Resource-aware Authorization Policies for Statically Typed Cryptographic Protocols

Michele Bugliesi Stefano Calzavara
Università Ca' Foscari Venezia
{bugliesi,calzavara}@dais.unive.it

Fabienne Eigner Matteo Maffei
Saarland University
{eigner,maffei}@cs.uni-saarland.de

***Abstract*—Type systems for authorization are a popular device for the specification and verification of security properties in cryptographic applications. Though promising, existing frameworks exhibit limited expressive power, as the underlying specification languages fail to account for powerful notions of authorization based on access counts, usage bounds, and mechanisms of resource consumption, which instead characterize most of the modern online services and applications.**

We present a new type system that features a novel combination of affine logic, refinement types, and types for cryptography, to support the verification of resource-aware security policies. The type system allows us to analyze a number of cryptographic protocol patterns and security properties, which are out of reach for existing verification frameworks based on static analysis.

I. Introduction

Authorization policies provide a well-established device for security analysis and specification of distributed protocols and applications. Given an access request to a sensitive resource in a system, an authorization policy determines whether the request should be allowed [1], [2]. Authorization policies are often expressed in code by means of annotations: *assumptions* introduce new hypotheses – formulas that are assumed to hold – and *assertions* declare formulas that are expected to logically follow from the previously introduced hypotheses [3]. Then, to prove that a protocol complies with a given authorization policy, one must prove that at run-time the assumptions are guaranteed to entail all the assertions. To illustrate, consider the policy defined by the formula:

$$\forall c, s, r.\ \mathsf{Req}(c, s, r) \Rightarrow \mathsf{Grant}(c, s, r) \qquad (1)$$

and the annotated code for the one-step protocol below:

$$A \triangleq \mathsf{assume}\ \mathsf{Req}(A, B, res)$$
$$\mathsf{out}(net, \mathsf{sign}((A, B, res), \mathsf{sk}(k_A)))$$

$$B \triangleq \mathsf{in}(net, y).\mathsf{let}\ (A, B, x) = \mathsf{ver}(y, \mathsf{vk}(k_A))\ \mathsf{in}$$
$$\mathsf{assert}\ \mathsf{Grant}(A, B, x)$$

The policy states that a server s may grant access to a resource r to any client c that makes a corresponding request. The protocol, in turn, is so defined as to allow client A to authenticate her request for resource res to server B. Before signing the message and sending it over *net*, A *assumes* $\mathsf{Req}(A, B, res)$. B, in turn, *asserts* $\mathsf{Grant}(A, B, x)$ to acknowledge the receipt of a signed request on x from A and grant access. Proving this protocol safe amounts to showing that, for all runs, the assertion $\mathsf{Grant}(A, B, res)$ made by B is entailed by a corresponding assumption $\mathsf{Req}(A, B, res)$ made by A.

Security proofs for annotated protocol code such as the one above can be carried out very effectively by static type systems: a number of papers have shown that strong security guarantees may be achieved by automated analysis at the expense of a modest effort required for annotating the code [4], [5] and that type systems outperform other analysis techniques in the verification of protocol implementations [3], [6].

One weakness of existing type systems, that still hinders their wide applicability, lies in the limited expressive power of their underlying policy languages. Indeed, except for a few noticeable exceptions [7], [8], current type systems for authorization draw on classical or intuitionistic logics, and hence fall short of capturing the *resource-conscious* nature of the authorization policies required in most practical scenarios and applications. Our simple specification above already shows the problem: if we interpret our formula (1) above intuitionistically, we realize that one request for a given resource is enough to justify unboundedly many permissions to access that resource. This is clearly problematic in various situations: to just name two, in e-banking protocols, where the *number* of transactions may not be overlooked, and in e-voting protocols, which rely critically on the property that the vote cast by each voter should be counted only once.

A. Resource-conscious policies

Resource-aware authorization policies are naturally accounted for by relying on *substructural logics* such as *linear logic* [11] or *affine logic* [10]. For instance, the desired correspondence between requests and permissions may be accommodated in our policy above by construing assumptions as linear formulas, and stating the formula in terms of linear implication ($\multimap$) in place of the original intuitionistic implication ($\Rightarrow$):

$$\forall c, s, r.\ \mathsf{Req}(c, s, r) \multimap \mathsf{Grant}(c, s, r)$$

Unlike classical and intuitionistic logic, which deal with stable truths (if A and $A \Rightarrow B$, then B, but A *still holds* [11]) linear/affine logic emphasizes the role of formulas as resources that are *consumed* along a proof, and of implication as a *reaction* that transforms formulas. Thus, in our example, each assertion $\mathsf{Grant}(A, B, res)$ by B now requires a corresponding (distinct) justifying assumption $\mathsf{Req}(A, B, res)$ by A, as expected. As it turns out, this resource interpretation of formulas perfectly fits the properties we often target in authorization policies. Policy statements like "registered users can download movies" provide rather poor means to govern streaming services. Instead, one would rather want to quantify the number of accesses to the service, and diversify it for the different classes of users. For instance, "users with a trial account may download at most one movie, while unboundedly many downloads are available to users with a premium account". Linear and affine logics offer direct support for expressing these policies, and as we show in this paper, our typed calculus provides an effective tool for their implementation in distributed settings, and their static verification.

B. Contributions

We present a type system for the verification of resource-aware security policies for distributed protocols, expressed in a variant of the applied π-calculus [12]. Our type system features a novel combination of affine logic, refinement types, and types for cryptography, which allows us to validate distributed resource-conscious properties that were out of reach for existing verification frameworks based on static analysis.

We identify and statically characterize several cryptographic patterns that enforce the freshness of the resources used in the authorization request, such as nonce handshakes and bounded usage of session keys. Interestingly, typing these patterns does not require ad-hoc mechanisms: instead, it relies on the resource interpretation of the refinement types associated with nonce and key types and thus fits naturally the general principles of logical entailment in the linear/affine logics underlying our type system.

While we are certainly not the first to propose linear and affine logics as formal devices for authorization policies, our system is unique in its use of affine formulas to refine the types of data exchanged over an insecure network. As it turns out, this is a rather challenging task: it requires strong protection against replay attacks on linearly refined data, which may easily defeat the purpose of any resource-conscious policy.

C. Advance over related work

Type systems supporting cryptography have been applied widely in the literature: to enforce authorization policies in distributed systems [4], guarantee run-time invariants in functional code [3], [6], model properties of zero-knowledge proofs [5] or secure multi-party computations [13]. In none of these papers, the underlying authorization logics are amenable to expressing resource-conscious policies, nor do they include mechanisms for injective agreement such as those described in early attempts [14], [15].

In a recent, still unpublished manuscript [8], Swamy et al. introduce F*, a core typed lambda calculus derived from Fine [7] and RCF [3] that supports programming with refinement types, security proofs, and erasure. F* combines RCF's refinements types for the static verification of functional implementations of cryptographic protocols, with Fine's provision for resource-aware authorization and information-flow policies in stateful programs [16]. In particular, F* does allow the use of digital signatures to witness the derivability of refinement formulas. However, it constrains the payload type of such signatures to include non-linear refinements. As a result, F* does not support the exchange of digitally signed linearly refined data, which is instead the distinguishing, and arguably the most challenging, feature of our system. Indeed, sending affine values over the network must be handled with care, due to potential replays by the attacker. This paper develops several patterns to support the conversion of affine values into a non-affine form suitable for transmission over the network. Recipients of non-affine messages may be able to convert these back to affine form upon verification of signatures using suitably typed affine verification keys. As such, our patterns resemble constructs proposed in the context of various programming languages, e.g., the adoption and focus of [17], which similarly permit temporary duplication/aliasing of affine values.

Linear and affine logics have been used by various authors to express properties of distributed systems. Among other approaches, two appear closer to ours. Cederquist et al. [18] introduce a framework for checking compliance against discretionary access control policies that capture properties based on resource consumption. However, their verification technique is dynamic and based on ex-post auditing, while our type system is static, and based on a textual inspection of the code that provides compile-time security guarantees.

Linear type systems have been proposed for a wide range of applications in programming languages, from garbage collection to encoding of side-effects, from compiler optimization to detection of programming mistakes [19], [21], [22]. The core ideas for the treatment of linear refinements have been explored in previous works, such as the general theory of type refinements for effectful programs presented by Mandelbaum et al. [23] and the work on modular typestate checking of object-oriented programs by Bierhoff et al. [24]. There are, however, significant differences from our approach, since we consider concurrency, cryptography, and active attackers operating on the network, which are the source of the main technical challenges we had to address in our work. Linear type systems have also been considered in concurrency theory [25], [26], [27], [28] but the focus there is on channel usage and resource consumption rather than on security, as these works consider neither cryptography nor attackers.

Bowers et al. [29] develop a mechanism to enforce the consumption of credentials and use it to provide a distributed implementation of the linear access-control logic by Garg et al. [30]. Our goal in the present paper is different, as we do not target the development of new distributed protocols; rather, we aim at widening the range of statically verifiable authorization systems. Indeed, we show that the protocol proposed in [29] can be validated by our type system.

By targeting the analysis of policies based on resource consumption, our type system is also related to work on injective agreement [31] and, in particular, to the type and effect systems for authenticity developed by Gordon and Jeffrey [14], [15]. These type systems use a kind of affine typing for nonces, where the creation of a nonce justifies one subsequent nonce check. Indeed, our type system constitutes a first step towards filling a long standing gap between statically-typed injective agreement properties [14], [15], [32], [33], [34], [35], [36] and statically-typed non-linear authorization policies [37], [5], [38], reconciling them within a unique framework where injective agreement is characterized as a mechanism of the same linear logic used to express authorization. This enables an interplay between the linear predicates (or formulas in general) that are statically transferred via a nonce handshake and the recipient's authorization policy: for instance, a linear predicate assumed on the sender's side may be used as an hypothesis of a linear implication on the recipient's side (we show an example of such an interplay in Section VI-D).

In this paper, we consider the fundamental paradigms to exchange linear information, namely, session keys and nonce handshakes. For simplicity, we focus on the nonce handshakes in which the response is authenticated, the so-called Public-Out-Secret-Home (POSH) handshakes, and we leave as a future work the Secret-Out-Public-Home (SOPH) and Secret-Out-Secret-Home (SOSH) patterns considered by the type and effect system of [15]. This choice is motivated by the fact that POSH handshakes are largely deployed in practice and best fit distributed authorization systems, in which authorization proofs are sent directly by one principal to another. Backes et al. have recently shown that other kinds of nonce handshakes (e.g., SOSH) can be enforced using union and intersection types [39]. Integrating these type systems is an interesting research direction.

In some scenarios, our approach turns out to be even more precise than existing frameworks for analysis of authentication based on theorem-proving. For instance, the state-of-the-art theorem-prover ProVerif [6] fails to validate the injective agreement between the $\mathsf{received}(m)$ and $\mathsf{sent}(m)$ events in the code below:

$$\begin{aligned}\mathsf{new}\ c.\ !\ (\ &\mathsf{new}\ m.\mathsf{event}\ \mathsf{sent}(m).\mathsf{out}(c,m)\\ &\mid \mathsf{in}(c,x).\mathsf{event}\ \mathsf{received}(x)\ )\end{aligned}$$

Here we have an unbounded number of copies of two processes that exchange a fresh value m over a private channel, not available to the attacker. Since each input consumes one output, the protocol does provide injective agreement. Yet, ProVerif fails to validate this property as it builds on an over-approximation where output messages can be received several times, no matter whether the channel is private or not.

Structure of the paper. Section II introduces affine authorization logics. Section III presents our dialect of the applied π-calculus. Section IV and Section V describe the type system, while Section VI shows it at work on various examples. Section VII concludes.

II. Resource Logics for Authorization

Our process calculus and type system rely on intuitionistic affine logic [10], and though they remain largely independent of the exact choice of the logic, we do make a number of assumptions on the set of connectives and proof-theoretic properties we expect. These assumptions are required to prove the soundness of the type system and suffice to characterize the examples considered in this paper. Considering additional connectives may be worthwhile for future work.

Like linear logic, affine logic builds on the view of formulas as transient resources rather than stable truths. This view arises as the result (i) of dispensing with the structural rule of contraction commonly found in classical and intuitionistic logic, and (ii) of the novel, multiplicative interpretation for linear connectives.

The consequences of this interpretation may be understood by looking at the proof-theoretic presentations of the left-introduction rule for linear implication ($\multimap$ LEFT) and the right-introduction rule for multiplicative conjunction ($\otimes$ RIGHT) in Definition II.1 below.

Definition II.1 (Authorization Logic) *An authorization logic* $(\mathcal{C}, !, \vdash)$ *is characterized by (i) the set of first-order formulas built over a set of connectives* $\mathcal{C} \supseteq \{\multimap, \otimes\}$*, (ii) an exponential modality* $!$*, and (iii) an entailment relation* $\Gamma \vdash A$ *between a multiset of formulas* Γ *and a formula* A*, that is closed under substitution, and includes a theory of equality and the following rules:*

$$\multimap\ \text{LEFT}\ \frac{\Gamma_1 \vdash A \qquad \Gamma_2, B \vdash C}{\Gamma_1, A \multimap B, \Gamma_2 \vdash C} \qquad \otimes\ \text{RIGHT}\ \frac{\Gamma_1 \vdash A \qquad \Gamma_2 \vdash B}{\Gamma_1, \Gamma_2 \vdash A \otimes B}$$

$$!\ \text{LEFT}\ \frac{\Gamma, !A, A \vdash B}{\Gamma, !A \vdash B} \qquad !\ \text{RIGHT}\ \frac{!\Gamma \vdash A}{!\Gamma \vdash !A} \qquad !\ \text{WEAKENING}\ \frac{\Gamma \vdash B}{\Gamma, !A \vdash B}$$

$$\text{IDENT}\ \frac{}{A \vdash A} \qquad \text{WEAKENING}\ \frac{\Gamma \vdash B}{\Gamma, A \vdash B} \qquad \text{CUT}\ \frac{\Gamma_1 \vdash A \qquad \Gamma_2, A \vdash B}{\Gamma_1, \Gamma_2 \vdash B}$$

Notice how, in both cases, the formulas on the left hand-side of the lower sequent get split in the hypotheses, so that each formula may be used just once in a proof. Indeed, in linear logic, each formula must be used *exactly* once. Affine logic relaxes this constraint by introducing the rule of weakening (cf. Definition II.1) which allows certain formulas to be disregarded along a proof. Thus, for instance, one has $A, B \vdash B$ in affine logic but not in linear logic. Besides simplifying the technical development, the choice of affine logic over linear logic has strong practical motivations: affine logic retains the strong control over resources distinctive of linear logic, but at the same time leaves the degree of flexibility that appears necessary for the kind of specifications we target. For example, in electronic voting protocols, we might wish to express that a voter can only vote once, but, of course, she is not forced to participate in the election. Finally, the modality "!" qualifies formulas which can be liberally duplicated: we refer to these as *exponential* formulas.

Except for weakening, which is distinctive of affine logic, our definition of authorization logic forms a standard fragment of any proof-theoretic presentation of intuitionistic linear logic. Interestingly, one can show that our logic can be embedded into richer logical fragments which enjoy efficient proof search strategies. An example of such fragments may be readily obtained by means of an affine extension of the linear logic proposed by Garg et al. [30] or of the programming language *Lolli* [40], [41], which admits a *logically complete*, goal-directed proof strategy. Indeed, all the authorization policies illustrated in the paper are very naturally expressed as *Lolli* programs.

III. Applied Pi-calculus

This section introduces the syntax and the operational semantics of the dialect of the applied π-calculus we employ in the paper.

Table I Terms and Constructors $\quad (\widetilde{M} = M_1, \ldots, M_n)$

K, L, M, N ::=	terms
a, b, c, k, m, n	names
x, y, z, v, w	variables
$f(\widetilde{M})_T$	constructor application
f ::=	constructors
	$\mathsf{ek}^1, \mathsf{dk}^1, \mathsf{vk}^1, \mathsf{sk}^1, \mathsf{enc}^2, \mathsf{sign}^2, \mathsf{senc}^2, \mathsf{pair}^2$

The set of terms is the free algebra built from names, variables and constructor applications. We let u range over both names and variables. The set of constructors comprises the symbols ek, dk, sk, and vk, to form keys (for encryption, decryption, signature, and verification, respectively) from a seed k as in $\mathsf{ek}(k)$; sig, enc and senc to construct digital signatures $\mathsf{sig}(M, \mathsf{sk}(k))$, public-key encryptions $\mathsf{enc}(M, \mathsf{ek}(k))$, and symmetric encryptions $\mathsf{senc}(M, k)$; and a final constructor pair to construct pairs. Constructor applications carry an (optional) typing annotation in the form of a subscript. These annotations are convenient for the proofs, do not influence the semantics by any means, and will be omitted when uninteresting. For the sake readability, we let (M, N) stand for $\mathsf{pair}(M, N)$.

Table II Destructor Evaluation $\quad g(M_1, M_2) \Downarrow N$

g ::= $\mathsf{eq}^2, \mathsf{dec}^2, \mathsf{ver}^2, \mathsf{sdec}^2, \mathsf{check}^2$ destructors

$\mathsf{eq}(M, M)$	$\Downarrow$	M
$\mathsf{dec}(\mathsf{enc}(M, \mathsf{ek}(K)), \mathsf{dk}(K))$	$\Downarrow$	M
$\mathsf{ver}(\mathsf{sign}(M, \mathsf{sk}(K)), \mathsf{vk}(K))$	$\Downarrow$	M
$\mathsf{sdec}(\mathsf{senc}(M, K), K)$	$\Downarrow$	M
$\mathsf{check}(\mathsf{fresh}(M, N), N)$	$\Downarrow$	M

Destructors are partial functions to decompose terms. eq performs equality tests, dec decrypts asymmetric encryptions, ver verifies signatures, sdec decrypts symmetric encryptions, and check checks the freshness of nonces. Applying a destructor g to terms M_1, M_2 either succeeds and yields a term N, noted $g(M_1, M_2) \Downarrow N$, or fails, noted $g(M_1, M_2) \not\Downarrow$. The semantics of these destructors is mostly standard, or self-explained by looking at the definition.

Table III Processes

$P, Q, R ::=$	processes
$\mathsf{assume}\ F$	assumption
$\mathsf{assert}\ F$	assertion
$\mathbf{0}$	null
$\mathsf{new}\ a : T.P$	restriction
$\mathsf{in}(M, x : T).P$	input of x from M
$!\mathsf{in}(M, x : T).P$	repl. input
$\mathsf{out}(M, N).P$	output of N on M
$P \mid Q$	parallel composition
$\mathsf{let}\ x : T = g(\widetilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q$	term destruction
$\mathsf{let}\ (x : T, y : U) = M\ \mathsf{in}\ P$	pair split
$\mathsf{let}\ x : T = f(\widetilde{M})\ \mathsf{in}\ P$	term construction

The structure of *processes* is reported in Table III. The syntax is explicitly typed, only to ease type checking, as the typing annotations do not affect the semantics of processes. We often omit the annotations when they are clear from the context or unimportant. The processes $\mathsf{assume}\ F$ and $\mathsf{assert}\ F$ are inert annotations that express security policies and help formalize the definition of safety (cf. Section III-A). Here, and throughout, we use $A, \dots, F$ to range over possibly exponential formulas, with the understanding that the exponential modality may only occur at the top level of a formula (i.e., not below a connective). The process forms for the null process, restriction, (replicated) input, output, and parallel composition are entirely standard. We introduce an explicit form for term construction and distinguish pair splitting from the remaining destructor forms for typing purposes only. For pair splitting we assume an implicit else-branch $\mathbf{0}$. We could easily extend this to arbitrary else-branches. We say that a process P is *static* if it does not contain any annotated constructor application. The scope of names and variables is delimited by restrictions, inputs, and lets. The notions of free variables $\mathit{fv}(P)$ and names $\mathit{fn}(P)$ arise as expected and we use $\mathit{fnfv}(P)$ to denote their union. A term is *ground* if it does not contain any variables. A process is *closed* if it does not have any free variables.

The semantics of the calculus is formalized in terms of structural congruence and reduction. The former allows for a syntactic rearrangement of processes, the latter rules process synchronizations and let evaluations. These rules are standard and omitted here.

A. Safety

Following [4], we decorate security-related protocol points with assumptions and assertions. Intuitively, the former introduce new hypotheses, while the latter declare formulas that should logically follow from the previously introduced hypotheses. Assumptions and assertions do not have any computational significance and are solely used to express security requirements. Intuitively, a process is safe if and only if, in all executions, the *multiplicative conjunction* of the active assertions is entailed by the active assumptions. This usage of assumptions and assertions to define the safety property resembles previous definitions of authenticity based on begin- and end-events [14], [15].

Definition III.1 (Safety) *A closed process P is* safe *iff $P \to^* \mathsf{new}\ \widetilde{a} : \widetilde{T}.(\mathsf{assert}\ C_1 | \dots | \mathsf{assert}\ C_n | Q)$ implies $Q \equiv \mathsf{new}\ \widetilde{b} : \widetilde{U}.(\mathsf{assume}\ F_1 | \dots | \mathsf{assume}\ F_m | Q')$, where $\{\widetilde{b}\} \cap (\mathit{fn}(C_1) \cup \dots \cup \mathit{fn}(C_n)) = \emptyset$, Q' has no top-level assertions, and $F_1, \dots, F_m \vdash C_1 \otimes \dots \otimes C_n$.*

As usual, we are interested in a stronger notion of safety, namely safety in the presence of an active opponent. Intuitively, the opponent is an arbitrary process that does not contain assertions (as the opponent could otherwise trivially break the safety property). Without loss of generality, we require opponents to be static processes: this restriction is technically convenient, and harmless, as the opponent can bind arbitrary terms to variables with term constructions. Similarly, we rule out the usage of check in the opponent code, since this destructor can be encoded by pair splitting and equality check.

Definition III.2 (Opponent) *A static, closed process is an* opponent *if it does not contain any assertions,* check *destructors, and the only type occurring therein is* Un.

Definition III.3 (Robust Safety) *A closed process P is* robustly safe *if and only if $P \mid O$ is safe for every opponent O.*

IV. Overview of the Type System

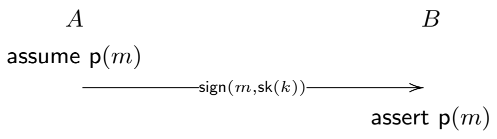
Existing type systems for authorization in cryptographic protocols rely on a few well-established patterns, based on the so-called *refinement types* [4].

A. Refinement type systems for authorization

Refinement types are dependent types of the form $\{x : T \mid C\}$ that *refine* the structural information T with the property encoded by the formula C. We illustrate the main idea underlying refinement type systems with a simplified version of the protocol discussed in Section I.

$$\begin{array}{rcl} A & \triangleq & \mathsf{new}\ m{:}\mathsf{Un}.\mathsf{assume}\ \mathsf{p}(m) \mid \mathsf{out}(c, \mathsf{sign}(m, \mathsf{sk}(k))) \\ B & \triangleq & \mathsf{in}(c, x).\mathsf{let}\ y = \mathsf{ver}(x, \mathsf{vk}(k))\ \mathsf{in}\ \mathsf{assert}\ p(y) \end{array}$$

A protocol run may be represented as follows:



Notice that in the process specification, the assertion is made by B in terms of the variable y, not of the message m as in the protocol run. The task of the type system is precisely to make sure that at run-time y will be bound to m, leading to the expected final assertion $\mathsf{p}(m)$. To capture this dynamic behavior statically, the type system relies on the type of keys (more generally, of communication links) to enforce the invariants on the payload exchanged needed to establish the desired connections between the two end-points. In the protocol in question, we may assume that $\mathsf{sk}(k) : \mathsf{SigKey}(T)$ and $\mathsf{vk}(k) : \mathsf{VerKey}(T)$, where the payload type T is the refinement type $\{x : \mathsf{Un} \mid \mathsf{p}(x)\}$. The type Un describes untrusted messages, i.e., message that may come from or be sent to the attacker. Under these typings, at A's end-point, the assumption $\mathsf{p}(m)$ together with the binding $m : \mathsf{Un}$ justifies the typing $m : \{x : \mathsf{Un} \mid \mathsf{p}(x)\}$, hence the use of $\mathsf{sk}(k)$ to sign m. At the other end-point, B may legally assume $y : \{x : \mathsf{Un} \mid \mathsf{p}(x)\}$ as a result of a successful signature verification, and use this assumption to justify the assertion $\mathsf{p}(y)$.

B. *Consumable refinements*

The typing we have just illustrated works well as long as the refinements of the data exchanged in a protocol encode stable truths. It fails, instead, when those refinements are meant to represent consumable resources. To see the problem, assume $\mathsf{p}(m)$ grants permissions that should not be iterated for free, and consider multiple copies of B running in parallel with A and an opponent. Since the opponent can duplicate the messages in transit on the network, B will end up verifying the same signature several times and asserting $\mathsf{p}(m)$ multiple times, thus breaking our safety property.

To fix this problem, we must gain stronger control over the exchange of payloads whose refinement is meant to represent *resources* to be consumed. This is easily achieved for exchanges that occur over protected channels. For communications over insecure links, instead, the problem is harder and the desired guarantees may hardly be enforced directly at the source of the exchange, because the opponent may *always* duplicate any payload sent over the network. Our type system provides two solutions, detailed below. The first solution is based on a mechanism that constrains the usage of decryption and verification keys at the target end of an exchange. A second solution exploits affine refinements to devise a novel typing discipline for nonce handshakes. The combination of the two mechanisms provides the type system with a high degree of expressive power, as illustrated by the typing examples of Section VI.

C. *Affine typing of decryption / verification keys*

We stipulate the following terminology: a refinement type is *exponential* if the formula it conveys can be asserted arbitrarily often; dually, a refinement type is *affine* if its refinement can be asserted at most once. We have a corresponding classification for the types of keys: a key-type is exponential if keys with that type can be used arbitrarily often; it is affine if keys of that type can be used at most once.

Signing keys are an example of exponential key-type, regardless of their payload type: in fact, even if we use these keys only once, the resulting signatures can still be duplicated by the opponent with an effect that is equivalent to using these keys multiple times. As a consequence, we may safely assume that types of the form $\mathsf{SigKey}(\{x : \mathsf{Un} \mid \mathsf{p}(x)\})$ are exponential.

For the types of verification keys, instead, the situation is different, as enforcing an affine usage of verification keys with affine payload type provides effective control over the payload type. Given that verification keys are public, however, we cannot constrain them to occur only once in a process, because they can be read and duplicated by the opponent. Instead, we can require a verification key $\mathsf{vk}(k)$ to be used at most once with the affine type $\mathsf{VerKey}(\{x : \mathsf{Un} \mid \mathsf{p}(x)\})$ and arbitrarily often with the weaker, exponential type $\mathsf{VerKey}(\mathsf{Un})$. As a consequence, $\mathsf{vk}(k)$ can be used at most once to justify the predicate $\mathsf{p}(x)$ but can otherwise occur arbitrarily often in the process.

D. *Affine typing of nonce handshakes*

Having affine keys as the only way to transfer affine information over the network would clearly be overly restrictive: first, our type system would only capture protocols based on session keys, thus failing to address all protocols based on long-term keys; second, we would not know how to distribute the keys initially, since we would need other affine keys to transfer them. We therefore provide a further typing mechanism for transferring affine information over the network based on nonce handshakes. The idea is well-known, namely:

keys can be used several times, as long as the receiver is able to check the freshness of the payload using a nonce. The following nonce handshake protocol is standard:

$$\begin{tikzcd}
A & & B \\
{} & & {} \arrow[ll, "n"] \\
\text{assume } \mathsf{p}(m) & & \\
{} \arrow[rr, "{\mathsf{sign}((m,n),\mathsf{sk}(k))}"] & & {} \\
& & \text{assert } \mathsf{p}(m)
\end{tikzcd}$$

The novelty is in the way we type-check the protocol, based on a rather natural use of affine refinements, which makes it just a special case of the general typing principles of resource-conscious authorization. The technical development may be summarized as follows. First, we introduce an *affine* type Nonce to enforce the freshness of nonces: n may be used at most once with type Nonce (to derive the affine information $\mathsf{p}(m)$, as detailed below) and arbitrarily often with the weaker type Un. The term (m, n) is given type $\mathsf{Fresh}(\{x : \mathsf{Un} \mid \mathsf{p}(x)\}, \mathsf{Un})$. Unlike key types, $\mathsf{Fresh}(\cdot, \cdot)$ types are always *exponential*, irrespective of the type of the payload, i.e., fresh packets containing affine information are still exponential. Hence, by our discussion on the typing of keys, this implies that also the verification key used to extract fresh packets is considered to be exponential, thus it can be used multiple times and sent over the network using standard techniques. In order to justify the typing $m : \{x : \mathsf{Un} \mid \mathsf{p}(x)\}$, which entails the affine information $p(m)$, B has to verify that n is fresh, which is achieved by pattern matching the pair (m, n) against the nonce n at type Nonce. Since Nonce is affine, the affine information may be derived only once.

V. Types and Typing Rules

The type system comprises several typing judgments: well-formed environments $\Gamma \vdash \diamond$, logical entailment $\Gamma \vdash C$, subtyping $\Gamma \vdash T <: U$, typing of terms $\Gamma \vdash M : T$, and typing of processes $\Gamma \vdash P$. Typing environments, denoted by Γ, track assumed formulas and map names and variables to types. Formally, Γ is a list of formulas and bindings of the form $u : T$.

A. Types, environments and logical entailment

The structure of types is defined in Table IV. Their intuitive meaning should be clear by the brief description provided there. The notion of well-formed environments is standard, and simply amounts to verifying that the free names and free variables of the formulas and types of Γ occur in the domain of Γ. As to logical entailment, we say that a typing environment Γ logically entails the formula C, written $\Gamma \vdash C$, if the formulas that can be extracted from Γ logically entail C: these formulas comprise the formulas in Γ and the formulas occurring in the refinement types in Γ. Thus, for instance: $a : \mathsf{Un}, \mathsf{Registered}(a), b : \{x : T \mid \mathsf{CanRead}(a, x)\} \vdash \mathsf{Registered}(a) \otimes \mathsf{CanRead}(a, b)$. The logical entailment relation and the function to extract formulas from a typing environment are defined in Table V.

Table IV Types

$T, U, V ::=$	
Un	untrusted
$\mathsf{Private}$	private
$\mathsf{Ch}(T)$	channel with T payload
$\{x : T \mid C\}$	refinement type (x bound in C)
$\mathsf{Pair}(x : T, U)$	dependent pair type (x bound in U)
$\mathsf{SigKey}(T)$	signing key for T payload
$\mathsf{VerKey}(T)$	verification key for T payload
$\mathsf{Signed}(T)$	signature with T payload
$\mathsf{SigSeed}(T; U)$	key-pair for digital signatures
$\mathsf{DecKey}(T)$	decryption key for T payload
$\mathsf{EncKey}(T)$	encryption key for T payload
$\mathsf{PubEnc}(T)$	encryption of T payload
$\mathsf{EncSeed}(T; U)$	key-pair for public-key cryptography
$\mathsf{SymKey}(T; U)$	symmetric key
$\mathsf{SymEnc}(T)$	symmetric key enc. of msg of type T
Nonce	nonce
$\mathsf{Fresh}(T, U)$	T payload with nonce of type U

Table V Logical Entailment $\Gamma \vdash C$

Entailed
$$\frac{\Gamma \vdash \diamond \qquad \mathit{fnfv}(C) \subseteq \mathit{dom}(\Gamma) \qquad \mathit{forms}(\Gamma) \vdash C}{\Gamma \vdash C}$$

Definition:
$\mathit{forms}(u : \{x : T \mid C\}) = \mathit{forms}(u : T), C\{u/x\}$
$\mathit{forms}(C) = C$
$\mathit{forms}(\Gamma_1, \Gamma_2) = \mathit{forms}(\Gamma_1), \mathit{forms}(\Gamma_2)$
$\mathit{forms}(\Gamma) = \varepsilon$, otherwise
Convention: *forms* returns a multiset of formulas.

B. Environment splitting

Consistently with their interpretation as resources, the affine formulas occurring in typing environments must be used at most once in typing derivations. A crucial feature of our system is that affine values are split into affine and non-affine components according to their type and shared among the sub-terms of a process. Table VI shows how to split bindings and formulas stored in Γ between Γ_1 and Γ_2, written $\Gamma = \Gamma_1 \bowtie \Gamma_2$.

As expected, an exponential formula $!F$ splits as $!F \bowtie\, !F$ and it is propagated to both environments (Split-Exp); an affine formula F splits as $F \bowtie \varepsilon$ and

Table VI Environment Splitting $\qquad \Gamma = \Gamma_1 \bowtie \Gamma_2$

Formulas:

$$\textsc{Split-Exp}\quad !F = !F \bowtie !F \qquad\qquad \textsc{Split-Aff}\quad \frac{F \ \textit{affine}}{F = F \bowtie \varepsilon}$$

Types:

$$\textsc{Split-Exp}\quad \frac{T \in \{\mathsf{Un}, \mathsf{Private}, \mathsf{Ch}(U), \mathsf{SigKey}(U), \mathsf{EncKey}(U), \mathsf{Signed}(U), \mathsf{PubEnc}(U), \mathsf{SymEnc}(U), \mathsf{Fresh}(U_1, U_2)\}}{T = T \bowtie T}$$

$$\textsc{Split-Reading-Key}\quad \frac{T = T \bowtie U \qquad \mathsf{T} \in \{\mathsf{VerKey}, \mathsf{DecKey}\}}{\mathsf{T}(T) = \mathsf{T}(T) \bowtie \mathsf{T}(U)}$$

$$\textsc{Split-Seed}\quad \frac{U = U \bowtie V \qquad \mathsf{T} \in \{\mathsf{SigSeed}, \mathsf{EncSeed}, \mathsf{SymKey}\}}{\mathsf{T}(T, U) = \mathsf{T}(T, U) \bowtie \mathsf{T}(T, V)}$$

$$\textsc{Split-Ref}\quad \frac{T = T \bowtie T' \qquad F = F \bowtie F'}{\{x : T \mid F\} = \{x : T \mid F\} \bowtie \{x : T' \mid F'\}}$$

$$\textsc{Split-Pair}\quad \frac{T = T \bowtie T' \qquad U = U \bowtie U'}{\mathsf{Pair}(x : T, U) = \mathsf{Pair}(x : T, U) \bowtie \mathsf{Pair}(x : T', U')}$$

$$\textsc{Split-Nonce}\quad \mathsf{Nonce} = \mathsf{Nonce} \bowtie \mathsf{Un}$$

Environments:

$$\textsc{Split-Bind}\quad \frac{T = T_1 \bowtie T_2}{u : T = u : T_1 \bowtie u : T_2} \qquad \textsc{Split-Seq}\quad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \qquad \Gamma' = \Gamma'_1 \bowtie \Gamma'_2}{\Gamma, \Gamma' = \Gamma_1, \Gamma'_1 \bowtie \Gamma_2, \Gamma'_2} \qquad \textsc{Split-Empty}\quad \varepsilon = \varepsilon \bowtie \varepsilon$$

Convention: $\bowtie$ is symmetric and we let $\{x : T \mid \varepsilon\}$ stand for T.

Notation: $\Gamma = \Gamma_1 \bowtie \ldots \bowtie \Gamma_n$ is a short-hand for $\Gamma = \Gamma' \bowtie \Gamma_n$ with $\Gamma' = \Gamma_1 \bowtie \ldots \bowtie \Gamma_{n-1}$.

it is thus propagated only to one environment (Split-Lin). The splitting of type bindings is defined by Split-Bind according to the splitting of types, written $T = T_1 \bowtie T_2$. We say that T is exponential if $T = T \bowtie T$: the set of exponential types comprises Un and Private, channel types, writing-key types, signature and ciphertext types, and the type $\mathsf{Fresh}(T, U)$. As we have seen, the treatment of reading-key types depends on their payload type. For example, the type $\mathsf{VerKey}(T)$ splits as $\mathsf{VerKey}(T) \bowtie \mathsf{VerKey}(U)$ if T splits as $T \bowtie U$ (Split-Reading-Key). This means that, if T is exponential ($T = T \bowtie T$), then also $\mathsf{VerKey}(T)$ is exponential; otherwise, if $T = T \bowtie U$ with $T \neq U$, then we have that U is a sort of shallow copy of T lacking all the affine information of the latter, and accordingly the splitting $\mathsf{VerKey}(T) = \mathsf{VerKey}(T) \bowtie \mathsf{VerKey}(U)$ allows to extract the affine information only in one branch of the type derivation. This form of splitting is also applied in Split-Ref and Split-Pair: this simplifies proofs and has a quite limited impact on the expressiveness of the formalism[1]. Key seeds and symmetric keys split according to the previous intuition, with the writing type being always exponential and the reading type handled on the basis of the payload type (Split-Seed). Finally, Nonce splits as Nonce $\bowtie$ Un, thus ensuring that nonces are used at most once with the strong type Nonce, still allowing for several usages with type Un (Split-Nonce). This is necessary to send nonces over the network (with type Un), while being able to check their freshness only once (which requires them to be of type Nonce). We explore this point more in depth in the nonce handshaking example of Section VI.

We note that we purposely chose affine types to hard-code the desirable properties of nonces and keys into the type system. In principle, it might be possible to rely only on refinements and capabilities handled by the logic, but the usage of the structured types underlines the fundamental principles of our approach and saves us from stating assumptions about such refinements.

C. Subtyping

The subtyping rules are given in Table VII. Sub-Pub-Tnt is borrowed from previous type systems for authorization policies [4], [3], [5]. Intuitively, if T is a subtype of Un, then messages of type T may be sent

[1]For example, we rule out protocols where the signing key is used to create signatures of type $\mathsf{Signed}(\{x : \{y : T \mid F_1\} \mid F_2\})$ and the verification key is used once to extract F_1 and once to extract F_2.

Table VII Subtyping $\qquad \Gamma \vdash T <: U$

SUB-REFL
$$\frac{\Gamma \vdash \diamond \qquad \mathit{fnfv}(T) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash T <: T}$$

SUB-PUB-TNT
$$\frac{\Gamma_1 \vdash T <: \mathsf{Un} \qquad \Gamma_2 \vdash \mathsf{Un} <: U}{\Gamma_1 \bowtie \Gamma_2 \vdash T <: U}$$

SUB-REFINE-LEFT
$$\frac{\Gamma \vdash T <: U \qquad fnfv(\{x : T \mid C\}) \subseteq dom(\Gamma)}{\Gamma \vdash \{x : T \mid C\} <: U}$$

SUB-REFINE-RIGHT
$$\frac{\Gamma_1, x : T' \vdash C \qquad \Gamma_2 \vdash T <: U \qquad T = T \bowtie T'}{\Gamma_1 \bowtie \Gamma_2 \vdash T <: \{x : U \mid C\}}$$

SUB-PAIR-COV
$$\frac{\Gamma_1 \vdash T_1 <: T_2 \qquad \Gamma_2, x : T_1' \vdash U_1 <: U_2 \qquad T_1 = T_1 \bowtie T_1'}{\Gamma_1 \bowtie \Gamma_2 \vdash \mathsf{Pair}(x : T_1, U_1) <: \mathsf{Pair}(x : T_2, U_2)}$$

SUB-INV
$$\frac{\Gamma' \vdash T <:> U \qquad \Gamma = \Gamma \bowtie \Gamma' \qquad \mathsf{T} \in \{\mathsf{Ch}, \mathsf{SigKey}, \mathsf{EncKey}, \mathsf{Signed}, \mathsf{PubEnc}, \mathsf{SymEnc}\}}{\Gamma \vdash \mathsf{T}(T) <: \mathsf{T}(U)}$$

SUB-COV
$$\frac{\Gamma' \vdash T <: U \qquad \Gamma = \Gamma \bowtie \Gamma' \qquad \mathsf{T} \in \{\mathsf{VerKey}, \mathsf{DecKey}\}}{\Gamma \vdash \mathsf{T}(T) <: \mathsf{T}(U)}$$

SUB-FRESH-COV
$$\frac{\Gamma' \vdash T_1 <: T_2 \qquad \Gamma' \vdash U_1 <: U_2 \qquad \Gamma = \Gamma \bowtie \Gamma'}{\Gamma \vdash \mathsf{Fresh}(T_1, U_1) <: \mathsf{Fresh}(T_2, U_2)}$$

SUB-SEED
$$\frac{\Gamma' \vdash T_1 <:> T_2 \qquad \Gamma' \vdash U_1 <: U_2 \qquad \Gamma = \Gamma \bowtie \Gamma' \qquad \mathsf{T} \in \{\mathsf{SigSeed}, \mathsf{EncSeed}, \mathsf{SymKey}\}}{\Gamma \vdash \mathsf{T}(T_1, U_1) <: \mathsf{T}(T_2, U_2)}$$

SUB-FRESH-PAIR
$$\frac{\Gamma \vdash \diamond \qquad \mathit{fnfv}(T) \cup \mathit{fnfv}(U) \subseteq \mathit{dom}(\Gamma) \qquad T = T \bowtie T' \qquad U = U \bowtie U' \qquad x \notin \mathit{dom}(\Gamma)}{\Gamma \vdash \mathsf{Fresh}(T, U) <: \mathsf{Pair}(x : T', U')}$$

Notation: $\Gamma \vdash T <:> U$ stands for $\Gamma \vdash T <: U$ and $\Gamma \vdash U <: T$.

to the attacker (i.e., they are public); similarly, if T is supertype of Un, then messages of type T may come from the attacker (i.e., they are tainted). SUB-PUB-TNT describes the fact that messages that may be sent to the attacker may also come from the attacker. The subtyping rules for Un are mostly standard [4]. SUB-REFINE-LEFT is borrowed from [3], [6] and allows removing refinements by subtyping. SUB-REFINE-RIGHT is more interesting: this rule allows to add refinements by subtyping, as long as the corresponding predicates are entailed in the environment. In standard refinement type systems, this rule looks as follows:

$$\frac{\Gamma, x : T \vdash C \qquad \Gamma \vdash T <: U}{\Gamma \vdash T <: \{x : U \mid C\}}$$

A natural adaptation to an affine setting would be:

$$\frac{\Gamma_1, x : T \vdash C \qquad \Gamma_2 \vdash T <: U}{\Gamma_1 \bowtie \Gamma_2 \vdash T <: \{x : U \mid C\}}$$

Unfortunately, this rule is unsound. In our setting we must split not only the typing environment as expected, but also T, otherwise the resources in T could be used twice and we could prove, for instance, $\varepsilon \vdash \{x : \mathsf{Un} \mid C\} <: \{z : \{x : \mathsf{Un} \mid C\} \mid C\}$. The rule for pairs is covariant and presents a similar problem: the type T_1 has to be split, as it would otherwise be used twice in the premise (SUB-PAIR-COV). For the sake of simplicity, in both of these rules we choose to push all affine resources into one type, similarly to what we do for environment splitting. As expected, channel types are invariant (SUB-INV). Perhaps surprisingly, we have to require the typing environment Γ' in the premises to be exponential ($\Gamma = \Gamma \bowtie \Gamma'$ implies that Γ' is a copy of Γ where all affine resources have been stripped away) as the type system would otherwise be unsound. To illustrate this point, suppose we do not require Γ' to be exponential and just use in the premises the whole environment $\Gamma = A$. We would then be able to prove $A \vdash \mathsf{Ch}(\{x : \mathsf{Un} \mid A\}) <: \mathsf{Un}$ by SUB-INV, since $\varepsilon \vdash \{x : \mathsf{Un} \mid A\} <: \mathsf{Un}$ by SUB-REFINE-LEFT and $A \vdash \mathsf{Un} <: \{x : \mathsf{Un} \mid A\}$ by SUB-REFINE-RIGHT. A channel of type Un can be sent to the opponent. The opponent could use this channel arbitrarily often to write messages of type Un and all of them would be given type $\{x : \mathsf{Un} \mid A\}$ by the processes reading from that channel. This would be unsound as we have only one instance of A in the typing environment and, in particular, we can prove $A \vdash \mathsf{Un} <: \{x : \mathsf{Un} \mid A\}$ only once. The remaining rules are standard, except for SUB-FRESH-PAIR, which allows for downgrading a fresh packet to a pair where all the affine information has been pruned away. Indeed, as we detail below, in our setting fresh packets are simply obtained by pairing a term with a nonce.

The lack of an explicit transitivity rule is an important design choice for our subtyping relation, as it is necessary to make this relation decidable. Transitivity can be recovered as a derived property through the interaction among the different subtyping rules.

D. Typing terms

The type system relies on a typed interface for constructors and destructors, which is defined in Table VIII and is mostly standard. We just note that we introduce a specific typing rule $\mathsf{pair} : (T, U) \mapsto \mathsf{Fresh}(T, U)$ for pairs intended to bind a term to a nonce. Standard pairs should instead be typed using PAIR, as explained below. This enables us to embed the nonce-checking mechanism into the type system.

The typing rules for terms are reported in Table IX. ENV is standard and allows for typing names and variables by projection from Γ. KEY is also standard and allows for typing keys. TERM gives signatures, encryptions, and fresh packets the type expressed in their typing annotation. The reader might wonder why we do not use standard rules for constructors, for instance, the following typing rule for signatures:

$$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash K : \mathsf{SigKey}(T)}{\Gamma \vdash \mathsf{sign}(M, K) : \mathsf{Signed}(T)}$$

The reason is that this rule, albeit standard in security type systems, is unsound in an affine setting. Assume T is affine: for applying such a rule, one needs to give M type T, i.e., to consume the affine information expressed in T. A crucial property of type systems is subject reduction, i.e., typing is preserved by process reduction. Since signatures can be duplicated (e.g., by the attacker), we should be able to type-check an unbounded number of signatures. The rule above, however, would allow us to type-check only one signature, thus breaking subject reduction. Duplicating signatures, however, should be safe, since the receiver derives the affine information not from the type of the signature, but from the type of the verification key. Hence, we allow for the duplication of signatures and we type them according to their type at creation time, namely, the type expressed in the typing annotation. The rule TERM is required in our soundness proofs to type terms obtained at run-time, but it is not part of the type-checker, since static processes do not contain annotated terms.

SUB is the standard subsumption rule. PAIR is used to type-check pairs, whenever we do not want to use them to bind a term with a nonce. REFINE allows for giving the type $\{x : T \mid C\}$ to a term M, provided that M has type T and the predicate C holds for M.

E. Typing processes

The typing rules for processes are listed in Table X. Below, we only comment the non-standard rules. PROC-REPL-IN requires that the typing environment Γ' used to type-check the continuation process is exponential (recall that $\Gamma \bowtie \Gamma'$ implies Γ' exponential). This is necessary as process $!\mathsf{in}(M, x).P$ may spawn an unbounded number of copies of P, but P is type-checked only once. PROC-DES and PROC-DES-EQ are self-explanatory: we just note that we track equalities as exponential formulas in the typing environment to make the type-checking more flexible (we recall that we assume the logic to be equipped with a theory of equality). Given a typed interface $f : (T_1, \ldots, T_n) \rightarrow T$ for constructor f, PROC-CONSTR checks that the arguments of the term construction have type $T_1, \ldots, T_n$, that the result variable x is annotated with type T, and that the continuation process is well-typed in a typing environment extended with the binding $x : T$. The typing rule PROC-SPLIT for pair splitting is standard for dependent pairs. PROC-ASSERT says that the assertion of C is justified if C is entailed in the typing environment. We say that a process is *simple* if all restrictions and assumptions occurring therein are guarded by an output, an input, or a let and, in the following, we let S range over simple processes. PROC-PAR applies to the parallel composition of simple processes: this rule splits the typing environment and type-checks each of the two processes independently. PROC-EXTR is used to type-check processes that are not simple. This rule relies on the relation $P \rightsquigarrow [\Gamma \parallel S]$, which, given a process P, yields a typing environment Γ capturing the top-level restrictions and assumptions in P and the simple process S obtained from P by erasing such elements. For instance, $\mathsf{new}\ m : \mathsf{Un}.(\mathsf{assume}\ \mathsf{p}(m) \mid \mathsf{out}(c, n)) \rightsquigarrow [m : \mathsf{Un}, \mathsf{p}(m) \parallel \mathbf{0} \mid \mathsf{out}(c, n)]$. The idea is that P is well-typed if the corresponding simple process S is well-typed in a typing environment extended with Γ.

F. Soundness results

The first property of our type system is subject reduction. Following the approach proposed by Kobayashi et al. in their type system for enforcing the linear usage of channels in the π-calculus [25], subject reduction is proved as a corollary of a more general theorem in which we track affine resources (i.e., keys and nonces) and weaken their types as soon as they are used[2].

[2]This is the reason why the definition of generative type in Table X accounts for types of the form $\mathsf{SigSeed}(U; V)$, with V being the exponential counterpart of U. This type is given to seeds after the corresponding verification key has been used to extract affine information, and thus can only be used with type V.

Table VIII Typing Constructors and Destructors $f : (T_1, \ldots, T_n) \mapsto U, g : (T_1, \ldots, T_n) \mapsto U$

ek	$: \mathsf{EncSeed}(T; U) \mapsto \mathsf{EncKey}(T)$	eq	$: (T, T) \mapsto T$
dk	$: \mathsf{EncSeed}(T; U) \mapsto \mathsf{DecKey}(U)$	dec	$: (\mathsf{PubEnc}(T), \mathsf{DecKey}(T)) \mapsto T$
sk	$: \mathsf{SigSeed}(T; U) \mapsto \mathsf{SigKey}(T)$	ver	$: (\mathsf{Signed}(T), \mathsf{VerKey}(T)) \mapsto T$
vk	$: \mathsf{SigSeed}(T; U) \mapsto \mathsf{VerKey}(U)$	sdec	$: (\mathsf{SymEnc}(U), \mathsf{SymKey}(T; U)) \mapsto U$
enc	$: (T, \mathsf{EncKey}(T)) \mapsto \mathsf{PubEnc}(T)$	check	$: (\mathsf{Fresh}(T, U), \mathsf{Nonce}) \mapsto T$
sign	$: (T, \mathsf{SigKey}(T)) \mapsto \mathsf{Signed}(T)$		
senc	$: (T, \mathsf{SymKey}(T; U)) \mapsto \mathsf{SymEnc}(T)$		
pair	$: (T, U) \mapsto \mathsf{Fresh}(T, U)$		

Table IX Typing Terms $\Gamma \vdash M : T$

$$\textsc{Env}\quad \frac{\Gamma \vdash \diamond \qquad u : T \in \Gamma}{\Gamma \vdash u : T} \qquad\qquad \textsc{Key}\quad \frac{f \in \{\mathsf{ek}, \mathsf{dk}, \mathsf{sk}, \mathsf{vk}\} \qquad f : T \mapsto U \qquad \Gamma \vdash M : T}{\Gamma \vdash f(M) : U}$$

$$\textsc{Term}\quad \frac{\begin{array}{c}\Gamma \vdash \diamond \qquad (f, T) \in \{(\mathsf{sign}, \mathsf{Signed}(U)), (\mathsf{enc}, \mathsf{PubEnc}(U)), (\mathsf{senc}, \mathsf{SymEnc}(U)), (\mathsf{pair}, \mathsf{Fresh}(U_1, U_2))\} \\ fn(M_1) \cup fn(M_2) \cup fn(T) \subseteq dom(\Gamma)\end{array}}{\Gamma \vdash f(M_1, M_2)_T : T}$$

$$\textsc{Sub}\quad \frac{\Gamma_1 \vdash M : T \qquad \Gamma_2 \vdash T <: T'}{\Gamma_1 \bowtie \Gamma_2 \vdash M : T'} \qquad \textsc{Pair}\quad \frac{\Gamma_1 \vdash M_1 : T_1 \qquad \Gamma_2 \vdash M_2 : T_2\{M_1/x\}}{\Gamma_1 \bowtie \Gamma_2 \vdash \mathsf{pair}(M_1, M_2) : \mathsf{Pair}(x : T_1, T_2)} \qquad \textsc{Refine}\quad \frac{\Gamma_1 \vdash C\{M/x\} \qquad \Gamma_2 \vdash M : T}{\Gamma_1 \bowtie \Gamma_2 \vdash M : \{x : T \mid C\}}$$

Theorem V.1 (Subject Reduction) *Let P be a closed, static process and Γ a typing environment such that $\Gamma \vdash P$. If $P \rightarrow^* Q$, then $\Gamma \vdash Q$.*

An additional property of our type system is that we can provide typing annotations such that any opponent O is well-typed. In particular, we write O_{Un} to denote the process obtained from O by changing the typing annotations for term constructions into $\mathsf{Signed}(\mathsf{Un}), \mathsf{PubEnc}(\mathsf{Un}), \mathsf{Fresh}(\mathsf{Un}, \mathsf{Un})$ and so on. Notice that typing annotations do not affect the semantics of processes, thus we are not limiting the opponent.

Lemma V.2 (Opponent Typability) *For all opponents O, $\widetilde{a : \mathsf{Un}} \vdash O_{\mathsf{Un}}$, where $fn(O) = \{\widetilde{a}\}$.*

By combining subject reduction and opponent typability, we can prove the main result of this section, i.e., well-typed processes are robustly safe.

Theorem V.3 (Robust Safety) *Let P be a closed, static process such that $fn(P) = \{m_1, \ldots, m_k\}$ and let $m_1 : \mathsf{Un}, \ldots, m_k : \mathsf{Un} \vdash P$. Then P is robustly safe.*

VI. Examples

In this section we show our type system at work, illustrating how to verify the nonce handshake presented in Section IV and several other protocols.

A. Nonce handshakes

The process for the nonce handshake from Section IV, along with the most important typing annotations, is shown below[3]:

$$\begin{aligned}
A &\triangleq \mathsf{in}(c, x).\mathsf{new}\ m : \mathsf{Un}. \\
&\quad\ \mathsf{assume}\ \mathsf{p}(m) \mid \mathsf{let}\ y = (m, x)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ z = \mathsf{sign}(y, \mathsf{sk}(k))\ \mathsf{in}\ \mathsf{out}(c, z) \\
B &\triangleq \mathsf{new}\ n : \mathsf{Nonce}.\mathsf{out}(c, n).\mathsf{in}(c, x). \\
&\quad\ \mathsf{let}\ y = \mathsf{ver}(x, \mathsf{vk}(k))\ \mathsf{in} \\
&\quad\ \mathsf{let}\ z = \mathsf{check}(y, n)\ \mathsf{in}\ \mathsf{assert}\ \mathsf{p}(z) \\
P &\triangleq \mathsf{new}\ k : \mathsf{SigSeed}(T; T).(A \mid B) \\
T &\triangleq \mathsf{Fresh}(\{x : \mathsf{Un} \mid \mathsf{p}(x)\}, \mathsf{Un})
\end{aligned}$$

The nonce generated by B has type Nonce. This type is affine and splits as $\mathsf{Nonce} \bowtie \mathsf{Un}$: as we said, in this way B can use n once with type Nonce and arbitrarily often with type Un. The nonce is sent on the network with type Un. A receives the nonce and signs the term (m, n), which binds the message m to the nonce n. This term has type $\mathsf{Fresh}(\{x : \mathsf{Un} \mid \mathsf{p}(x)\}, \mathsf{Un})$: this type is exponential, which makes also the type $\mathsf{VerKey}(T)$ exponential (i.e., this verification key can be used several times). After receiving and verifying the signature, B checks the freshness of the nonce via the destructor application $\mathsf{check}(y, n)$. The first argument has the exponential type $\mathsf{Fresh}(\{x : \mathsf{Un} \mid \mathsf{p}(x)\}, \mathsf{Un})$,

[3]For the sake of readability, we omit branches of the form "$\mathsf{else}\ \mathbf{0}$".

Table X Typing Processes $\Gamma \vdash P$

$$\textsc{Proc-Stop}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}} \qquad \textsc{Proc-Out}\quad \frac{\Gamma_1 \vdash M : \mathsf{Ch}(T) \qquad \Gamma_2 \vdash N : T \qquad \Gamma_3 \vdash P}{\Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \vdash \mathsf{out}(M, N).P} \qquad \textsc{Proc-In}\quad \frac{\Gamma_1 \vdash M : \mathsf{Ch}(T) \qquad \Gamma_2, x : T \vdash P}{\Gamma_1 \bowtie \Gamma_2 \vdash \mathsf{in}(M, x : T).P}$$

$$\textsc{Proc-Repl-In}\quad \frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma', x : T \vdash P \qquad \Gamma = \Gamma \bowtie \Gamma'}{\Gamma \vdash !\mathsf{in}(M, x : T).P} \qquad \textsc{Proc-Des}\quad \frac{g : (T_1, T_2) \mapsto T \qquad g \neq \mathsf{eq} \qquad i \in [1,2].\Gamma_i \vdash M_i : T_i \qquad \Gamma_3, x : T \vdash P \qquad \Gamma_3 \vdash Q}{\Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \vdash \mathsf{let}\ x : T = g(M_1, M_2)\ \mathsf{in}\ P\ \mathsf{else}\ Q}$$

$$\textsc{Proc-Des-Eq}\quad \frac{i \in [1,2].\Gamma_i \vdash M_i : T \qquad \Gamma_3, x : T, !(M_1 = M_2) \vdash P \qquad \Gamma_3 \vdash Q}{\Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \vdash \mathsf{let}\ x : T = \mathsf{eq}(M_1, M_2)\ \mathsf{in}\ P\ \mathsf{else}\ Q} \qquad \textsc{Proc-Constr}\quad \frac{f : (T_1, \dots, T_n) \mapsto T \qquad i \in [1,n].\ \Gamma_i \vdash M_i : T_i \qquad \Gamma_{n+1}, x : T \vdash P \qquad \Gamma = \Gamma_1 \bowtie \dots \bowtie \Gamma_{n+1}}{\Gamma \vdash \mathsf{let}\ x : T = f(M_1, \dots, M_n)\ \mathsf{in}\ P}$$

$$\textsc{Proc-Split}\quad \frac{\Gamma_1 \vdash M : \mathsf{Pair}(x : T, U) \qquad \Gamma_2, x : T, y : U \vdash P}{\Gamma_1 \bowtie \Gamma_2 \vdash \mathsf{let}\ (x : T, y : U) = M\ \mathsf{in}\ P} \qquad \textsc{Proc-Assert}\quad \frac{\Gamma \vdash C}{\Gamma \vdash \mathsf{assert}\ C} \qquad \textsc{Proc-Par}\quad \frac{\Gamma_1 \vdash S_1 \qquad \Gamma_2 \vdash S_2}{\Gamma_1 \bowtie \Gamma_2 \vdash S_1 \mid S_2}$$

$$\textsc{Proc-Extr}\quad \frac{P \rightsquigarrow [\Gamma' \parallel S] \qquad \Gamma, \Gamma' \vdash S \qquad \mathit{fnfv}(P) \subseteq \mathit{dom}(\Gamma) \qquad \neg(P\ \mathit{simple})}{\Gamma \vdash P}$$

Extraction:

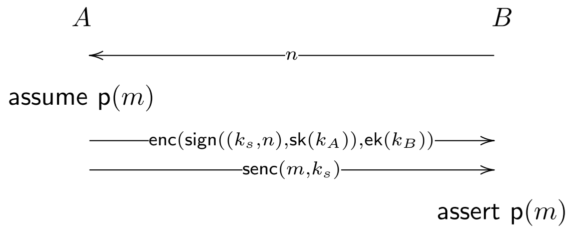
$$\textsc{Extr-New}\quad \frac{P \rightsquigarrow [\Gamma \parallel S] \qquad T \text{ generative}}{\mathsf{new}\ a : T.P \rightsquigarrow [a : T, \Gamma \parallel S]} \qquad \textsc{Extr-Assume}\quad \mathsf{assume}\ C \rightsquigarrow [C \parallel \mathbf{0}] \qquad \textsc{Extr-Empty}\quad S \rightsquigarrow [\varepsilon \parallel S] \qquad \textsc{Extr-Par}\quad \frac{P \rightsquigarrow [\Gamma_P \parallel S_P] \qquad Q \rightsquigarrow [\Gamma_Q \parallel S_Q] \qquad \neg(P \mid Q\ \mathit{simple})}{P \mid Q \rightsquigarrow [\Gamma_P, \Gamma_Q \parallel S_P \mid S_Q]}$$

T generative iff $T \in \{\mathsf{Un}, \mathsf{Ch}(U), \mathsf{Private}, \mathsf{Nonce}\} \cup \{\mathsf{SigSeed}(U; V), \mathsf{EncSeed}(U; V), \mathsf{SymKey}(U; V) \mid U = V \vee U = U \bowtie V\}$

while the second argument has the affine type Nonce. Our type system ensures that whenever such a destructor application succeeds, the result z has indeed the affine type $\{x : \mathsf{Un} \mid \mathsf{p}(x)\}$. Hence, this refinement type justifies the final assert $\mathsf{p}(z)$.

B. Session keys and private channels

Session keys are used in real-life protocols for both efficiency and security reasons. However, the interesting point in our setting is that, by their own ephemeral nature, session keys can additionally be used to exchange an affine formula. In the protocol shown below, a session key is exchanged between A and B and then used by A to authenticate a message with B.

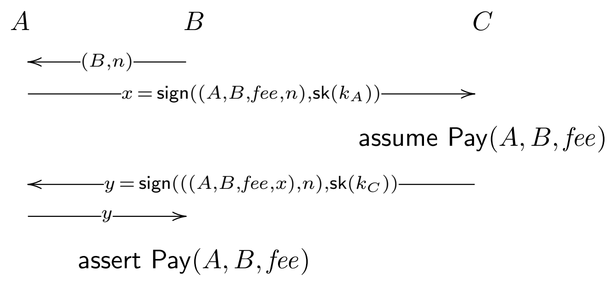


The (exponential) long-term keys derived from k_A and k_B are typed similarly to the signature key pair described in Section VI-A, so we will focus on the typing of the session key. The session key k_s is created with type $\mathsf{SymKey}(T; T)$, where $T = \{x : \mathsf{Private} \mid \mathsf{p}(x)\}$. The type T is affine, since it contains the affine predicate $\mathsf{p}(x)$. A accesses k_s twice, once to sign it and once to encrypt m. The type binding for k_s splits as $k_s : \mathsf{SymKey}(T; T) \bowtie k_s : \mathsf{SymKey}(T; \mathsf{Private})$, where the former (strong) type is used in the signing operation, while the latter (weak) type is used to encrypt m. Hence B receives k_s with the strong type. This allows him to give type $\{x : \mathsf{Private} \mid \mathsf{p}(x)\}$ to the message w, which he obtains by decrypting the last message $\mathsf{senc}(m, k_s)$ using k_s. This type justifies the final assertion. Notice that this protocol constitutes a cryptographic implementation of a private channel of type $\mathsf{Ch}(\{x : \mathsf{Private} \mid \mathsf{p}(x)\})$. Though this implementation is not fully abstract, since messages sent over restricted channels cannot be blocked, as opposed to cryptographic packets sent over public channels, this encoding does preserve safety.

C. *Ratification of credentials*

Bowers et al. exemplify the concept of consumable credentials [29] on an e-payment protocol. We consider a slightly simplified version of this protocol and show how to type-check it.

The idea behind a consumable credential is simple: if Alice wants to buy a good from Bob's store, she may prefer to avoid paying cash and just provide a proof that Bob will indeed be paid. If Bob does not need to receive the money immediately, as it happens for credit card payments, this credential suffices as a credit check. It is essential for such a proof to be *consumable*: on Alice's side, this ensures that Bob cannot receive the payment more than once; on Bob's side, this guarantees that the transaction is fresh and Alice is actually allowed to withdraw the money. The protocol involves Alice (A), Bob (B), and a Clearing House (C) acting as a trusted third party:


A
B
C
(B,n)
$x = \mathsf{sign}((A,B,\mathit{fee},n),\mathsf{sk}(k_A))$
assume $\mathsf{Pay}(A,B,\mathit{fee})$
$y = \mathsf{sign}(((A,B,\mathit{fee},x),n),\mathsf{sk}(k_C))$
y
assert $\mathsf{Pay}(A,B,\mathit{fee})$

B creates a nonce n and sends it to A, who signs it along with a withdrawal request *fee* to pay B. C verifies that A can indeed withdraw the amount of money *fee* and generates a consumable credential for A, which consists of a signature on the message received from A, the nonce, and the other parameters. Now A can provide the payment proof to B, who checks the freshness of the nonce and can then exercise his payment rights whenever he prefers. The type-checking of this protocol follows the same lines as the protocol discussed in Section VI-A. The only interesting type is that of C's verification key that is used by B to verify the received signature y, i.e, $\mathsf{VerKey}(\mathsf{Fresh}(\langle x_{id} : \mathsf{Un}, y_{id} : \mathsf{Un}, \{z_{fee} : \mathsf{Un} \mid \mathsf{Pay}(x_{id}, y_{id}, z_{fee})\}, \mathsf{Un}\rangle, \mathsf{Un}))$[4]. This type says that C's key is used to sign messages whose components are, respectively, the identifiers x_{id} and y_{id}, the withdrawal request z_{fee}, for which the linear predicate $\mathsf{Pay}(x_{id}, y_{id}, z_{fee})$ is derivable, and the nonce at type Un. The types of the messages and the key are exponential, i.e., the verification key can be used arbitrarily often: the freshness of the withdrawal request is ensured by the nonce.

[4]For the sake of readability, we use dependent tuple types, which can be encoded straightforwardly through dependent pairs.

D. *One-click file hosting*

One-click file hosting services like Megaupload and Rapidshare allow users to easily upload a file from their hard drive to a server free of charge, returning a URL used to retrieve the file later on. Here we design a simple one-click file hosting service, GIGASHARE (GS). In the following we show how to model GS and we show by type-checking that its implementation is robustly safe. This example relies on the nonce handshake mechanism to transfer affine information over the network, which is then used to enforce an authorization policy.

For the sake of readability, we rely on some shorthands: we again omit empty else-branches and use dependent tuple types of the form $\langle x_1{:}T_1, \ldots, x_n{:}T_n\rangle$. We omit bound variables in dependent tuple types if we never refer to them and we use the distinguished symbol "_" to stand for a variable in a binder when that variable is not used in the continuation. We let $\mathsf{if}\ (M{=}{=}N)\ \mathsf{then}\ P\ \mathsf{else}\ Q$ denote $\mathsf{let}\ \_ = \mathsf{eq}(M,N)\ \mathsf{in}\ P\ \mathsf{else}\ Q$

1) Description of GIGASHARE*:* the architecture of GS divides users in two categories: *trial* users and *premium* users. Both kinds of users must register to the site to access the service: the registration is free for trial users, while it requires the payment of a fee for premium users. The differences between trial users and premium users concern both the access to uploaded contents and the management of downloads: trial users can download a limited number of files a day from GS and suffer limitations on the available bandwidth, due to the allocation of only a fixed number of free download slots; conversely, premium users can download an arbitrary number of files a day and launch as many parallel downloads as they like. We assume that each trial user can download only two files and only a single trial user can download from the site at a time: these parameters can be changed straightforwardly.

2) Defining the usage policy: to model the usage policy of GIGASHARE we rely on five different predicates: $\mathsf{Req}(x,y)$, $\mathsf{CanDL}(x,y)$, Slot, $\mathsf{Trial}(x)$, and $\mathsf{Premium}(x)$. The predicate $\mathsf{Req}(x,y)$ states that a user x has requested the file y, the predicate $\mathsf{CanDL}(x,y)$ grants x the permission to download y from the server, while the predicate Slot is used to constrain the number of parallel downloads by trial users. Finally, the predicates $\mathsf{Trial}(x)$ and $\mathsf{Premium}(x)$ distinguish trial and premium users, giving them different capabilities on the basis of formulas defining the GS usage policy. For example, if the assumption $\mathsf{Premium}(p)$ is spawned an unbounded number of times for each premium user p, the policy F below allows premium users to obtain download capabilities on demand:

$$F \triangleq !\forall x.\forall y.((\mathsf{Premium}(x) \otimes \mathsf{Req}(x,y)) \multimap \mathsf{CanDL}(x,y))$$

Conversely, if the predicate $\mathsf{Trial}(t)$ is assumed n times for each trial user t, the following policy G allows trial users to download at most n files from the site, provided that there are available download slots:

$$G \triangleq !\forall x.\forall y.((\mathsf{Trial}(x) \otimes \mathsf{Req}(x,y) \otimes \mathsf{Slot}) \multimap \mathsf{CanDL}(x,y))$$

3) Registering to GIGASHARE*:* we omit from our model the registration phase of the service. We just point out that we assume that at the end of the registration procedure the site stores in a database a triple $(u, acc, \mathsf{vk}(k_u))$, where u is the id of the user, acc defines the account type, and $\mathsf{vk}(k_u)$ is a verification key used to authenticate requests from u. The database can be queried for an identity u via the channel udb and returns the triple associated to u along the restricted channel $udbr$, if u is a registered user that is still allowed to download files. Thus, the database is the component of the system that keeps track of whether a trial user has reached the limit of his allowed downloads: we assume that it answers only two queries about each specific trial user while it answers an unlimited number of queries about premium users. Modelling this component in the applied π-calculus is straightforward.

4) Downloading from GIGASHARE*:* the server waits for requests on the public channel c and replies with a nonce on channel d for the requesting user, who in turn answers providing his identity u and a signed triple (u, f, r), paired with the nonce. This signed request specifies the desired file f and the return address r used to get the file. After receiving the response, the server queries the user database on udb to retrieve the account information. Finally, the server dispatches the request, along with the associated verification key and the nonce, to a specific download manager component PM or TM, depending on the account being premium or trial. Before the dispatching, the download interface assumes the predicate associated to the account of the user. The code for the described process is shown below:

```
DL ≜ !in(c, _).new n : Nonce.out(d, n).in(d, x).
     let (x_u, x_r) = x in
     out(udb, x_u).in(udbr, y).
     let (y_u, y_a, y_v) = y in
     if (x_u == y_u) then
     if (y_a == premium)
     then
       assume Premium(y_u) | out(pm, (y_u, x_r, y_v, n))
     else
       assume Trial(y_u) | out(tm, (y_u, x_r, y_v, n))
```

Download managers wait for requests from the download interface over a restricted channel, verify their validity using the provided verification key, assess their freshness via the associated nonce, and then proceed depending on the account type. If the requesting user has a premium account, the manager asserts that she can download the file, forwards the request to a file database fdb, and finally provides the file to the end user:

```
PM ≜ !in(pm, x).let (x_u, x_r, x_v, x_n) = x in
     let z = ver(x_r, x_v) then
     let y = check(z, x_n) then
     let (y_u, y_f, y_r) = y in
     if (x_u == y_u) then
     assert CanDL(y_u, y_f) |
     out(fdb, y_f).in(fdbr, w).out(y_r, w)
```

Otherwise, if the requesting user has a trial account, the manager asks for the token on a free download slot via an input over channel $slot$ and only then behaves as the corresponding component for premium users; finally, it releases the token by outputting the dummy message ack on $slot$:

```
TM ≜ !in(tm, x).let (x_u, x_r, x_v, x_n) = x in
     let z = ver(x_r, x_v) then
     in(slot, _).
     let y = check(z, x_n) then
     let (y_u, y_f, y_r) = y in
     if (x_u == y_u) then
     assert CanDL(y_u, y_f) |
     out(fdb, y_f).in(fdbr, w).out(y_r, w).
     assume Slot | out(slot, ack)
```

5) Type-checking GIGASHARE*:* the file hosting service can then be modeled as the process GS below:

```
GS ≜ assume F | assume G |
     new udb : T_un.new udbr : T_udbr.
     new pm : T_p.new tm : T_t.
     new fdb : T_un.new fdbr : T_un.new slot : T_s.
     (assume Slot | out(slot, ack) | DL | PM | TM)
```

This process is well-typed in our type system under the following assumptions:

$$\begin{array}{lcl}
T_r & \triangleq & \mathsf{Fresh}(\langle x{:}\mathsf{Un}, \{y{:}\mathsf{Un} \mid \mathsf{Req}(x,y)\}, \mathsf{Ch}(\mathsf{Un})\rangle, \mathsf{Un}) \\
T_{udbr} & \triangleq & \mathsf{Ch}(\langle \mathsf{Un}, \mathsf{Un}, \mathsf{VerKey}(T_r)\rangle) \\
T_p & \triangleq & \mathsf{Ch}(\langle\{x : \mathsf{Un} \mid \mathsf{Premium}(x)\}, \mathsf{Signed}(T_r), \\
 & & \mathsf{VerKey}(T_r), \mathsf{Nonce}\rangle) \\
T_t & \triangleq & \mathsf{Ch}(\{x : \mathsf{Un} \mid \mathsf{Trial}(x)\}, \mathsf{Signed}(T_r), \\
 & & \mathsf{VerKey}(T_r), \mathsf{Nonce}\rangle) \\
T_s & \triangleq & \mathsf{Ch}(\{x : \mathsf{Un} \mid \mathsf{Slot}\}) \\
T_{un} & \triangleq & \mathsf{Ch}(\mathsf{Un})
\end{array}$$

The nonce n is created by the DL with type Nonce, which splits as $\mathsf{Nonce} = \mathsf{Nonce} \bowtie \mathsf{Un}$. The nonce is sent to the user with the weak type Un and, after a request, to the respective download manager PM or

TM with the strong type Nonce, which is used to verify the freshness of the request. Due to this nonce handshaking, the type $\mathsf{VerKey}(T_r)$ of the verification keys used to validate download requests is exponential, thus allowing an unbounded number of usages for these keys. The predicate $\mathsf{Req}(x, y)$, which binds the identity of the user x to the name of the requested file y, is extracted by the download managers upon verification of signed requests; this predicate is assumed by the requesting users and consumed on the server's side by the affine implications in the aforementioned authorization policy. The process modeling the requesting users is straightforward. The types T_p and T_t, given to the channels guarding the download managers for premium and trial users respectively, allow the transfer of the assumed predicate about the requesting user from DL to the download managers. Finally, the type T_s of the channel slot is used to handle the token on the predicate Slot: reading from the channel takes the token, while writing on it releases the token.

VII. Conclusions

Resource-aware authorization policies are a crucial ingredient for the analysis of real-life applications, where the freshness of the communication and the effective number of transactions cannot be overlooked (e.g., e-banking, e-voting, etc.). In this paper we presented the first type system to statically enforce distributed resource-aware authorization policies in cryptographic protocols. The distinctive feature of our type system is that the derivability of affine information is witnessed by the (affine) type of the cryptographic material. We showed how our approach can be used to verify a number of interesting applications, including authentication protocols, nonce handshakes, session-key establishment protocols, an e-payment protocol, and a model of a file hosting service.

As a future work, it would be interesting to integrate the typing discipline illustrated in this paper in a much richer typed language such as $F^{\star}$ [8] and to consider more advanced cryptographic primitives, such as zero-knowledge proofs and secure multi-party computations.

Acknowledgements. This work was supported by the Emmy Noether program and the initiative for excellence of the German federal government and by MIUR Project *IPODS Interacting Processes in Open Distributed Systems*. We would like to thank Nikhil Swamy for his many precious suggestions.

References

[1] M. Abadi, "Logic in access control," in *Proc. 18th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 2003, pp. 228–233.

[2] L. Bauer, L. Jia, and D. Sharma, "Constraining credential usage in logic-based access control," in *Proc. 23rd IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, 2010, pp. 154–168.

[3] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," in *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, 2008, pp. 17–32.

[4] C. Fournet, A. D. Gordon, and S. Maffeis, "A type discipline for authorization in distributed systems," in *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, 2007, pp. 31–45.

[5] M. Backes, C. Hriţcu, and M. Maffei, "Type-checking zero-knowledge," in *15th ACM Conference on Computer and Communications Security (CCS 2008)*. ACM Press, 2008, pp. 357–370.

[6] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *Proc. 37th Symposium on Principles of Programming Languages (POPL)*. ACM, 2010, pp. 445–456.

[7] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in fine," in *Proc. 19th European Symposium on Programming (ESOP)*, 2010, pp. 529–549.

[8] N. Swamy, J. Chen, C. Fournet, K. Bharagavan, and J. Yang, "Security programming with refinement types and mobile proofs," Microsoft Research, Tech. Rep. MSR-TR-2010-149, 2010.

[9] J.-Y. Girard, "Linear logic: its syntax and semantics," in *Advances in Linear Logic*, ser. London Mathematical Society Lecture Note Series, vol. 22, 1995, pp. 3–42.

[10] A. S. Troelstra, "Lectures on linear logic," CSLI Stanford, Lecture Notes Series nr. 29, 1992.

[11] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Proc. 28th Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2001, pp. 104–115.

[12] M. Backes, M. Maffei, and E. Mohammadi, "Computationally sound abstraction and verification of secure multi-party computations," in *Proc. IARCS Annual Conference on on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, ser. LIPIcs, vol. 8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010, pp. 352–363.

[13] A. D. Gordon and A. Jeffrey, "Authenticity by typing for security protocols," *Journal of Computer Security*, vol. 11, no. 4, pp. 451–519, 2003.

[14] ——, “Types and effects for asymmetric cryptographic protocols,” *Journal of Computer Security*, vol. 12, no. 3, pp. 435–484, 2004.

[15] J. Borgstrom, J. Chen, and N. Swamy, “Verifying stateful programs with substructural state and hoare types,” in *Proc. 5th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV)*. ACM Press, 2011, pp. 15–26.

[16] M. Fahndrich and R. DeLine, “Adoption and focus: practical linear types for imperative programming,” in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2002, pp. 13–24.

[17] J. G. Cederquist, R. Corin, M. A. C. Dekker, S. Etalle, J. I. den Hartog, and G. Lenzini, “Audit-based compliance control,” *International Journal of Information Security*, vol. 6, no. 2-3, pp. 133–151, 2007.

[18] P. Wadler, “Linear types can change the world!” in *Programming Concepts and Methods*. North, 1990.

[19] N. Kobayashi, “Quasi-linear types,” in *Proc. 26th Symposium on Principles of Programming Languages (POPL)*, 1999, pp. 29–42.

[20] D. N. Turner, P. Wadler, and C. Mossin, “Once upon a type,” in *Proc. Conference on Functional Programming Languages and Computer Architecture (FPCA)*, 1995, pp. 1–11.

[21] Y. Mandelbaum, D. Walker, and R. Harper, “An effective theory of type refinements,” in *Proc. of the 8th ACM SIGPLAN international conference on Functional programming (ICFP)*. ACM Press, 2003, pp. 213–225.

[22] K. Bierhoff and J. Aldrich, “Modular typestate checking of aliased objects,” in *Proc. 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA)*. ACM Press, 2007, pp. 301–320.

[23] N. Kobayashi, B. C. Pierce, and D. N. Turner, “Linearity and the pi-calculus,” *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 914–947, 1999.

[24] A. Igarashi and N. Kobayashi, “Type-based analysis of communication for concurrent programming languages,” in *Proc. 4th International Static Analysis Symposium (SAS)*, 1997, pp. 187–201.

[25] D. Sangiorgi, “The name discipline of uniform receptiveness,” *Theoretical Computer Science*, vol. 221, no. 1-2, pp. 457–493, 1999.

[26] M. Giunti and V. T. Vasconcelos, “A linear account of session types in the pi calculus,” in *Proc. 21st International Conference on Concurrency Theory (CONCUR)*, 2010, pp. 432–446.

[27] K. D. Bowers, L. Bauer, D. Garg, F. Pfenning, and M. K. Reiter, “Consumable credentials in linear-logic-based access-control systems,” in *Proc. Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2007.

[28] D. Garg, L. Bauer, K. D. Bowers, F. Pfenning, and M. K. Reiter, “A linear logic of authorization and knowledge,” in *Proc. 11th European Symposium on Research in Computer Security (ESORICS)*, 2006, pp. 297–312.

[29] G. Lowe, “A Hierarchy of Authentication Specifications,” in *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 1997, pp. 31–44.

[30] M. Bugliesi, R. Focardi, and M. Maffei, “Compositional analysis of authentication protocols,” in *Proc. 13th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 2986. Springer-Verlag, 2004, pp. 140–154.

[31] M. Maffei, “Tags for multi-protocol authentication,” in *Proceedings of the 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SECCO)*. ENTCS, August 2004.

[32] M. Bugliesi, R. Focardi, and M. Maffei, “Analysis of typed-based analyses of authentication protocols,” in *Proc. 18th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 2005, pp. 112–125.

[33] ——, “Dynamic types for authentication,” *Journal of Computer Security*, vol. 15, no. 6, pp. 563–617, 2007.

[34] R. Focardi and M. Maffei, “Types for security protocols,” in *Formal Models and Techniques for Analyzing Security Protocols*, ser. Cryptology and Information Security Series. IOS Press, 2011, vol. 5, ch. 7, pp. 143–181.

[35] C. Fournet, A. D. Gordon, and S. Maffeis, “A type discipline for authorization policies,” *ACM Transactions on Programming Languages and Systems*, vol. 29, 2007.

[36] M. Backes, M. P. Grochulla, C. Hritcu, and M. Maffei, “Achieving security despite compromise using zero-knowledge,” in *Proc. 22nd IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, 2009, pp. 308–323.

[37] M. Backes, C. Hriţcu, and M. Maffei, “Union and intersection types for secure protocol implementations,” 2011, to appear in Theory of Security and Applications (TOSCA). Lecture Notes in Computer Science. Springer-Verlag.

[38] J. S. Hodas, “Lolli: An extension of Lambda-Prolog with linear logic context management,” in *Prolog Worshop*, 1992.

[39] J. S. Hodas and D. Miller, “Logic programming in a fragment of intuitionistic linear logic,” *Information and Computation*, vol. 110, no. 2, pp. 327–365, 1994.