associated with that site. Dually, extracting the payload from a encrypted packet, as in $\mathit{dec}(y, k)$, justifies the assumption of the formula $C(y)$ conveyed by the key type, hence the use of $C(y)$ in a proof of the credentials acting as access guards. In our example, we may use the type $\mathit{Key}(x_u : T_u, \{x_b : T_b \mid \mathit{Order}(x_u, x_b)\})$ for k_{user} to convey the credential $\mathit{Order}(x_u, x_b)$ from the user process to the bookstore site, and derive a static safety proof based on that.

3 Authorization, authentication and affine refinements

Continuing with our example, consider extending the system with the new component:

$$\mathit{dup} :: * \, \mathit{in}(\mathit{net}, x).(\mathit{out}(\mathit{net}, x) \mid \mathit{out}(\mathit{net}, x))$$

to form the composition $\gg \mathcal{A} \mid \mathit{user} \mid \mathit{bookstore} \mid \mathit{dup}$. Unlike the original system, the extended one is unsafe (hence the original is not robustly safe), as the presence of the *dup* process causes *bookstore* to clear each order twice. Clearly, the problem arises from the absence of an authentication mechanism providing adequate guarantees of timeliness for the orders. The effect is captured by our definition of safety. To see that, first observe that a reduction sequence exists that unleashes two assertions $\mathit{Clear}(\mathit{user}, \mathit{book})$ for a single assumption $\mathit{Order}(\mathit{user}, \mathit{book})$. Then, note that $\mathit{Order}(\mathit{user}, \mathit{book}), \mathcal{A} \not\vdash \mathit{Clear}(\mathit{user}, \mathit{book}) \otimes \mathit{Clear}(\mathit{user}, \mathit{book})$, as the assumption $\mathit{Order}(\mathit{user}, \mathit{book})$ is consumed in the proof of either of the two clearing assertions at the bookstore, thereby causing the derivation of the second assertion to fail.

Strong authentication is a long studied problem in static protocol analysis and verification. First formalized as *injective agreement* in [23], it has subsequently been approached with a variety of typing techniques, targeted at the analysis of various low-level mechanisms (timestamps, nonce handshakes, and session keys) [18, 19, 9, 10, 20, 11]. Though such mechanisms are fundamental building blocks for distributed authorization frameworks, little (if any) of the work on strong authentication has resurfaced in existing typing systems for authorization [14, 15, 4, 6, 7]. Our proposal in the present paper aims at reconciling these two streams of research, by building a unifying foundation for authentication and authorization, based on an affine refinement type system.

Just like traditional refinement typing systems, we employ key types to capture the transfer of authorization credentials within a protocol. However, since our refinements are affine formulas, we must control the use of keys so as to protect against any unintended duplication of the refinements, upon decryption. Specifically, when transferring a message $m : T$ packaged with, say, $k : \mathit{Key}(\{x : T \mid C(x)\})$ we must ensure that each extraction of $C(x)$ by the receiver correspond to a derivation of $C(m)$ at the source site. To accomplish that, our type system protects the refinement $C(x)$ with a *guard*, as in:

$$k : \mathit{Key}(w : U, \{x : T \mid G(w) \multimap C(x)\})$$

where $G(w)$ is a receiver-controlled formula that must be proved to derive the credential $C(x)$. In a nonce-handshake protocol, w represents the challenger-generated nonce, call it n, and $G(n)$ is the corresponding guard assumed by the challenger. A responder willing to prove the possession of a credential $C(m)$ for the payload m will be able to do so, as follows. Upon receiving the nonce, the responder transmits the pair (n, m) under the key k: that's possible when the responder has (or may derive) $C(m)$, because $C(m) \vdash G(n) \multimap C(m)$ in affine logic. At the challenger end, extracting the payload (w, x) and checking that $w = n$ makes it possible to derive $C(x)$, as $G(n), w = n, G(w) \multimap C(x) \vdash C(x)$. If we can ensure that $G(n)$ can be proved at most once, we also ensure that $C(x)$ is derived at most once, as desired.

Though the details vary for the different low-level mechanisms, the core intuitions we just outlined apply uniformly: data exchanged over the network is inherently exposed to replays, hence their credentials must be protected so that copying the data does not duplicate the credentials. In the type system, that is accomplished by embedding the credentials into multiplicative implications guarded by system-controlled formulas, which are built around *reserved* predicate symbols, and are guaranteed to be assumed in at most one position in the protocol code. As a result, key refinements become safely *copyable*, as the system-controlled guards guarantee that the credentials they embed are unleashed at most once, irrespective of any duplication the refinement may undergo.

In the next section, we provide full details of these mechanisms.

4 The type system

The syntax of types T, U, V is defined by the following grammar.

$$\begin{array}{rll} T, U, V ::= & \mathit{Un} \mid \mathit{Private} \mid \mathit{Ch}(T) \mid \{x : T \mid F\} \mid (x : T, U) \mid T + U & \\ \mid & \eta \mathit{Key}_{(x)}(T) & \eta \in \{\mathit{Enc}, \mathit{Dec}, \mathit{Sig}, \mathit{Ver}, \mathit{Sym}\} \\ \mid & \eta \mathit{Pkt}_{(x)}(T) & \eta \in \{\mathit{Enc}, \mathit{Sig}, \mathit{Sym}\}. \end{array}$$

The variable x is bound in $\{x : T \mid F\}$ with scope F, in $(x : T, U)$ with scope U, and in $\eta \mathit{Key}_{(x)}(T)$ and $\eta \mathit{Pkt}_{(x)}(T)$ with scope T. Un is the type of data

$$\text{(TYPE-BASE)}\ \frac{fnfv(T) \subseteq dom(\Gamma) \quad \Gamma \vdash \diamond \quad T \neq \eta Key_{(x)}(U)}{\Gamma \vdash T} \qquad \text{(TYPE-KEY)}\ \frac{fnfv(T) \setminus \{x\} \subseteq dom(\Gamma) \quad \Gamma \vdash \diamond \quad T \text{ copyable}}{\Gamma \vdash \eta Key_{(x)}(T)} \qquad \text{(ENV-EMPTY)}\ \frac{}{\varepsilon \vdash \diamond}$$

$$\text{(ENV-FORM)}\ \frac{\Gamma \vdash \diamond \quad fnfv(F) \subseteq dom(\Gamma)}{\Gamma, F \vdash \diamond} \qquad \text{(ENV-BIND)}\ \frac{u \notin dom(\Gamma) \quad \Gamma \vdash T \quad T \text{ a copyable, non-refinement type}}{\Gamma, u : T \vdash \diamond}$$

Table 1. Well-formed types and environments

coming from / flowing to the opponent (standard since [1]). *Private* is the type of untainted, secret data; $Ch(T)$ the type of channels with T payload; $\{x : T \mid F\}$ the type of $M : T$ such that $F\{M/x\}$ holds. A pair (M, N) has type $(x : T, U)$ if M has type T and N has type $U\{M/x\}$. A term of type $T + U$ is either $inl(M)$ where M has type T, or $inr(M)$ where M has type U. Finally, we devise two new types for cryptographic material – $\eta Key_{(x)}(T)$ and $\eta Pkt_{(x)}(T)$ – for keys with T payload and ciphertexts with T payload, respectively[4]. In both cases the binder x acts as a placeholder for the encryption key or the verification key: this technical device, first proposed in [13], is very effective and convenient to achieve a uniform treatment for nonce handshakes and session keys in our type system.

Typing environments and well-formed types. Typing environments, noted Γ, collect bindings for names and variables, as usual, and formulas occurring in assumptions. The domain of Γ, noted $dom(\Gamma)$, is the set of the values bound to a type in Γ. $forms(\Gamma)$ denotes the multiset of the formulas occurring in Γ. $bindings(\Gamma)$ is the environment obtained by erasing all the formulas from Γ. ε is the empty environment. Well-formed types and environments are defined in terms of the notions of *copyable* formulas and types, given below.

Definition 3 (Copyable Formulas and Types). *A formula F is copyable if it has either of the two forms $p(M_1, \ldots, M_n) \multimap F'$ with p reserved, or $!F'$. Copyable types, then, are defined inductively as follows:*

- *$Un, Private, Ch(T), \eta Key_{(x)}(T)$ and $\eta Pkt_{(x)}(T)$ are copyable;*
- *$\{x : T \mid F\}$, $(x : T, U)$ and $T + U$ are copyable if so are T, U and F.*

The rules for types and environments are in Table 1. Notice that, by (TYPE-KEY), well-formed key types can only convey *copyable* payloads: hence, formulas may occur as refinements of key types only if they are guarded by system-reserved predicates, or they are prefixed by a bang modality: in the former case, the

[4] When x does not occur in T we often omit the binder and write simply $\eta Key(T)$, and $\eta Pkt(T)$, to ease the notation.

guard protects them against uncontrolled replication, in the latter, replication is harmless as the formula may be duplicated in the logic as well. A similar mechanism is enforced in the type system for injective agreement in [21].

Type environments only include bindings for non-refinement typed names and variables: that does not involve any loss of expressive power, as we may simply define the environment $\Gamma, u : \{x : T \mid F\}$ as $\Gamma, u : T, F\{u/x\}$. Also, type bindings must introduce copyable types, to protect against the unintended duplication of affine refinements (occurring in dependent pair or disjoint union types) upon the *environment splitting* distinctive of sub-structural type systems. We say that Γ splits as Γ_1 and Γ_2 ($\Gamma = \Gamma_1 \bullet \Gamma_2$) when $\mathit{bindings}(\Gamma) = \mathit{bindings}(\Gamma_1) = \mathit{bindings}(\Gamma_2)$ and $\mathit{forms}(\Gamma) = \mathit{forms}(\Gamma_1), \mathit{forms}(\Gamma_2)$. More in general, we write $\Gamma = \Gamma_1 \bullet \ldots \bullet \Gamma_n$ when $\Gamma = \Gamma' \bullet \Gamma_n$ and $\Gamma' = \Gamma_1 \bullet \ldots \bullet \Gamma_{n-1}$. Finally, we write $\Gamma \vdash F$ whenever $\mathit{forms}(\Gamma) \vdash F$, provided that $\Gamma \vdash \diamond$ and $\mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma)$.

$$\text{(Term-Env)}\ \frac{\Gamma \vdash \diamond \qquad u : T \in \Gamma}{\Gamma \vdash u : T} \qquad \text{(Term-EncKey)}\ \frac{\Gamma \vdash M : \mathit{DecKey}_{(x)}(T)}{\Gamma \vdash \mathit{ek}(M) : \mathit{EncKey}_{(x)}(T)} \qquad \text{(Term-VerKey)}\ \frac{\Gamma \vdash M : \mathit{SigKey}_{(x)}(T)}{\Gamma \vdash \mathit{vk}(M) : \mathit{VerKey}_{(x)}(T)}$$

$$\text{(Term-Pair)}\ \frac{\Gamma_1 \vdash M : T \qquad \Gamma_2 \vdash N : U\{M/x\}}{\Gamma_1 \bullet \Gamma_2 \vdash (M, N) : (x : T, U)} \qquad \text{(Term-Refine)}\ \frac{\Gamma_1 \vdash M : T \qquad \Gamma_2 \vdash F\{M/x\}}{\Gamma_1 \bullet \Gamma_2 \vdash M : \{x : T \mid F\}}$$

$$\text{(Term-AEnc)}\ \frac{\Gamma_1 \vdash M : T\{N/x\} \qquad \Gamma_2 \vdash N : \mathit{EncKey}_{(x)}(T)}{\Gamma_1 \bullet \Gamma_2 \vdash \mathit{enc}(M, N) : \mathit{EncPkt}_{(x)}(T)} \qquad \text{(Term-Left)}\ \frac{\Gamma \vdash M : T \qquad \Gamma \vdash U}{\Gamma \vdash \mathit{inl}(M) : T + U}$$

$$\text{(Term-Sign)}\ \frac{\Gamma_1 \vdash M : T\{\mathit{vk}(N)/x\} \qquad \Gamma_2 \vdash N : \mathit{SigKey}_{(x)}(T)}{\Gamma_1 \bullet \Gamma_2 \vdash \mathit{sign}(M, N) : \mathit{SigPkt}_{(x)}(T)} \qquad \text{(Term-Right)}\ \frac{\Gamma \vdash M : U \qquad \Gamma \vdash T}{\Gamma \vdash \mathit{inr}(M) : T + U}$$

$$\text{(Term-SEnc)}\ \frac{\Gamma_1 \vdash M : T\{N/x\} \qquad \Gamma_2 \vdash N : \mathit{SymKey}_{(x)}(T)}{\Gamma_1 \bullet \Gamma_2 \vdash \mathit{senc}(M, N) : \mathit{SymPkt}_{(x)}(T)}$$

Table 2. Typing rules for terms

Typing rules for terms. Table 2 details the typing rules for terms. We omit the rules that define the subtype relation as well as the kinding rules for tainted and public types: all details can be found in [12]. The novel rules for cryptographic packets (Term-AEnc), (Term-Sign) and (Term-SEnc) exploit a form of dependent typing to track the shared information between encryption and decryption keys (respectively, signing and verification keys) [13].

(Proc-Out)
$$\frac{\mathit{\Gamma}_1 \vdash M : \mathit{Ch}(T) \qquad \mathit{\Gamma}_2 \vdash N : T \qquad \mathit{\Gamma}_3 \vdash P}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \bullet \mathit{\Gamma}_3 \vdash \mathit{out}(M,N).P}$$

(Proc-In)
$$\frac{\mathit{\Gamma}_1 \vdash M : \mathit{Ch}(T) \qquad \mathit{\Gamma}_2, x : T \vdash P}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \vdash \mathit{in}(M,x).P}$$

(Proc-Repl)
$$\frac{\mathit{\Gamma}_1 \vdash M : \mathit{Ch}(T) \qquad \mathit{\Gamma}_2, x : T \vdash P \qquad \mathit{\Gamma}_2 \text{ copyable}}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \vdash *\mathit{in}(M,x).P}$$

(Proc-Stop)
$$\frac{\mathit{\Gamma} \vdash \diamond}{\mathit{\Gamma} \vdash \mathbf{0}}$$

(Proc-Cond)
$$\frac{\begin{array}{c}\mathit{\Gamma}_1 \vdash M : T \qquad \mathit{\Gamma}_2 \vdash N : T \\ \mathit{\Gamma}_3, !(M = N) \vdash P \qquad \mathit{\Gamma}_3 \vdash Q\end{array}}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \bullet \mathit{\Gamma}_3 \vdash \mathit{if}\ (M = N)\ \mathit{then}\ P\ \mathit{else}\ Q}$$

(Proc-Split)
$$\frac{\begin{array}{c}\mathit{\Gamma}_1 \vdash M : (x : T, U) \\ \mathit{\Gamma}_2, x : T, y : U \vdash P \qquad \mathit{\Gamma}_2 \vdash Q\end{array}}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \vdash \mathit{let}\ (x,y) = M\ \mathit{in}\ P\ \mathit{else}\ Q}$$

(Proc-Case-Left)
$$\frac{\mathit{\Gamma}_1 \vdash M : T + U \qquad \mathit{\Gamma}_2, x : T \vdash P \qquad \mathit{\Gamma}_2 \vdash Q}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \vdash \mathit{let}\ x = \mathit{casel}(M)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

(Proc-Case-Right)
$$\frac{\mathit{\Gamma}_1 \vdash M : T + U \qquad \mathit{\Gamma}_2, x : U \vdash P \qquad \mathit{\Gamma}_2 \vdash Q}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \vdash \mathit{let}\ x = \mathit{caser}(M)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

(Proc-ADec)
$$\frac{\begin{array}{c}\mathit{\Gamma}_1 \vdash M : \mathit{EncPkt}_{(y)}(T) \\ \mathit{\Gamma}_2 \vdash N : \mathit{DecKey}_{(y)}(T) \qquad \mathit{\Gamma}_3, x : T\{\mathit{ek}(N)/y\} \vdash P \qquad \mathit{\Gamma}_3 \vdash Q\end{array}}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \bullet \mathit{\Gamma}_3 \vdash \mathit{let}\ x = \mathit{dec}(M,N)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

(Proc-Ver)
$$\frac{\mathit{\Gamma}_1 \vdash M : \mathit{SigPkt}_{(y)}(T) \qquad \mathit{\Gamma}_2 \vdash N : \mathit{VerKey}_{(y)}(T) \qquad \mathit{\Gamma}_3, x : T\{N/y\} \vdash P \qquad \mathit{\Gamma}_3 \vdash Q}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \bullet \mathit{\Gamma}_3 \vdash \mathit{let}\ x = \mathit{ver}(M,N)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

(Proc-SDec)
$$\frac{\begin{array}{c}\mathit{\Gamma}_1 \vdash M : \mathit{SymPkt}_{(y)}(T) \\ \mathit{\Gamma}_2 \vdash N : \mathit{SymKey}_{(y)}(T) \qquad \mathit{\Gamma}_3, x : T\{N/y\} \vdash P \qquad \mathit{\Gamma}_3 \vdash Q\end{array}}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \bullet \mathit{\Gamma}_3 \vdash \mathit{let}\ x = \mathit{sdec}(M,N)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

(Proc-Assert)
$$\frac{\mathit{\Gamma} \vdash F}{\mathit{\Gamma} \vdash \ll F}$$

(Proc-Par)
$$\frac{\mathit{\Gamma}_1 \vdash A_1 \qquad \mathit{\Gamma}_2 \vdash A_2}{\mathit{\Gamma}_1 \bullet \mathit{\Gamma}_2 \vdash A_1 \mid A_2}$$

(Proc-Extr)
$$\frac{P \rightsquigarrow [\mathit{\Gamma}' \ \| \ A] \qquad \mathit{\Gamma}, \mathit{\Gamma}' \vdash A \qquad \mathit{fnfv}(P) \subseteq \mathit{dom}(\mathit{\Gamma})}{\mathit{\Gamma} \vdash P}$$

(Weak)
$$\frac{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash P \qquad \mathit{\Gamma}_1, F, \mathit{\Gamma}_2 \vdash \diamond}{\mathit{\Gamma}_1, F, \mathit{\Gamma}_2 \vdash P}$$

(Contr)
$$\frac{\mathit{\Gamma}_1, !F, !F, \mathit{\Gamma}_2 \vdash P}{\mathit{\Gamma}_1, !F, \mathit{\Gamma}_2 \vdash P}$$

($\otimes$-Left)
$$\frac{\mathit{\Gamma}_1, F, G, \mathit{\Gamma}_2 \vdash P}{\mathit{\Gamma}_1, F \otimes G, \mathit{\Gamma}_2 \vdash P}$$

($\multimap$-Left)
$$\frac{\mathit{\Gamma}_1' \vdash F \qquad \mathit{\Gamma}_2', G \vdash P \qquad \mathit{\Gamma}_1, \mathit{\Gamma}_2 = \mathit{\Gamma}_1' \bullet \mathit{\Gamma}_2'}{\mathit{\Gamma}_1, F \multimap G, \mathit{\Gamma}_2 \vdash P}$$

($\forall$-Left)
$$\frac{\mathit{\Gamma}_1, F(M), \mathit{\Gamma}_2 \vdash P}{\mathit{\Gamma}_1, \forall x.F(x), \mathit{\Gamma}_2 \vdash P}$$

(!-Left)
$$\frac{\mathit{\Gamma}_1, F, \mathit{\Gamma}_2 \vdash P}{\mathit{\Gamma}_1, !F, \mathit{\Gamma}_2 \vdash P}$$

Table 3. Typing rules for actions and processes

(Extr-New)
$$\frac{P \rightsquigarrow [\Gamma \;\|\; A] \qquad T \in \{\mathit{Un}, \mathit{Private}, \mathit{Ch}(U), \mathit{SigKey}_{(x)}(U), \mathit{DecKey}_{(x)}(U), \mathit{SymKey}_{(x)}(U)\}}{\mathit{new}\ a : T.P \rightsquigarrow [a : T, \Gamma \;\|\; A]}$$

(Extr-Assume)
$$\frac{}{\gg F \rightsquigarrow [F \;\|\; \mathbf{0}]}$$

(Extr-Empty)
$$\frac{}{A \rightsquigarrow [\varepsilon \;\|\; A]}$$

(Extr-Par)
$$\frac{P \rightsquigarrow [\Gamma_P \;\|\; A_P] \qquad Q \rightsquigarrow [\Gamma_Q \;\|\; A_Q]}{P \mid Q \rightsquigarrow [\Gamma_P, \Gamma_Q \;\|\; A_P \mid A_Q]}$$

Table 4. The extraction relation $P \rightsquigarrow [\Gamma \;\|\; A]$

Typing rules for processes. Table 3 presents the typing rules for actions and processes: we only discuss the most interesting points. The side-condition to (Proc-Repl-In), requiring that the continuation typechecks in a copyable environment, is needed for subject reduction as replicated processes may spawn an unbounded number of copies of their continuation [22]. (Proc-Cond) keeps track of the equality between two terms in the successful branch of a conditional check: since this information can be used an arbitrary number of times, it is made exponential. The rules for cryptography (Proc-ADec), (Proc-Ver) and (Proc-SDec) mirror the idea of the corresponding rules for cryptographic packets in Table 2. (Proc-Extr) is the only rule for proper processes, which are typechecked by first extracting the top-level restrictions and assumptions into a typing environment, and then using that environment to typecheck the residual action process. Extracting the assumptions is needed to protect against using them more than once in the same type derivation; extracting the restrictions keeps all names in scope (cf. Table 4).

The type system is completed by a set of structural rules to enable weakening and contraction, as well as the manipulation of the logical connectives in typing derivations. Since proofs in sub-structural logics require careful management of the environment, like in [24] these rules are needed to improve the expressiveness of our framework.

Theorem 1 (Robust Safety). *Let P be a closed process such that $fn(P) = \{a_1, \ldots, a_n\}$ and let $a_1 : \mathit{Un}, \ldots, a_n : \mathit{Un} \vdash P$, then P is robustly safe.*

5 Case study: cryptographic sessions

We show the type system at work on two small, but realistic case studies that demonstrate the flexibility and effectiveness of our framework.

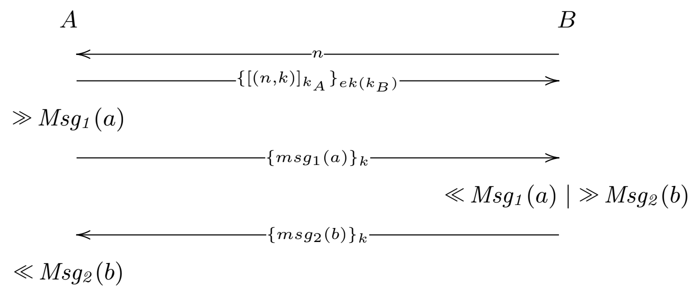
We start introducing additional notation for the system-controlled guards. First, we presuppose two system predicates $\textsc{key}(\cdot)$ and $\textsc{nonce}(\cdot)$, to serve as guards in the refinements associated with session keys, and (long-term) keys in nonce-based protocols, respectively. As discussed earlier, $\textsc{nonce}(\cdot)$ guards are used with keys such as $k : \eta Key(w : U, \{x : T \mid \textsc{nonce}(w) \multimap C(x)\})$ to exchange a nonce $n : U$, packaged with a payload $M : T$ such that $C(M)$. The underlying verification pattern presupposes that (at most one occurrence of) the formula $\textsc{nonce}(n)$ be available at the receiver to obtain a proof that the sender possesses the credential $C(M)$. The pattern for session keys has the same rationale. In that case, it is built around keys such as $k : \mathit{SymKey}_{(y)}(\{x : T \mid \textsc{key}(y) \multimap C(x)\})$ intended for the exchange of payloads $M : T$ such that $C(M)$, with $\textsc{key}(k)$ acting as the controlling guard, predicating on the key k itself through the binder y.

Both patterns can be generalized to enable multiple checks of the same nonce and multiple uses of the same key within a session. That is achieved with key types of the form $\eta\mathit{Key}_{(y)}(\sum_{i=1}^{\ell}(w : U, \{x : T_i \mid \textsc{nonce}(w, M_i) \multimap C_i(x)\})$ and similarly $\mathit{SymKey}_{(y)}(\sum_{i=1}^{\ell}\{x : T_i \mid \textsc{key}(y, M_i) \multimap C_i(x)\})$, used in conjunction with assumptions of the form $\textsc{nonce}(n, M_i)$ and $\textsc{key}(k, M_i)$, respectively. The M_i's are closed, pairwise distinct terms that serve as tags to mark the ℓ different program points where the same nonce may be checked (the same key used)[5]. The following notation helps structure our specification patterns:

$$\begin{aligned}
&\textsc{def}\ k = \textsc{SessionKey}[\textstyle\sum_{i=1}^{\ell}(M_i, T_i, C_i(x))]\ \textsc{in}\ P \triangleq \\
&\qquad\qquad new(k : \mathit{SymKey}_{(y)}(\textstyle\sum_{i=1}^{\ell}\{x : T_i \mid \textsc{key}(y, M_i) \multimap C_i(x)\})). \\
&\qquad\qquad (\gg \textsc{key}(k, M_1) \mid \cdots \mid \gg \textsc{key}(k, M_\ell) \mid P)
\end{aligned}$$

$$\begin{aligned}
&\textsc{def}\ n = \textsc{nonce}[U, \textstyle\sum_{i=1}^{\ell} M_i]\ \textsc{in}\ P \triangleq\ new(n : U). \\
&\qquad\qquad\qquad (\gg \textsc{nonce}(n, M_1) \mid \ldots \mid \gg \textsc{nonce}(n, M_\ell) \mid P)
\end{aligned}$$

Bounded sessions. Our first example is a protocol that implements a bounded session, built around a finite, and fixed, flow of messages. It involves two agents that perform a nonce-handshake to exchange a symmetric key k and then use the key in a session that exchanges two messages, as shown in the diagram below:



There is no global policy defined here, and the assumptions and assertions of the specification are only meant to track the session steps, ensuring the timeliness of the messages exchanged. The interesting part of the encoding in our applied pi-calculus is in the choice of key types. We start with the type T_k of the shared session key k. Assuming a and b may be given type Un, we define:

$$T_k = \mathit{SymKey}_{(y)}(\textstyle\sum_{i=1}^{2}\{x : \mathit{Un} \mid \textsc{key}(y, i) \multimap \mathit{Msg}_i(x)\})$$

[5] When $\ell \leq 1$, and we need no tag, we simply omit them from the notation.

T_k presupposes two uses of k, to extract the two different types of messages conveying the affine formulas $Msg_1(a)$ and $Msg_2(b)$. In the applied pi-calculus specification below, this type is introduced together with the two assumptions for the guard predicates $\textsc{key}(k,1)$ and $\textsc{key}(k,2)$ (we use natural numbers as the closed terms serving as tags). Based on T_k we may then construct the type T_{k_A} of A's signing key: $T_{k_A} = \mathit{SigKey}(y : \mathit{Un}, \{x : T_k \mid \textsc{nonce}(y) \multimap \textsc{key}(x,1)\})$. Notice that the credential protected by the nonce is the system guard $\textsc{key}(k,1)$ that will allow B to use the shared k and extract the credential $Msg_1(a)$ marking the completion of the first exchange. The other guard, $\textsc{key}(k,2)$ remains with A itself, to enable A's own use of the key at the completion of the protocol.

We are ready to define the protocol code: to ease the notation, we coalesce subsequent destructor applications in a single let statement:

$$
\begin{array}{rl}
A \triangleq & in(net, x_n). \\
& \textsc{def}\ k = \textsc{SessionKey}[(1, \mathit{Un}, Msg_1(x)) + (2, \mathit{Un}, Msg_2(x))]\ \textsc{in} \\
& out(net, enc(sign((x_n, k), k_A), ek(k_B))). \\
& \gg Msg_1(a) \\
& \mid out(net, senc(inl(a), k)). \\
& \quad in(net, x).let\ y = caser(sdec(x, k))\ in \ll Msg_2(y)
\end{array}
$$

$$
\begin{array}{rl}
B \triangleq & \textsc{def}\ n = \textsc{nonce}[\mathit{Un}]\ \textsc{in}\ out(net, n). \\
& in(net, x). \\
& let\ (y_n, y_k) = ver(dec(x, k_B), vk(k_A))\ in \\
& if\ (y_n = n)\ then \\
& in(net, z). \\
& let\ w = casel(sdec(z, y_k))\ in \ll Msg_1(w) \\
& \mid \gg Msg_2(b) \mid out(net, senc(inr(b), y_k))
\end{array}
$$

Typechecking the code goes as follows: we only comment on the most important steps, looking at the code of A and B separately.

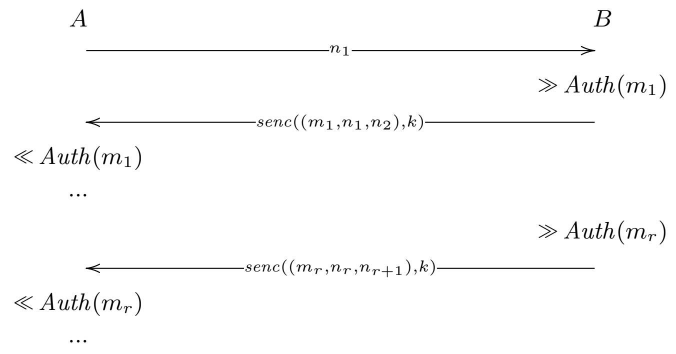
At A's side, introducing the shared key k extends the typing environment with the assumptions $\textsc{key}(k,1)$ and $\textsc{key}(k,2)$. Then, to sign k with k_A, one derives $(x_n, k) : (y{:}\mathit{Un}, \{x{:}T_k \mid \textsc{nonce}(y) \multimap \textsc{key}(x,1)\})$ by (Term-Pair), (Term-Refine), and a proof of $\textsc{key}(k,1), \textsc{key}(k,2) \vdash \textsc{nonce}(x_n) \multimap \textsc{key}(k,1)$. Similarly, to send the first message encrypted under k, an application of (Term-SEnc) requires one to show $a : \{x : \mathit{Un} \mid \textsc{key}(k,1) \multimap Msg_1(x)\}$, which in turn derives by (Term-Refine) based on a proof of $Msg_1(a) \vdash \textsc{key}(k,1) \multimap Msg_1(a)$.

At B's side, creating the fresh nonce n extends the environment with the guard $\textsc{nonce}(n)$. When B gets a response, she decrypts it and verifies the signature to extract the pair (y_n, y_k) and the formula $\textsc{nonce}(y_n) \multimap \textsc{key}(y_k,1)$ by (Proc-Ver). Then, checking y_n against n extends the typing environment with the equality $!(y_2 = n)$, and the analysis of B proceeds breaking the implication as follows:

$$
\frac{\cdots, \textsc{nonce}(n), !(y_n = n) \vdash \textsc{nonce}(y_n) \qquad \cdots, \textsc{key}(y_k,1) \vdash in(net, z).\cdots}{\cdots, \textsc{nonce}(n), !(y_n = n), \textsc{nonce}(y_n) \multimap \textsc{key}(y_k,1) \vdash in(net, z).\cdots}\ (\multimap\text{-Left})
$$

The left premise is derived directly in the affine logical system. The right premise, in turn, leaves the guard $\textsc{key}(y_k,1)$ for B to use y_k (the session key) in the continuation. Indeed, when the packet z reaches B, it is decrypted as w using key y_k (and the sum left destructor). By applying (PROC-SDEC) (and subsequently (PROC-CASE-LEFT)), the environment is extended with the information $w : Un, \textsc{key}(y_k, 1) \multimap Msg_1(w)$. Note that the actual parameter y_k in the code replaces the formal parameter y in type T_k upon decryption, as dictated by (PROC-SDEC). Now, consuming the guard $\textsc{key}(y_k, 1)$, we may derive the assertion $Msg_1(w)$ as desired.

Unbounded sessions. Though effective, the use of session keys illustrated in the previous example only applies to sessions in which the key is used a predefined (and finite) number of times. The next protocol, proposed in [18], shows how to account for unbounded sessions, exchanging an arbitrarily long stream of timely messages.



Unlike the previous protocol, here the message flow is always in the same direction, from B to A, and the message exchanged at step i conveys a payload m_i, which is authenticated by consuming nonce n_i, and a fresh nonce n_{i+1}, which is used to authenticate the next exchange. Again, the assumptions and assertions of the specification only serve for verifying the timeliness of each exchange. In this case, the timeliness proof relies on the following type for the shared key:

$$k : SymKey_{(y)}(x_1 : Un, (x_2 : Un, \{x_3 : Un \mid \textsc{nonce}(x_2) \multimap (Auth(x_1) \otimes \textsc{nonce}(x_3))\})).$$

At each use of k for decryption, A consumes the guard on the current nonce to obtain a proof of the expected authorization credential, and the nonce to repeat the process at the subsequent iteration.

The applied pi-calculus code for the protocol is as follows.

$$\begin{array}{ll}
A \ = new\ a : Ch(\{x : Un \mid \textsc{nonce}(x)\}). & B \ = new\ b : Ch(Un). \\
\quad\ \ \textsc{def}\ n = \textsc{nonce}[Un]\ \textsc{in} & \quad\ \ in(net, x).(out(b, x) \mid B^*) \\
\quad\ \ out(net, n).(out(a, n) \mid A^*) & B^* = *in(b, x).new\ m : Un. \\
A^* = *in(a, x).in(net, y). & \quad\ \ \gg Auth(m) \\
\quad\ \ \ let\ (z_1, z_2, z_3) = sdec(y, k)\ in & \quad\ \ \mid \textsc{def}\ n = \textsc{nonce}[Un]\ \textsc{in} \\
\quad\ \ \ if\ (z_2 = x)\ then & \quad\ \ \ out(net, senc((m, x, n), k)). \\
\quad\ \ \ \ll Auth(z_1) \mid out(a, z_3) & \quad\ \ \ out(b, n)
\end{array}$$

Both agents include replicated sub-processes whose iterations are controlled via synchronization over private channels. While this is a standard practice in the pi-calculus, a remark is in order on the type chosen for channel a: this type is needed to provide nonce capabilities to the replicated process A^*, since rule (Proc-Repl-In) requires to typecheck this process in a copyable environment. The guard formula $\textsc{nonce}(n)$ is used to typecheck the output of n on a, so that the associated capability can be recovered upon an input from the channel.

6 Conclusion

Authentication and authorization have been studied extensively in the literature on protocol verification, yet mostly as independent problems. We have proposed a unifying technique based on a novel affine refinement type system. The approach appears promising, as it supports a modular design of the framework, and is effective in the analysis of interesting classes of authentication protocols and authorization systems.

We are currently investigating the applications of our technique to further authentication mechanisms commonly employed in practice, e.g., timestamps and session identifiers, and to further protocols where the same nonce is checked by different principals. From our initial results, our approach appears to generalize smoothly to all such cases. We are also porting our framework from the applied pi-calculus to RCF [6], a concurrent functional programming language strongly related to F#. By recasting our technique to this new setting, we will be able to conduct our type-based analysis directly on application code.

Acknowledgments. Work partially supported by MIUR Project IPODS "*Interacting Processes in Open-ended Distributed Systems*".

References

1. Abadi, M.: Secrecy by typing in security protocols. Journal of the ACM 46(5), 749–786 (1999)
2. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proc. 28th Symposium on Principles of Programming Languages (POPL). pp. 104–115. ACM Press (2001)
3. Abadi, M.: Logic in access control. In: Proc. 18th Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 228–233. IEEE Computer Society Press (2003)
4. Backes, M., Hriţcu, C., Maffei, M.: Type-checking zero-knowledge. In: 15th ACM Conference on Computer and Communications Security (CCS 2008). pp. 357–370. ACM Press (2008)
5. Bauer, L., Jia, L., Sharma, D.: Constraining credential usage in logic-based access control. In: Proc. 23rd IEEE Symposium on Computer Security Foundations (CSF). pp. 154–168. IEEE Computer Society Press (2010)
6. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: Proc. 21th IEEE Symposium on Computer Security Foundations (CSF). pp. 17–32. IEEE Computer Society Press (2008)

7. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular verification of security protocol code by typing. In: Proc. 37th Symposium on Principles of Programming Languages (POPL). pp. 445–456. ACM (2010)
8. Blanchet, B.: From Secrecy to Authenticity in Security Protocols. In: Hermenegildo, M., Puebla, G. (eds.) 9th International Static Analysis Symposium (SAS'02). Lecture Notes on Computer Science, vol. 2477, pp. 342–359. Springer Verlag, Madrid, Spain (Sep 2002)
9. Bugliesi, M., Focardi, R., Maffei, M.: Compositional analysis of authentication protocols. In: Proc. 13th European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 2986, pp. 140–154. Springer-Verlag (2004)
10. Bugliesi, M., Focardi, R., Maffei, M.: Analysis of typed-based analyses of authentication protocols. In: Proc. 18th IEEE Computer Security Foundations Workshop (CSFW). pp. 112–125. IEEE Computer Society Press (2005)
11. Bugliesi, M., Focardi, R., Maffei, M.: Dynamic types for authentication. Journal of Computer Security 15(6), 563–617 (2007)
12. Bugliesi, M., Calzavara, S., Eigner, F., Maffei, M.: Resource-aware authorization policies for statically typed cryptographic protocols. In: Proc. 24th IEEE Computer Security Foundations Symposium (CSF). pp. 83–98 (2011)
13. Focardi, R., Maffei, M.: Types for security protocols. In: Formal Models and Techniques for Analyzing Security Protocols, Cryptology and Information Security Series, vol. 5, chap. 7, pp. 143–181. IOS Press (2011)
14. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization policies. In: Proc. 14th European Symposium on Programming (ESOP). pp. 141–156. Lecture Notes in Computer Science, Springer-Verlag (2005)
15. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization in distributed systems. In: Proc. 20th IEEE Symposium on Computer Security Foundations (CSF). pp. 31–45. IEEE Computer Society Press (2007)
16. Freeman, T., Pfenning, F.: Refinement types for ml. In: Wise, D.S. (ed.) PLDI. pp. 268–277. ACM (1991)
17. Girard, J.Y.: Linear logic: its syntax and semantics. In: Advances in Linear Logic. London Mathematical Society Lecture Note Series, vol. 22, pp. 3–42 (1995)
18. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. Journal of Computer Security 11(4), 451–519 (2003)
19. Gordon, A.D., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. Journal of Computer Security 12(3), 435–484 (2004)
20. Haack, C., Jeffrey, A.: Timed spi-calculus with types for secrecy and authenticity. In: Proc. 16th International Conference on Concurrency Theory (CONCUR). vol. 3653, pp. 202–216. Springer-Verlag (2005)
21. Haack, C., Jeffrey, A.: Pattern-matching spi-calculus. Information and Computation 204(8), 1195–1263 (2006)
22. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Transactions on Programming Languages and Systems 21(5), 914–947 (1999)
23. Lowe, G.: A Hierarchy of Authentication Specifications. In: Proc. 10th IEEE Computer Security Foundations Workshop (CSFW). pp. 31–44. IEEE Computer Society Press (1997)
24. Mandelbaum, Y., Walker, D., Harper, R.: An effective theory of type refinements. In: Proc. of the 8th ACM SIGPLAN international conference on Functional programming (ICFP). pp. 213–225. ACM Press (2003)
25. Troelstra, A.S.: Lectures on linear logic. CSLI Stanford, Lecture Notes Series nr. 29 (1992)