Logical Foundations of Secure Resource Management in Protocol Implementations

Michele Bugliesi[1], Stefano Calzavara[1], Fabienne Eigner[2], and Matteo Maffei[2]

[1] Università Ca' Foscari Venezia
[2] Saarland University

Abstract. Recent research has shown that it is possible to leverage general-purpose theorem proving techniques to develop powerful type systems for the verification of a wide range of security properties on application code. Although successful in many respects, these type systems fall short of capturing resource-conscious properties that are crucial in large classes of modern distributed applications. In this paper, we propose the first type system that statically enforces the safety of cryptographic protocol implementations with respect to authorization policies expressed in affine logic. Our type system draws on a novel notion of "exponential serialization" of affine formulas, a general technique to protect affine formulas from the effect of duplication. This technique allows to formulate an expressive logical encoding of the authentication mechanisms underpinning distributed resource-aware authorization policies. We further devise a sound and complete type checking algorithm. We discuss the effectiveness of our approach on a case study from the world of e-commerce protocols.

1 Introduction

Verifying the security of modern distributed applications is an important and complex challenge, which has attracted the interest of a growing research community audience over the last decade. Among various static analysis approaches, security type systems have played a major role, since they are able to statically provide security proofs for an unbounded number of concurrent executions, even in presence of an active attacker; they are modular, and scale remarkably well in practice. Recent research has shown that it is possible to leverage general-purpose theorem proving techniques to develop powerful type systems for the verification of a wide range of security properties on application code, thus narrowing the gap between the formal model designed for the analysis and the actual implementation of the protocols [4,2,26]. The integration between type systems and theorem proving is achieved by resorting to a form of dependent types, known as *refinement* types. A refinement type $\{x : T \mid F(x)\}$ qualifies the structural information of the type T with a property specified by the logical formula F: a value M of this type is a value of type T such that $F(M)$ holds.

Authorization systems based on refinement types use the refinement formulas to express (and gain static control of) the credentials associated with the data and the cryptographic keys involved in the authorization checks. Clearly, the expressiveness of the resulting analysis hinges on the choice of the underlying logic, and indeed several logics have been proposed for the specification and verification of security properties [14]. A number of proposals have thus set logic *parametricity* as a design goal, to gain modularity and scalability of the resulting systems. Though parametricity is in principle a sound and wise choice, current attempts in this direction draw primarily (if not exclusively) on classical (or intuitionistic) logical frameworks. Classical logic, however, is unsuitable to express several interesting resource-aware authorization policies, such as those based on consumable credentials, or predicating over access counts and/or usage bounds [16,10]. The natural choice for expressing and reasoning about such classes of policies are instead *substructural* logics, such as linear and affine logic [17,29]. On the other hand, integrating substructural logics with existing refinement type systems for distributed authorization is challenging, as one must build safeguards against the ability of an attacker to duplicate the data exchanged over the network, and correspondingly duplicate the associated credentials, thus undermining their bounded nature [13].

Contributions. In this paper, we present an *affine refinement type system* for RCF [4], a concurrent λ-calculus which can be directly mapped to a large subset of a real functional programming language like F#. The type system guarantees that well-typed programs comply with any given authorization policy expressed in affine logic, even in the presence of an active opponent.

This type system draws on the novel concept of *exponential serialization*, a general technique to protect affine formulas from the effect of duplication. This technique makes it possible to factor the authorization-relevant invariants of the analysis out of the type system, and to characterize them directly as proof obligations for the underlying affine logical system. This leads to a rather general and modular design of the system, and sheds new light on the logical foundations of standard cryptographic patterns underpinning distributed authorization frameworks. Furthermore, the concept of serialization enhances the expressiveness of the type system, capturing programming patterns out of the scope of many substructural type systems.

The clean separation between typing and logical entailment has the additional advantage of enabling the formulation of an algorithmic version of our system, in which the non-deterministic proof search distinctive of substructural type systems can be dispensed with and expressed in terms of a single proof obligation to be discharged to an external theorem prover. This is the key to achieve an efficient implementation of our analysis technique. We prove the algorithmic version sound and complete.

We show the effectiveness of the type system on a realistic case study, namely the *EPMO* electronic purchase protocol proposed in [20]. The proof obligation generated by the type derivation for the customer code is validated by the linear logic theorem prover `llprover` [27] in less than 20 ms.

Related work. Several papers develop expressive type systems for (variants of) RCF [6,4,15,2,26] but, with the exception of F* [26], they do not support resource-aware authorization policies: in fact, even for simple linearity properties like injective agreement they rely on hand-written proofs [5]. F* [26] is a dependently typed functional language for secure distributed programming, featuring refinement types to reason about authorization policies and affine types to reason about stateful computations on affine *values*. Similarly to companion proposals for RCF, the type system of F* assumes the existence of the contraction rule in the underlying logic, hence it does not support authorization policies built over affine *formulas*. While some simple authentication patterns (e.g., basic nonce handshakes) may certainly be expressed by encoding affine predicates in terms of affine values, other more complex authentication mechanisms are much harder to handle in these terms. The *EPMO* protocol we analyze in Section 6 provides one such case, as (*i*) the nonce it employs may not be construed as an affine value because it is used twice, and (*ii*) the logical formulas justified by cryptographic message exchanges are more structured than simple predicates. Though it might be possible to come up with sophisticated encodings of these authentication mechanisms in the programming language (by resorting to, e.g., pairs of affine tokens to encode a double usage of the same nonce and special functions to eliminate logical implications), such encodings are hard to formulate in a general manner and, we argue, are much better expressed in terms of policy annotations than in some ad-hoc programming pattern.

Bhargavan et al. [7] propose a technique for the verification of F# protocol implementations by automatically extracting ProVerif models [9]. Remarkably, the framework can deal with injective agreement. On the other hand, the analysis carried out with ProVerif is not modular and has been shown less robust and scalable than type-checking [6]. Furthermore, the fragment of F# considered is rather restrictive: for instance, it does not include higher-order functions and admits only very limited uses of recursion and state.

A formal account on the integration of refinement types and substructural logics was first proposed by Mandelbaum et al. [21] with a system for local reasoning about program state built around a fragment of intuitionistic linear logic. Later, Bierhoff and Aldrich developed a framework for modular type-state checking of object-oriented programs [8,25,23]. Contrary to our proposal, none of these type systems deals with the presence of hostile (or untyped) program components, or attackers, a feature that would require fundamental changes in these systems and has deep impact on the type rules and the analysis technique.

Tov and Pucella [28] have recently shown how to use behavioral contracts to link code written in an affine language to code in a conventionally typed language. The idea is to coerce affine values to non-affine ones that can be shared with the context, but can still be reasoned about safely using dynamic access counts. There are intriguing similarities between this approach and the usage of nonces and session keys to enforce freshness in a distributed setting, which are worth investigating in the future. The two type systems are, however, fundamentally different, since our present work deals with an affine refinement logic and considers an adversarial setting, which makes a precise comparison hard to formulate.

There exist a number of types and effects systems targeted at the analysis of authenticity properties of cryptographic protocols [18,19,11]. These type systems incorporate ad-hoc mechanisms to deal with nonce handshakes and, thus, to enforce injective agreement properties. Our exponential serialization technique can be seen as a logic-based generalization of such mechanisms, independent of the language and type system. As a consequence, our type system is similarly able to verify authenticity in terms of injective agreement, while allowing for expressing also a number of more sophisticated properties involving access counts and usage bounds. As a downside, the current formulation of our type system does not allow to validate some specific nonce-handshake idioms, like the SOSH scheme [19]. Still, this can be recovered by extending our type system with union and intersection types, as shown in [2].

In a previous work [13], we made initial steps towards the design of a sound system for resource-sensitive authorization, drawing on techniques from type systems for authentication and an affine extension of existing refinement type systems for the applied pi-calculus [1]. That work aims at analyzing cryptographic protocols as opposed to their implementations. Furthermore, the type system is designed around a specific cryptographic library: the consequence is that extending the analysis to new primitives requires significant changes in the soundness proof of the type system. In contrast, the usage of RCF in this work allows us to encode cryptography in the language using a standard sealing mechanism (cf. Section 5.8), which makes the analysis technique easily extensible to new cryptographic primitives. Finally, the non-standard nature of our previous type system makes it difficult to devise an efficient algorithmic variant.

Structure of the paper. Section 2 overviews the challenges and the most important aspects of our theory on a simple example. Section 3 presents the meta-theory of exponential serialization. Section 4 reviews RCF. Section 5 outlines the type system and our treatment of formal cryptography. Section 6 presents the case study. Section 7 discusses the algorithmic version of our type system. Section 8 concludes. Due to space constraints, we refer to the long version [12] for the complete formalization of the type system and its algorithmic variant, full proofs, and a discussion on a further case study (the Kerberos protocol).

2 Overview of the Framework

We give an intuitive overview of our approach, based on a simple example of a distributed protocol involving a streaming service S and a client C that subscribes to the service and pays for watching a movie, chosen from a database of available contents.

Verifying the protocol with a refinement type system requires to first decorate the protocol with security annotations, structured as *assumptions* and *assertions*. The former introduce logical formulas which are assumed to hold at a given point (and express the credentials available at the client's side); the latter specify logical formulas which are expected to be entailed by the previously introduced assumptions (and are employed as guards for the resources at the server end). For our example, we start by assuming the authorization policy encoded by the formula below:

$$\forall x, y.(\mathsf{Paid}(x, \$1) \Rightarrow \mathsf{Watch}(x, y))$$

This is a first-order logic formula stating that each client paying one dollar can watch any movie from the database. We can then encode C and S in RCF as follows, using some standard syntactic sugar to enhance readability:

$$\begin{aligned}
C &\triangleq \lambda x_C.\, \lambda x_{addS}.\, \lambda x_m.\, \lambda x_k.\, \textsf{assume Paid}(x_C, \$1);\\
&\quad \textsf{let } x_{msg} = \textsf{sign } (x_C, x_m)\ x_k \textsf{ in send } x_{addS}\ x_{msg}\\
S &\triangleq \lambda x_S.\, \lambda x_{addS}.\, \lambda x_{vk}.\, \textsf{let } y_{msg} = \textsf{recv } x_{addS} \textsf{ in}\\
&\quad \textsf{let } (z_C, z_m) = \textsf{verify } y_{msg}\ x_{vk} \textsf{ in assert Watch}(z_C, z_m)
\end{aligned}$$

C and S are structured as functions abstracting over the parameters defined by the protocol specification. Initially, C makes the assumption $\mathsf{Paid}(x_C, \$1)$, invokes the function sign to produce a signed request for movie x_m under her private key x_k, and sends it to S on channel x_{addS}. When S receives the message, she invokes the function verify to check the signature using the public key x_{vk}, retrieves the two components of the request z_C and z_m, and asserts the formula $\mathsf{Watch}(z_C, z_m)$. Crucially, the assertion by S is done in terms of the variables z_C and z_m occurring in her code, not of the variables x_C and x_m reported in the code of C. The specification will be judged safe if for all protocol runs the assertion made at the server side is entailed by the assumption made at the client and the underlying authorization policy.

Indeed, the specification can be proved safe, but a closer look shows that the authorization policy is too liberal to enforce the expected access constraints. In fact, we have $\forall x, y.(\mathsf{Paid}(x, \$1) \Rightarrow \mathsf{Watch}(x, y)), \mathsf{Paid}(C, \$1) \vdash \mathsf{Watch}(C, m) \wedge \mathsf{Watch}(C, m')$, i.e., a single payment by C allows her to arbitrarily access the movie database for unboundedly many movies. In other words, the policy does not protect against replay attacks (to which the protocol is indeed exposed).

Affine logic for specification. As we noted earlier, the problem may be addressed by resorting to substructural logics, which capture the intended interpretation of $\mathsf{Paid}(x, \$1)$ as a consumable credential (i.e., a resource).

We focus on a simple, yet expressive, fragment of intuitionistic affine logic [29]:

$$\begin{array}{l}
F ::= A \mid F \otimes F \mid F \multimap F \mid \forall x.F \mid\ !F \mid \mathbf{0}\\
A ::= p(t_1, \ldots, t_n) \mid t = t' \quad p \text{ of arity } n \text{ in } \Sigma\\
t\ \ ::= x \mid f(t_1, \ldots, t_n) \quad f \text{ of arity } n \text{ in } \Sigma
\end{array}$$

This is the multiplicative fragment of affine logic with conjunction ($\otimes$) and implication ($\multimap$), the universal quantifier ($\forall$), the exponential modality ($!$) to express persistent truths, false ($\mathbf{0}$) to express negation, and equality. We presuppose an underlying signature Σ of predicate symbols, ranged over by p, and function symbols, ranged over by f. The set of terms, ranged over by t, is defined by variables and function symbols as expected. We mention here that RCF terms can be encoded into the logic using the locally nameless representation of syntax with binders, as shown by Bengtson et al. [4]. The true boolean predicate is written $\mathbf{1}$ and encoded as $() = ()$, where $()$ is the nullary function symbol encoding the RCF unit value. Atomic formulas, noted A in the above productions, consist of predicates and equalities.

We show some selected rules of our entailment relation in Table 1. Intuitively, proofs in affine logic must use each formula in the environment *at most* once.

$$\text{(Weak)}\ \frac{\Delta \vdash F'}{\Delta, F \vdash F'} \qquad \text{(Contr)}\ \frac{\Delta, !F, !F \vdash F'}{\Delta, !F \vdash F'} \qquad (\otimes\text{-Left})\ \frac{\Delta, F_1, F_2 \vdash F'}{\Delta, F_1 \otimes F_2 \vdash F'} \qquad (\otimes\text{-Right})\ \frac{\Delta_1 \vdash F_1 \qquad \Delta_2 \vdash F_2}{\Delta_1, \Delta_2 \vdash F_1 \otimes F_2}$$

$$(\multimap\text{-Left})\ \frac{\Delta_1 \vdash F_1 \qquad \Delta_2, F_2 \vdash F'}{\Delta_1, F_1 \multimap F_2, \Delta_2 \vdash F'} \qquad (\multimap\text{-Right})\ \frac{\Delta, F_1 \vdash F_2}{\Delta \vdash F_1 \multimap F_2} \qquad (!\text{-Right})\ \frac{\Delta \vdash F \qquad \Delta = !F_1, \ldots, !F_n}{\Delta \vdash !F}$$

Table 1. The entailment relation $\Delta \vdash F$ (selected rules)

The duplication of resources is prevented by the splitting of environments among the premises of each rule. The presence of the *weakening* rule distinguishes our relation from linear logic, in which all formulas in the environment have to be used *exactly* once in the proof.

We can then re-express the authorization policy for our example as the persistent formula: $!\,\forall x, y.(\mathsf{Paid}(x, \$1) \multimap \mathsf{Watch}(x, y))$, stating that each payment grants access to a *single* movie. In affine logic, given the environment $\forall x, y.(\mathsf{Paid}(x, \$1) \multimap \mathsf{Watch}(x, y)), \mathsf{Paid}(C, \$1)$, one can derive $\mathsf{Watch}(C, m)$ but not $\mathsf{Watch}(C, m) \otimes \mathsf{Watch}(C, m')$, since the latter derivation would require a double usage of the affine hypothesis $\mathsf{Paid}(C, \$1)$.

Affine refinement types for verification. We move on to typing the previous RCF code, to illustrate how refinement types are employed to provide a static account of the transfer of credentials required for authorization. In our example, this amounts to showing how to statically transfer the payment assumption made by C to S. That assumption is needed by S to justify (i.e., type-check) her assertion according to the underlying authorization policy; the transfer of the assumption, in turn, is achieved by giving x_k and x_{vk} suitable types.

Namely, assuming $x_c : T_1$ and $x_m : T_2$, the existing refinement type systems would give x_k type $\mathsf{SigKey}(x : T_1 * \{y : T_2 \mid \mathsf{Paid}(x, \$1)\})$, formalizing that x_k is a private key intended to sign a pair bearing the expected formula as a refinement; x_{vk}, instead, would be given the corresponding verification key type[3]. The type of x_k requires C to assume the formula $\mathsf{Paid}(x_C, \$1)$ upon signing, while the type of x_{vk} allows S to retrieve the formula $\mathsf{Paid}(z_C, \$1)$ upon verification, which is enough to entail $\mathsf{Watch}(z_C, z_m)$ and make the protocol type-check.

With affine formulas, however, such a solution deserves some special care [13], since if $\mathsf{Paid}(z_C, \$1)$ is extracted with no additional constraint by the type of x_{vk}, a replay attack mounted by an opponent could fool S into reusing the formula multiple times. We discuss next how to deal with such issues.

2.1 Exponential serialization

There are various possibilities to protect the previous protocol against replay attacks. Here, we decide to run the protocol on top of a nonce-handshake, leading

[3] In RCF we do not have any primitive notion of cryptography and, therefore, we do not have types for cryptography in our type system. We still use this notation to simplify the presentation and we discuss the encoding of these types in Section 5.8.

to the following updated RCF code:

$$\begin{aligned}
C &\triangleq \lambda x_C.\, \lambda x_{addC}.\, \lambda x_{addS}.\, \lambda x_m.\, \lambda x_k. \\
&\quad \mathsf{let}\ y_n = \mathsf{recv}\ x_{addC}\ \mathsf{in\ assume\ Paid}(x_C, \$1); \\
&\quad \mathsf{let}\ x_{msg} = \mathsf{sign}\ (x_C, x_m, y_n)\ x_k\ \mathsf{in\ send}\ x_{addS}\ x_{msg} \\
S &\triangleq \lambda x_{addS}.\, \lambda x_{addC}.\, \lambda x_{vk}.\, \mathsf{let}\ x_n = \mathsf{mkNonce}(\,)\ \mathsf{in\ send}\ x_{addC}\ x_n; \\
&\quad \mathsf{let}\ y_{msg} = \mathsf{recv}\ x_{addS}\ \mathsf{in\ let}\ (z_C, z_m, z_n) = \mathsf{verify}\ y_{msg}\ x_{vk}\ \mathsf{in} \\
&\quad \mathsf{if}\ x_n = z_n\ \mathsf{then\ assert\ Watch}(z_C, z_m) \\
\mathsf{mkNonce} &\triangleq \lambda\_ : \mathsf{unit}.\, \mathsf{let}\ x_f = \mathsf{mkFresh}()\ \mathsf{in\ assume\ N}(x_f); x_f
\end{aligned}$$

We assume to be given access to a function $\mathsf{mkFresh} : \mathsf{unit} \to \mathsf{bytes}$, which generates fresh bit-strings. The function $\mathsf{mkNonce} : \mathsf{unit} \to \{x : \mathsf{bytes} \mid \mathsf{N}(x)\}$ is a wrapper around $\mathsf{mkFresh}$, which additionally assumes the formula $\mathsf{N}(x_f)$ over the return value x_f of such a function. This new assumption is reflected by the refined return type of $\mathsf{mkNonce}$. Then, the typing of the key x_k may be structured as follows:

$$x_k : \mathsf{SigKey}(x : T_1 * y : T_2 * \{z : \mathsf{bytes} \mid !\,(\mathsf{N}(z) \multimap \mathsf{Paid}(x, \$1))\})$$

to protect the affine formula $\mathsf{Paid}(x_C, \$1)$ with the guard $\mathsf{N}(x_n)$: if $\mathsf{N}(x_n)$ can be proved only once, also $\mathsf{Paid}(x_C, \$1)$ can be extracted only once, irrespectively of the number of signature verifications performed. Remarkably, the guarded version of $\mathsf{Paid}(x_C, \$1)$ is an exponential formula, i.e., a stable truth: as such, it can be safely transmitted over the network, unaffected by replay attacks.

There is one problem left: the assumption $\mathsf{Paid}(x_C, \$1)$ available at the client C does not entail the guarded, exponential formula $!\,(\mathsf{N}(x_n) \multimap \mathsf{Paid}(x_C, \$1))$, which C needs to prove in order to use the key x_k to transmit her request. This is indeed the most intriguing bit of our construction: to construct the desired proof, we may introduce a *serializer* for $\mathsf{Paid}(x_C, \$1)$ among the assumptions of C, to automatically provide for the creation of the guarded version of $\mathsf{Paid}(x_C, \$1)$. The serializer has the form:

$$!\,\forall x, y.(\mathsf{Paid}(x, \$1) \multimap !(\mathsf{N}(y) \multimap \mathsf{Paid}(x, \$1)))$$

that is, an exponential and universally quantified formula, serving for multiple communications of different predicates built over Paid. Serializers may be generated automatically for any given affine formula, and introducing them as additional assumptions is sound, in that it does not affect the set of entailed assertions, as we discuss in the next section. Furthermore, serializers capture a rather general class of mechanisms for ensuring timely communications, like session keys or timestamps, which are all based on the consumption of an affine resource to assess the freshness of an exchange.

3 Metatheory of Exponential Serialization

In principle, the introduction of serializers among the assumed hypotheses could alter the intended semantics of the authorization policy, due to the subtle interplay of formulas through the entailment relation. Here, we isolate sufficient conditions under which exponential serialization leads to a sound protection mechanism for affine formulas.

We presuppose that the signature Σ of predicate symbols is partitioned in two sets Σ_A and Σ_C. Atomic formulas A have the form $p(t_1, \ldots, t_n)$ for some $p \in \Sigma_A$; control formulas C have the same form, though with $p \in \Sigma_C$. We identify various categories of formulas defined by the following productions.

$$\begin{array}{ll} B ::= A \mid B \otimes B \mid B \multimap B \mid \forall x.B \mid !B & \text{base formulas} \\ P ::= B \mid C \mid P \otimes P & \text{payload formulas} \\ G ::= C \multimap P \mid !G & \text{guarded formulas} \end{array}$$

Base formulas B are formulas of an authorization policy, which are used as security annotations in the application code. For simplicity, we dispense in this section with equalities and $\mathbf{0}$, since they are used in the analysis but they are never assumed in the code. (Notice that compromised principals can be modelled also without negation [4].) Payload formulas P are formulas which we want to serialize for communication over the untrusted network. Importantly, payload formulas comprise also control formulas, which allows, e.g., for the transmission of fresh nonces to remote verifiers: this pattern is present in several authentication protocols [18]. Finally, guarded formulas G are used to model the serialized version of payload formulas, suitable for transmission. We let S denote an arbitrary serializer of the form $!\forall \tilde{x}.(P \multimap !(C \multimap P))$ and we write $\Delta \vdash F^n$ for $\Delta \vdash F \otimes \ldots \otimes F$ (n times), with the proviso that $\Delta \vdash F^0$ stands for $\Delta \nvdash F$.

Given a multiset of assumptions Δ, the extension of Δ with the serializers $S_1, \ldots, S_n$ is sound if Δ and its extension derive the same payload formulas. As it turns out, this is only true when Δ satisfies additional conditions, which we formalize next.

Definition 1 (Rank). *Let $rk : \Sigma_C \to \mathbb{N}$ be a total, injective function. Given a formula F, we define the* rank *of F with respect to rk, noted $rk(F)$, as follows:*

$$\begin{array}{lll} rk(p(t_1, \ldots, t_n)) & = rk(p) & \textit{if } p \in \Sigma_C \\ rk(F_1 \otimes F_2) & = \min \{rk(F_1), rk(F_2)\} & \\ rk(F) & = +\infty & \textit{otherwise} \end{array}$$

Definition 2 (Stratification). *A formula F is* stratified *with respect to a rank function rk if and only if: (i) $F = C \multimap P$ implies $rk(C) < rk(P)$; (ii) $F = P \multimap G$ implies that G is stratified; (iii) $F = \forall x.F'$ implies that F' is stratified; (iv) $F = !F'$ implies that F' is stratified. We assume F to be stratified in all the other cases. A multiset of formulas Δ is stratified if and only if there exists a rank function rk such that each formula in Δ is stratified with respect to rk.*

For instance, the multiset $C_1 \multimap C_2, C_2 \multimap C_3$ is stratified, given an appropriate choice of a rank function, while the multiset $C_1 \multimap C_2, C_2 \multimap C_1$ is not stratified. Stratification is required precisely to disallow such circular dependencies among control formulas in the proof of our soundness result, Theorem 1 below. To prove that result, we need a further definition:

Definition 3 (Guardedness). *Let $\Delta = P_1, \ldots, P_m, S_1, \ldots, S_n$ be a stratified multiset of formulas. We say that Δ is* guarded *if and only if $\Delta \vdash C^k$ implies $k \leq 1$ for any control formula C.*

$$
\begin{array}{lll}
M, N ::= x \mid () \mid (M, N) \mid \lambda x.\, E \mid h\ M & & \text{values } (h \in \{\mathsf{inl}, \mathsf{inr}, \mathsf{fold}\}) \\
D, E ::= \ M \mid M\ N \mid M = N \mid \mathsf{let}\ x = E\ \mathsf{in}\ E' \mid & & \text{expressions} \\
\qquad \mathsf{let}\ (x, y) = M\ \mathsf{in}\ E \mid & & \\
\qquad \mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ E\ \mathsf{else}\ E' \mid & & \\
\qquad (\nu a)E \mid E \upharpoonright E' \mid a!M \mid a? \mid \mathsf{assume}\ F \mid \mathsf{assert}\ F & &
\end{array}
$$

Table 2. Syntax of RCF

The intuition underlying guardedness may be explained as follows. Consider a multiset Δ, a payload formula P such that $\Delta \vdash P$ and let $S = !\forall \tilde{x}.(P \multimap !(C \multimap P))$ be a serializer for P. Now, the only way that S may affect derivability is by allowing the duplication of the payload formula P via the exponential implication $!(C \multimap P)$. However, this effect is prevented if we are guaranteed that the control formula C guarding P is derived at most once in Δ: that is precisely what the guardedness condition ensures.

Theorem 1 (Soundness of Exponential Serialization). *Let $\Delta = P_1, \ldots, P_m$. If $\Delta' = \Delta, S_1, \ldots, S_n$ is guarded and $\Delta' \vdash P$, then $\Delta \vdash P$ for all P.*

While guardedness is convenient to use in the proof of Theorem 1, it is clearly an undecidable condition. Fortunately, it is not difficult to isolate a sufficient criterion to decide whether a multiset of formulas is guarded based on a simple syntactic check.

Proposition 1. *If $\Delta = P_1, \ldots, P_m, S_1, \ldots, S_n$ is stratified and the control formulas occurring in $P_1, \ldots, P_m$ are pairwise distinct, then Δ is guarded.*

4 Review of RCF

The syntax of values and expressions of RCF [4] is overviewed in Table 2. We assume collections of names (a, b, c, m, n) and variables (x, y, z). Values include variables, unit, pairs, functions and constructions; constructors account for the creation of standard sum types and iso-recursive types. Expressions of RCF include standard λ-calculus constructs like values, applications, equality checks, lets, pair splits, and pattern matching, as well as primitives for concurrent, message-passing computations. For space reasons, we keep the presentation intuitive and mostly informal (we refer to [4] and the long version for complete details). The semantics of expressions is standard, so we just discuss the RCF-specific constructs. Expression $(\nu a)E$ generates a fresh channel name a and then behaves as E. Expression $E \upharpoonright E'$ evaluates E and E' in parallel, and returns the result of E'. Expression $a!M$ asynchronously outputs M on channel a and returns $()$. Expression $a?$ waits until a term N is available on channel a and returns N.

Definition 4 (Safety). *A closed expression E is* safe *if and only if, in all evaluations of E, the conjunction of the asserted formulas is entailed by the introduced assumptions.*

We let an *opponent* be any closed expression of RCF which does not contain any assumption or assertion. Our goal is to guarantee that safety holds, despite the best efforts of an active opponent.

Definition 5 (Robust Safety). *A closed expression E is* robustly safe *if and only if, for any opponent O, the application $O\ E$ is safe*[4].

5 The Type System

Our refinement type system builds on previous work by Bengtson et al. [4], extending it to guarantee the correct usage of affine formulas and to enforce our revised notion of (robust) safety.

5.1 Types, typing environments, and base judgements

The syntax of types is defined as follows. The unit value is given type unit. Sum types have form $T + U$, iso-recursive types are denoted by $\mu\alpha.T$. Type variables are denoted by α. There exist various forms of dependent types: a function of type $x : T \to U$ takes as an input a value M of type T and returns a value of type $U\{M/x\}$; a pair (M, N) has type $x : T * U$ if M has type T and N has type $U\{M/x\}$; a value M has a refinement type $\{x : T \mid F\}$ if M has type T and the formula $F\{M/x\}$ holds true. We use type $\mathsf{Un} \triangleq \mathsf{unit}$ to model data that may come from, or be sent to the opponent, as it is customary for security type systems. Type $\mathsf{bool} \triangleq \mathsf{unit} + \mathsf{unit}$ is inhabited by $\mathsf{true} \triangleq \mathsf{inl}()$ and $\mathsf{false} \triangleq \mathsf{inr}()$.

Our type system comprises several typing judgements of the form $\Gamma; \Delta \vdash \mathcal{J}$, where $\Gamma; \Delta$ is a typing environment collecting all the information which can be used to derive $\mathcal{J}$. In particular, Γ contains the type bindings, while Δ comprises logical formulas that are known to hold at run-time. Formally, we let Γ be an ordered list of entries $\mu_1, \dots, \mu_n$ and Δ be a multiset of affine logic formulas. Each entry μ_i in Γ denotes either a type variable (α), a kinding annotation ($\alpha :: k$), or a type binding for channels ($a \updownarrow T$) or variables ($x : T$).

We use the judgement $\Gamma; \Delta \vdash \diamond$ to denote that the typing environment $\Gamma; \Delta$ is well-formed, i.e., it satisfies some standard syntactic conditions (for instance, it does not contain duplicate type bindings for the same variable). The only remarkable point in the definition of $\Gamma; \Delta \vdash \diamond$ is that we forbid variables in Γ to be mapped to a refinement type: indeed, when extending a typing environment with a new type binding $x : T$, we use the function ψ to place the structural type information in Γ and the function *forms* to place the associated refinements in Δ. As an example, we have $\psi(\{y : \mathsf{unit} \mid F(y)\}) = \mathsf{unit}$ and $\mathit{forms}(x : \{y : \mathsf{unit} \mid F(y)\}) = F(x)$.

Finally, we use the judgement $\Gamma; \Delta \vdash F$ to denote that the formulas in Δ entail F. The formal definition also syntactically checks that $\Gamma; \Delta$ is well-formed.

5.2 Environment rewriting

All the type information stored in Γ can be used arbitrarily often in the derivation of any judgement of our type system. The treatment of the formulas in Δ, instead, is subtler, since affine resources must be used at most once during type-checking. In particular, we need to split environment Δ among subderivations

[4] Here, the notation $O\ E$ is standard syntactic sugar for $\mathsf{let}\ x = O\ \mathsf{in}\ \mathsf{let}\ y = E\ \mathsf{in}\ x\ y$.

to avoid the duplication of resources. The general structure of the rules of our system will thus be the following:

$$\frac{\Gamma; \Delta_1 \vdash \mathcal{J}_1 \qquad \ldots \qquad \Gamma; \Delta_n \vdash \mathcal{J}_n \qquad \Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \ldots, \Delta_n}{\Gamma; \Delta \vdash \mathcal{J}}$$

where $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$ denotes the *environment rewriting* of $\Gamma; \Delta$ to $\Gamma'; \Delta'$.

The environment rewriting relation is defined as:

(Rewrite)

$$\frac{\Delta \vdash \Delta' \qquad \Gamma; \Delta \vdash \diamond \qquad \Gamma; \Delta' \vdash \diamond}{\Gamma; \Delta \hookrightarrow \Gamma; \Delta'}$$

where we write $\Delta \vdash F_1, \ldots, F_n$ to denote that $\Delta \vdash F_1 \otimes \ldots \otimes F_n$, with the proviso that $\Delta \vdash \emptyset$ stands for $\Delta \vdash \mathbf{1}$. The adoption of the environment rewriting relation as an house-keeping device for the formulas of Δ greatly improves the expressiveness of the type system in a very natural way. Interestingly, all the non-determinism introduced by the application of the rewriting rules and the splitting of the logical formulas among the premises can be effectively tamed by the algorithmic type system presented in Section 7.

5.3 Kinding and subtyping

Security type systems often rely on a kinding relation to discriminate whether or not messages of a specific type may be known to the attacker or received from it. The kinding judgement $\Gamma; \Delta \vdash T :: k$ denotes that type T is of kind k. Kind $k = \mathsf{pub}$ denotes public messages which may be sent to the attacker, while kind $k = \mathsf{tnt}$ characterizes tainted message which may come from the attacker. The type Un is both public and tainted.

The subtyping judgment $\Gamma; \Delta \vdash T <: U$ expresses the fact that T is a subtype of U and, thus, values of type T can be used in place of values of type U. The subtyping judgment makes public types subtype of tainted types and further describes standard subtyping relations for types sharing the same structure (e.g., pair types are covariant and function types contra-variant in their arguments).

Our treatment of kinding and subtyping resembles other security type systems [4,2] and only differs in the management of affine formulas, which is similar to the one we employ for typing values and expressions (see below).

5.4 Typing values

The typing judgement $\Gamma; \Delta \vdash M : T$ denotes that value M is given type T under environment $\Gamma; \Delta$. Some selected rules for assigning types to values are given in the top part of Table 3.

Rule (Val Refine) is a natural adaptation to an affine setting of the standard rule for refinement types. Rules (Val Fun) and (Val Pair) are more interesting: notice that our type system does not incorporate affine types, in that the type information in Γ is propagated to all the premises of a typing rule. It is thus crucial for soundness that both pairs and functions are type-checked in an *exponential* environment, i.e., an environment of the form $!\Delta = !F_1, \ldots, !F_n$. For instance, using an affine formula F from the typing environment to give a pair (M, N) type $x : T * \{y : U \mid F\}$ would lead to an unbounded usage of F upon replicated pair splitting operations on (M, N), as we discuss in Section 5.7. Allowing for affine refinements but forbidding affine types confines the problem of resource management to the formula environment, which simplifies the system but might seem overly restrictive. In Section 5.7 we explain how the exponential serialization technique can be leveraged to encode affine types in our framework and, thereby, enhance its expressiveness.

5.5 Typing expressions

The typing judgement $\Gamma; \Delta \vdash E : T$ denotes that expression E is given type T under environment $\Gamma; \Delta$. Some selected typing rules for expressions are given in the bottom part of Table 3.

Rule (Exp Subsum) is a standard subsumption rule for expressions. In rule (Exp Split) we exploit the logic to keep track of the performed pair splitting operation and make type-checking more precise. Rule (Exp Assert) is standard and requires an asserted formula F to be derivable from the formulas collected by the environment.

The most complex rule is (Exp Fork): intuitively, when type-checking the parallel expressions $E_1 \upharpoonleft E_2$, assumptions in E_1 can be used to type-check assertions in E_2 and vice-versa. On the other hand, we need to prevent an affine assumption in E_1 from being used twice to justify assertions in both E_2 *and* E_1. This is achieved through the *extraction* relation, i.e., through the premises of the form $E_i \rightsquigarrow [\Delta_i \mid D_i]$: the extraction operation destructively collects all the assumptions from the expression E_i and returns the expression D_i obtained by purging E_i of its assumptions. The typing environment is then extended with the collected assumptions and partitioned to type-check the purged expressions D_1 and D_2 respectively. The extraction relation is reported in Table 4. Notice that we prevent formulas containing free names from being extracted outside of the scope of the respective binders (cf. Extr Assume).

The extraction relation is also used to type-check any expression possibly containing active assumptions, i.e., lets, restrictions, and assumptions themselves.

5.6 Formal results

The main soundness result of our type system is reported below.

Theorem 2 (Robust Safety). *If $\varepsilon; \emptyset \vdash E : \mathsf{Un}$, then E is robustly safe.*

Theorem 2 above and Theorem 1 in Section 3 (establishing the soundness of exponential serialization) constitute the two building blocks of our static verification technique, which we may finally summarize as follows.

Given any expression E, we identify the payload formulas assumed in E, and construct the corresponding exponential serializers $S_1, \ldots, S_n$ for those formulas. Let then $E^\star = \mathsf{assume}\ S_1 \otimes \cdots \otimes S_n \upharpoonleft E$. By Theorem 2, if $\varepsilon; \emptyset \vdash E^\star : \mathsf{Un}$, then $E^\star$ is robustly safe. By Theorem 1, so is the original expression E, provided

(Val Var)
$$\frac{\Gamma;\Delta\vdash\diamond\qquad (x:T)\in\Gamma}{\Gamma;\Delta\vdash x:T}$$

(Val Fun)
$$\frac{\Gamma,x:\psi(T);!\Delta',\mathit{forms}(x:T)\vdash E:U\qquad \Gamma;\Delta\hookrightarrow\Gamma;!\Delta'}{\Gamma;\Delta\vdash\lambda x.\,E:x:T\to U}$$

(Val Pair)
$$\frac{\begin{array}{c}\Gamma;!\Delta_1\vdash M:T\qquad \Gamma;!\Delta_2\vdash N:U\{M/x\}\\ \Gamma;\Delta\hookrightarrow\Gamma;!\Delta_1,!\Delta_2\end{array}}{\Gamma;\Delta\vdash(M,N):x:T*U}$$

(Val Refine)
$$\frac{\begin{array}{c}\Gamma;\Delta_1\vdash M:T\qquad \Gamma;\Delta_2\vdash F\{M/x\}\\ \Gamma;\Delta\hookrightarrow\Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta\vdash M:\{x:T\mid F\}}$$

(Exp Subsum)
$$\frac{\begin{array}{c}\Gamma;\Delta_1\vdash E:T\\ \Gamma;\Delta_2\vdash T<:T'\\ \Gamma;\Delta\hookrightarrow\Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta\vdash E:T'}$$

(Exp Let)
$$\frac{\begin{array}{c}E\leadsto^{\emptyset}[\Delta'\mid E']\qquad \Gamma;\Delta_1\vdash E':T\\ \Gamma,x:\psi(T);\Delta_2,\mathit{forms}(x:T)\vdash D:U\qquad x\notin \mathit{fv}(U)\\ \Gamma;\Delta,\Delta'\hookrightarrow\Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta\vdash \mathsf{let}\ x=E\ \mathsf{in}\ D:U}$$

(Exp Split)
$$\frac{\begin{array}{c}\Gamma;\Delta_1\vdash M:x:T*U\\ \Gamma,x:\psi(T),y:\psi(U);\Delta_2,\mathit{forms}(x:T),\mathit{forms}(y:U),!((x,y)=M)\vdash E:V\\ \{x,y\}\cap \mathit{fv}(V)=\emptyset\qquad \Gamma;\Delta\hookrightarrow\Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta\vdash \mathsf{let}\ (x,y)=M\ \mathsf{in}\ E:V}$$

(Exp Assume)
$$\frac{\begin{array}{c}\Gamma;\Delta,F\vdash \mathsf{assume}\ \mathbf{1}:T\\ F\neq\mathbf{1}\end{array}}{\Gamma;\Delta\vdash \mathsf{assume}\ F:T}$$

(Exp True)
$$\frac{\Gamma;\Delta\vdash\diamond}{\Gamma;\Delta\vdash \mathsf{assume}\ \mathbf{1}:\mathsf{unit}}$$

(Exp Assert)
$$\frac{\Gamma;\Delta\vdash F}{\Gamma;\Delta\vdash \mathsf{assert}\ F:\mathsf{unit}}$$

(Exp Fork)
$$\frac{\begin{array}{c}E_1\leadsto^{\emptyset}[\Delta_1\mid D_1]\qquad E_2\leadsto^{\emptyset}[\Delta_2\mid D_2]\qquad \Gamma;\Delta_1'\vdash D_1:T_1\qquad \Gamma;\Delta_2'\vdash D_2:T_2\\ \Delta,\Delta_1,\Delta_2\hookrightarrow\Delta_1',\Delta_2'\end{array}}{\Gamma;\Delta\vdash E_1\Rsh E_2:T_2}$$

Notation: For $\Delta=F_1,\ldots,F_n$ we write $!\Delta$ to denote $!F_1,\ldots,!F_n$.

Table 3. Typing values and expressions (selected rules)

that a further invariant holds for $E^\star$, namely that all multisets of formulas assumed during the evaluation of $E^\star$ are guarded. While this latter invariant is not enforced by our type system, the desired guarantees may be achieved by requiring that the assumption of control formulas be confined within system code packaged into library functions providing certified access and management of the capabilities associated with those formulas. The certification of the system code provided by the library function, in turn, may be achieved with limited effort, based on the syntactic guardedness condition provided by Proposition 1.

5.7 Encoding affine types

Here we discuss how we can take advantage of exponential serialization to encode affine types and, thus, enhance the expressiveness of our type system. For the sake of simplicity, we focus on the encoding of affine pairs.

$$\text{(Extr Fork)}\quad \frac{E_1 \rightsquigarrow^{\widetilde{a}} [\Delta_1 \mid D_1] \qquad E_2 \rightsquigarrow^{\widetilde{a}} [\Delta_2 \mid D_2]}{E_1 \upharpoonright E_2 \rightsquigarrow^{\widetilde{a}} [\Delta_1, \Delta_2 \mid D_1 \upharpoonright D_2]}$$

$$\text{(Extr Let)}\quad \frac{E_1 \rightsquigarrow^{\widetilde{a}} [\Delta \mid D_1]}{\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \rightsquigarrow^{\widetilde{a}} [\Delta \mid \mathsf{let}\ x = D_1\ \mathsf{in}\ E_2]}$$

$$\text{(Extr Res)}\quad \frac{E \rightsquigarrow^{a,\widetilde{b}} [\Delta \mid D]}{(\nu a)E \rightsquigarrow^{\widetilde{b}} [\Delta \mid (\nu a)D]}$$

$$\text{(Extr Assume)}\quad \frac{F \neq \mathbf{1} \qquad fn(F) \cap \{\widetilde{a}\} = \emptyset}{\mathsf{assume}\ F \rightsquigarrow^{\widetilde{a}} [F \mid \mathsf{assume}\ \mathbf{1}]}$$

$$\text{(Extr Exp)}\quad \frac{\text{no other rule applies}}{E \rightsquigarrow^{\widetilde{a}} [\emptyset \mid E]}$$

Table 4. Extraction

Consider the typing environment $\Gamma; \Delta \triangleq x : \mathsf{Un}, y : \mathsf{Un}; A(x), B(y)$. Standard refinement type systems as [4] allow for the following type judgement:

$$\Gamma; \Delta \vdash (x, y) : \{x : \mathsf{Un} \mid A(x)\} * \{y : \mathsf{Un} \mid B(y)\}$$

If the formulas $A(x)$ and $B(y)$ are interpreted as affine resources, however, the previous type assignment is sound only as long as the pair (x, y) can be split only once, since every application of rule (Exp Split) for pair destruction introduces the formulas $A(x), B(y)$ into the typing environment. Since our type system does not feature affine types and has no way to enforce a single deconstruction of a pair, it conservatively forbids the previous type judgement, in that the premises of rule (Val Pair) require an exponential typing environment.

Nevertheless, the following type judgement is allowed by our type system:

$$x : \mathsf{Un}, y : \mathsf{Un}; A(x), B(y), S_1, S_2 \vdash (x, y) : \{x : \mathsf{Un} \mid A'(x)\} * \{y : \mathsf{Un} \mid B'(y)\}$$

where $A'(x) \triangleq !(P_1(x) \multimap A(x))$ and $B'(y) \triangleq !(P_2(y) \multimap B(y))$ are the serialized variants of $A(x)$ and $B(y)$ respectively, while $S_1 \triangleq !\forall x.(A(x) \multimap A'(x))$ and $S_2 \triangleq !\forall y.(B(y) \multimap B'(y))$ are the corresponding serializers. Here, the main idea for type-checking is to appeal to environment rewriting to consume the affine formulas $A(x)$ and $B(y)$, and introduce their exponential counterparts $A'(x)$ and $B'(y)$ into the environment before assigning a type to the pair components.

The interesting point now is that the pair (x, y) can be split arbitrarily often, but the affine formulas $A(x)$ and $B(y)$ can be retrieved at most once, as long as the control formulas $P_1(x)$ and $P_2(y)$ are assumed at most once in the application code. In this way, we recover the expressiveness provided by affine types. We actually even go beyond that, allowing for a liberal usage of the value itself, as opposed to enforcing the affine usage of any data structure which contains an affine component, as dictated by many earlier substructural frameworks.

5.8 Encoding cryptography

Formal cryptography can be encoded inside RCF in terms of *sealing* [22,24]. A *seal* k for a type T is a pair of functions: a sealing function $T \to \mathsf{Un}$ and an unsealing function $\mathsf{Un} \to T$. For symmetric cryptography, these functions model encryption and decryption operations, respectively. A payload of type T can be

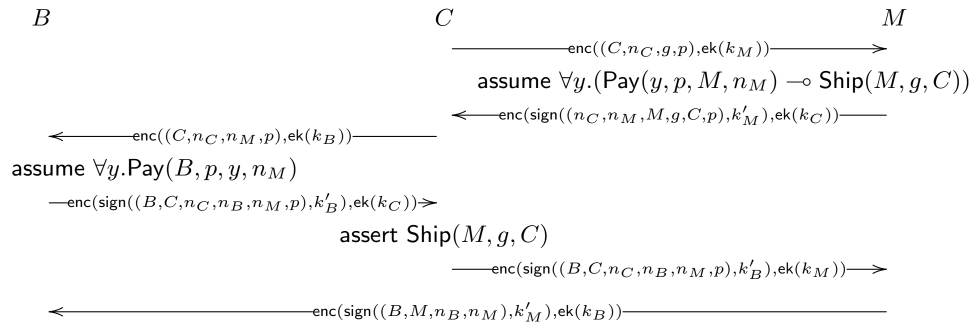
B
C
M
$\mathsf{enc}((C,n_C,g,p),\mathsf{ek}(k_M))$
$\mathsf{assume}\ \forall y.(\mathsf{Pay}(y,p,M,n_M) \multimap \mathsf{Ship}(M,g,C))$
$\mathsf{enc}(\mathsf{sign}((n_C,n_M,M,g,C,p),k'_M),\mathsf{ek}(k_C))$
$\mathsf{enc}((C,n_C,n_M,p),\mathsf{ek}(k_B))$
$\mathsf{assume}\ \forall y.\mathsf{Pay}(B,p,y,n_M)$
$\mathsf{enc}(\mathsf{sign}((B,C,n_C,n_B,n_M,p),k'_B),\mathsf{ek}(k_C))$
$\mathsf{assert}\ \mathsf{Ship}(M,g,C)$
$\mathsf{enc}(\mathsf{sign}((B,C,n_C,n_B,n_M,p),k'_B),\mathsf{ek}(k_M))$
$\mathsf{enc}(\mathsf{sign}((B,M,n_B,n_M),k'_M),\mathsf{ek}(k_B))$

Table 5. A variant of the *EPMO* protocol

sealed to type Un and sent over the untrusted network; conversely, a message retrieved from the network with type Un can be unsealed to its correct type T.

The sealing/unsealing mechanism is implemented in terms of a list of pairs, which is stored in a global reference that can only be accessed using the sealing and unsealing functions. Upon sealing, the payload is paired with a fresh, public value (the *handle*) representing its sealed version, and the pair is stored in the list; conversely, the unsealing function looks for the handle in the list and returns the associated payload. Different cryptographic primitives, like public key encryptions and signature schemes, can be encoded following such a recipe.

One interesting benefit of our exponential serialization technique is that we can directly leverage the sealing-based cryptographic library proposed by Bengtsson et al. [4]. The reason is that we never apply cryptography directly on messages with affine refinements, but we rely on their exponentially serialized variants. Without the serialization, we would need to define a different implementation of the sealing/unsealing mechanism: namely, we would have to enforce that an affine payload is never extracted more than once from the list stored in the global reference, i.e., the unsealing function would have to remove the payload from the list upon invocation. This would complicate the sealing-based abstraction of cryptography and require additional reasoning to justify its soundness.

6 Example: Electronic Purchase

We consider a variant of *EPMO*, a nonce-based e-payment protocol proposed by Guttman et al. [20]. The protocol narration is reported in Table 5.

Initially, a customer C contacts a merchant M to buy some goods g for a given price p; the request is encrypted under the public key of the merchant, $\mathsf{ek}(k_M)$, and includes a fresh nonce, n_C. If M agrees to proceed in the transaction by providing a signed response, C informs her bank B to authorize the payment. The bank replies by providing C a receipt of authorization, called the *money order*, which is then forwarded to M. Now M can verify that C is entitled to pay for the goods and complete the transaction by sending a signed request to B to cash the money order. At the end of the run, the bank transfers the funds and the merchant ships the goods.

A peculiarity of the protocol is that the identifier n_C is employed by C to authenticate *two* different messages, namely the replies by M and B. This pattern cannot be validated by most existing type systems, since the mechanisms hardcoded therein to deal with nonce-handshakes enforce the freshness of each nonce to be checked only once. Our framework, instead, allows for a very natural treatment of such authentication pattern, whose implementation can be written mostly oblivious of the security verification process based on lightweight logical annotations. For space reasons, we focus only on the aspects of the verification connected to the guarantees provided to C.

We define two predicates used in the analysis: $\mathsf{Pay}(B, p, M, n_M)$ states that B authorizes the payment p to M in reference to the order identified by n_M, while $\mathsf{Ship}(M, g, C)$ formalizes that M can ship the goods g to C. The protocol code for the customer, enriched with the most interesting type annotations, is reported below[5].

```
type MsgMC = MsgMC of (xnC: Un * xnM: Un * xM: Un * xg: Un * xC: Un * xp: Un)
  {!(N1(xnC) --o forall y.(Pay(y,xp,xM,xnM) --o Ship(xM,xg,xC))}

type MsgBC = MsgBC of (yB: Un * yC: Un * ynC: Un * ynB: Un * ynM: Un * yp: Un)
  {!(N2(ynC) --o forall y.(Pay(yB, yp, y, ynM))}

let (mkTid : unit -> {x: bytes | N1(x) times N2(x)}) () =
  let xf = mkFresh () in assume (N1(xf) times N2(xf)); xf

let cust C addC M addM B addB g p kC ekM ekB
         (vkM: (MsgMC, MsgMB) either VerKey) (vkB: MsgBC VerKey) =
  let nC = mkTid () in
  let msgCM1 = encrypt (C, nC, g, p) ekM in send addM msgCM1;
  let signMC = decrypt (receive addC) kC in
  let plainMC = verify signMC vkM in
  match plainMC with MsgMC (=nC, xnM, =M, =g, =C, =p) ->
      let msgCB = encrypt (C, nC, xnM, p) ekB in send addB msgCB;
      let signBC = decrypt (receive addC) kC in
      let plainBC = verify signBC vkB in
        match plainBC with MsgBC (=B, =C, =nC, xnB, =xnM, =p) ->
          assert Ship(M, g, C);
          let msgCM2 = encrypt signBC ekM in send addM msgCM2
```

Initially, we let the customer call the library function mkTid, which generates a fresh transaction identifier, corresponding to n_C in the protocol specification, and provides via its return type two distinct capabilities $\mathsf{N}_1(n_C)$ and $\mathsf{N}_2(n_C)$, later employed to authenticate two different messages received by C. Since the signing key of M is used to certify messages of two different types, at steps 2 and 6 of the protocol, the corresponding verification key available to the customer through the variable vkM refers to a sum type. We present only the MsgMC component of such type, since it is the one needed to type-check the code of C: the corresponding refined formula in the type definition describes the promise by M to ship the goods as soon as the requested payment has been authorized by any bank. We then use vkB to convey the other formula which is needed to type-check C, namely a statement that B authorizes the payment to any merchant to whom C wishes to transfer the money order. The hypotheses collected by C are enough to prove her assertion, i.e., to be sure that the request by M has been fulfilled and the goods will be shipped, hence the implementation is well-typed.

[5] For the sake of readability, we use F#- like syntax and some syntactic sugar like tuples and pattern matching to present code snippets from our example: these can be encoded in RCF using standard techniques [4].

7 Algorithmic Typing

The type system presented in Section 5 includes several non-deterministic rules, which make it hard to implement an efficient decision procedure. In this section, we present an algorithmic version, which we prove sound and complete.

7.1 Algorithmic type system

While standard sources of non-determinism like subtyping or refining value types can be eliminated using type annotations, the rewriting of logical environments, the distinctive source of non-determinism of our system, is harder to deal with. The core idea underlying the algorithmic version of the type system is to dispense with logical environments and to construct bottom-up a single logical formula that characterizes all the proof obligations that would normally be introduced along the type derivation. More in detail, every typing judgment of the form $\Gamma; \Delta \vdash \mathcal{J}$ is matched by an algorithmic counterpart of the form $\Gamma \vdash_{\mathsf{alg}} \mathcal{J}; F$. Intuitively, typing an expression algorithmically constitutes of two steps:

1. The expression (decorated with type annotations whenever needed) is type-checked using the algorithmic type system. This process is fully deterministic and in case of success yields *one* proof obligation F.
2. The proof obligation is verified, e.g., using an external theorem prover.

If both steps succeed, then the expression is well-typed.

In the remainder of this section we focus on selected rules for typing values and expressions: the remaining rules follow along the same lines.

7.2 Typing values and expressions

We present some selected algorithmic typing rules in Table 6.

Following standard practice, we rely on typing annotations to deal with non-structural rules. For instance, we explicitly annotate values that are expected to be given a refinement type (cf. (Val Ref (Alg)) and expressions whose type should be derived using subtyping (cf. Exp Subsum (Alg)). In this way, every possible syntactic form is matched exactly by a single type rule.

We now exemplify the general concepts underlying our technique by contrasting the standard typing rule (Val Fun) with its algorithmic counterpart (Val Fun (Alg)). The main source of non-determinism in (Val Fun) is the rewriting of Δ to $!\Delta'$. As previously mentioned, our goal is to dispense with logical environments and their rewriting, by collecting a single proof obligation that accounts for the proof obligations generated in the original type system. In the algorithmic version, the proof obligation obtained by giving $\lambda x : T.\, E$ type $V := x : T \to U$ in Γ is $!\forall x.(\mathit{forms}(x : T) \multimap F')$, where F' is the proof obligation collected by giving E type U in $\Gamma, x : \psi(T)$. In the following, we briefly justify why this approach is sound, i.e., we argue why $\Gamma; \Delta \vdash \lambda x.\, E : V$ for any Δ such that $\Gamma; \Delta \vdash\, !\forall x.(\mathit{forms}(x : T) \multimap F')$ (i.e., Δ entails $!\forall x.(\mathit{forms}(x : T) \multimap F')$ and both are well-formed with respect to Γ). From $\Gamma; \Delta \vdash\, !\forall x.(\mathit{forms}(x : T) \multimap F')$, using the

(Val Var (Alg))
$$\frac{\Gamma \vdash_{\mathsf{alg}} \diamond \qquad (x : T) \in \Gamma}{\Gamma \vdash_{\mathsf{alg}} x : T; \mathbf{1}}$$

(Val Fun (Alg))
$$\frac{\Gamma, x : \psi(T) \vdash_{\mathsf{alg}} E : U; F' \qquad \mathit{fnfv}(T) \subseteq \mathit{dom}(\Gamma) \cup \{x\}}{\Gamma \vdash_{\mathsf{alg}} \lambda x : T.\, E : (x : T \to U); !\forall x.(\mathit{forms}(x : T) \multimap F')}$$

(Val Pair (Alg))
$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{alg}} M : T; F_1 \\ \Gamma \vdash_{\mathsf{alg}} N : U\{M/x\}; F_2\end{array}}{\Gamma \vdash_{\mathsf{alg}} (M, N) : x : T * U; !F_1 \otimes !F_2}$$

(Val Ref (Alg))
$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{alg}} M : T; F' \\ \mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma) \cup \{x\}\end{array}}{\Gamma \vdash_{\mathsf{alg}} M_{\{x:\_ \ \mid \ F\}} : \{x : T \mid F\}; F' \otimes F\{M/x\}}$$

(Exp Subsum (Alg))
$$\frac{\Gamma \vdash_{\mathsf{alg}} E : T; F_1 \qquad \Gamma \vdash_{\mathsf{alg}} T <: T'; F_2}{\Gamma \vdash_{\mathsf{alg}} E_{\_<:T'} : T'; F_1 \otimes F_2}$$

(Exp Let (Alg))
$$\frac{\begin{array}{c}E \rightsquigarrow^{\emptyset} [\Delta' \mid E'] \\ \Gamma \vdash_{\mathsf{alg}} E' : T; F_1 \qquad \Gamma, x : \psi(T) \vdash_{\mathsf{alg}} D : U; F_2 \qquad x \notin \mathit{fv}(U) \qquad \mathit{fnfv}(\Delta') \subseteq \mathit{dom}(\Gamma)\end{array}}{\Gamma \vdash_{\mathsf{alg}} \mathsf{let}\ x = E\ \mathsf{in}\ D : U; \Delta' \multimap (F_1 \otimes \forall x.(\mathit{forms}(x : T) \multimap F_2))}$$

Notation: In logical formulas, we write $F_1, \ldots, F_n$ to denote $F_1 \otimes \ldots \otimes F_n$.

Table 6. Selected algorithmic rules for typing values and expressions

rules of the logic, we can show that there exists Δ' such that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$ and $\Gamma; !\Delta' \vdash \forall x.\mathit{forms}(x : T) \multimap F'$. Intuitively, this means that we can eliminate the exponential modality by rewriting the logical environment in exponential form. Furthermore, the well-formedness of $\Gamma; !\Delta'$ ensures that $x \notin \mathit{fv}(!\Delta')$: in this case, we can further eliminate the universal quantification, adding a type binding for x in order to keep the logical environment well-formed (the actual type is not relevant from the logic point of view), i.e., $\Gamma, x : \psi(T); !\Delta' \vdash \mathit{forms}(x : T) \multimap F'$. Using rule ($\multimap$-Left), we can finally prove $\Gamma, x : \psi(T); !\Delta', \mathit{forms}(x : T) \vdash F'$. By inductive reasoning, $\Gamma, x : \psi(T); !\Delta', \mathit{forms}(x : T) \vdash E : U$. Finally, (Val Fun) allows us to derive $\Gamma; \Delta \vdash \lambda x.\, E : V$. The algorithmic variant is similarly proved complete.

If a typing rule contains multiple premises, then we combine the proof obligations obtained from the premises conjunctively (cf. Val Pair (Alg)). Whenever a typing rule relies on extraction (e.g., Exp Let) and adds the extracted environment Δ' to the environment before rewriting, the algorithmic variant of the rule (e.g., Exp Let (Alg)) creates a proof obligation of the form $\Delta' \multimap F$, where F is the proof obligation obtained by combining the proof obligations of the premises using the techniques described above.

7.3 Main results

Let $\langle E \rangle$ denote the expression obtained from E by erasing all typing annotations.

Theorem 3 (Soundness and Completeness of Algorithmic Typing).

1. *If $\Gamma \vdash_{\mathsf{alg}} E : T; F$ and $\Gamma; \Delta \vdash F$, then $\Gamma; \Delta \vdash \langle E \rangle : T$.*

2. If $\Gamma;\Delta \vdash E : T$, then there exists E', F such that $\langle E' \rangle = E$, $\Gamma \vdash_{\mathsf{alg}} E' : T; F$, and $\Gamma;\Delta \vdash F$.

7.4 Typing the example

The proof obligation assigned to the `cust` function in Section 6 is shown below.

$$
\begin{array}{l}
\forall \mathtt{C}.\forall \mathtt{M}.\forall \mathtt{B}.\forall \mathtt{G}.\forall \mathtt{p}.\\
\quad \forall \mathtt{nC}.((\mathtt{N1(nC)} \otimes \mathtt{N2(nC)}) \multimap\\
\qquad \forall \mathtt{xnM}.(!(\mathtt{N1(nC)} \multimap (\forall \mathtt{y}.\mathtt{Pay(y,p,M,xnM)} \multimap \mathtt{Ship(M,g,C)})) \multimap\\
\qquad\qquad !(\mathtt{N2(nC)} \multimap (\forall \mathtt{z}.\mathtt{Pay(B,p,z,xnM)})) \multimap\\
\qquad\qquad \mathtt{Ship(M,g,C)}))
\end{array}
$$

For the sake of readability we removed all unnecessary occurrences of **1** and unused quantified variables.

In this example, as well as in all other protocols we considered, the problem of solving equalities is reduced to the unification of variables[6]. This allows us to use the `llprover` [27] theorem prover, which at the moment does not support equality theories. The above formula is discharged in less than 20 ms.

8 Conclusion

We presented the first type system for statically enforcing the (robust) safety of cryptographic protocol implementations with respect to authorization policies expressed in affine logic. Our type system benefits from the novel concept of exponential serialization to achieve a general and flexible treatment of affine resources. We further proposed an efficient algorithmic variant of the type system.

We are currently working on the mechanization of our theory by implementing a type-checker based on the algorithmic typing rules. We plan to facilitate type-checking and reduce the need for manual type annotations by taking advantage of recent research on type inference in intuitionistic linear logic [3].

References

1. 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 (2001)
2. Backes, M., Hriţcu, C., Maffei, M.: Union and Intersection Types for Secure Protocol Implementations. In: TOSCA'11. pp. 1–28. LNCS, Springer (2011)
3. Baillot, P., Hofmann, M.: Type Inference in Intuitionistic Linear Logic. In: PPDP'10. pp. 219–230. ACM (2010)
4. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement Types for Secure Implementations. TOPLAS 33(2), 8 (2011)
5. Bhargavan, K., Corin, R., Deniélou, P.M., Fournet, C., Leifer, J.J.: Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. In: CSF'09. pp. 124–140. IEEE (2009)

[6] Equalities are introduced by pattern-matching, a syntactic sugar which we encode in our system using standard techniques [4].

6. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular Verification of Security Protocol Code by Typing. In: POPL'10. pp. 445–456. ACM (2010)
7. Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified Interoperable Implementations of Security Protocols. TOPLAS 31(1) (2008)
8. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA'07. pp. 301–320. ACM (2007)
9. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: CSFW'01. pp. 82–96. IEEE (2001)
10. Bowers, K.D., Bauer, L., Garg, D., Pfenning, F., Reiter, M.K.: Consumable Credentials in Linear-Logic-Based Access-Control Systems. In: NDSS'07. Internet Society (2007)
11. Bugliesi, M., Focardi, R., Maffei, M.: Dynamic Types for Authentication. JCS 15(6), 563–617 (2007)
12. Bugliesi, M., Calzavara, S., Eigner, F., Maffei, M.: Logical Foundations of Secure Resource Management in Protocol Implementations (Long Version), `http://www.lbs.cs.uni-saarland.de/affine-rcf/`
13. Bugliesi, M., Calzavara, S., Eigner, F., Maffei, M.: Resource-Aware Authorization Policies for Statically Typed Cryptographic Protocols. In: CSF'11. pp. 83–98. IEEE (2011)
14. Chapin, P.C., Skalka, C., Wang, X.S.: Authorization in Trust Management: Features and Foundations. ACM Computing Surveys 40(3) (2008)
15. Fournet, C., Kohlweiss, M., Strub, P.Y.: Modular Code-Based Cryptographic Verification. In: CCS'11. pp. 341–350. ACM (2011)
16. Garg, D., Bauer, L., Bowers, K.D., Pfenning, F., Reiter, M.K.: A Linear Logic of Authorization and Knowledge. In: ESORICS'06. pp. 297–312. LNCS, Springer (2006)
17. Girard, J.Y.: Linear Logic: Its Syntax and Semantics. In: Advances in Linear Logic. London Mathematical Society LNS, vol. 22, pp. 1–42. Cambridge University Press (1995)
18. Gordon, A.D., Jeffrey, A.: Authenticity by Typing for Security Protocols. JCS 11(4), 451–519 (2003)
19. Gordon, A.D., Jeffrey, A.: Types and Effects for Asymmetric Cryptographic Protocols. JCS 12(3), 435–484 (2004)
20. Guttman, J.D., Thayer, F.J., Carlson, J.A., Herzog, J.C., Ramsdell, J.D., Sniffen, B.T.: Trust Management in Strand Spaces: A Rely-Guarantee Method. In: ESOP'04. pp. 325–339. LNCS, Springer (2004)
21. Mandelbaum, Y., Walker, D., Harper, R.: An effective theory of type refinements. In: ICFP'03. pp. 213–225. ACM (2003)
22. Morris, J.: Protection in Programming Languages. CACM 16(1), 15–21 (1973)
23. Naden, K., Bocchino, R., Aldrich, J., Bierhoff, K.: A Type System for Borrowing Permissions. In: POPL'12. pp. 557–570. ACM (2012)
24. Sumii, E., Pierce, B.: A Bisimulation for Dynamic Sealing. TCS 375(1-3), 169–192 (2007)
25. Sunshine, J., Naden, K., Stork, S., Aldrich, J., Tanter, E.: First-Class State Change in Plaid. In: OOPSLA'11. pp. 713–732. ACM (2011)
26. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure Distributed Programming with Value-Dependent Types. In: ICFP'11. pp. 266–278. ACM (2011)
27. Tomura, N.: llprover - A Linear Logic Prover, `http://bach.istc.kobe-u.ac.jp/llprover/`
28. Tov, J., Pucella, R.: Stateful Contracts for Affine Types. In: ESOP'10, pp. 550–569. LNCS, Springer (2010)
29. Troelstra, A.S.: Lectures on Linear Logic. CSLI Stanford, LNS, vol. 29 (1992)