Pattern-Matching Spi-Calculus[*]

Christian Haack
Security of Systems Group
Faculty of Science, Radboud University
Postbus 9010, 6500 GL Nijmegen, The Netherlands
chaack@cs.ru.nl

Alan Jeffrey
Security Technology Research Department
Bell Labs, Lucent Technologies
2701 Lucent Lane, Room 9F-934, Lisle, IL 60532, USA
ajeffrey@bell-labs.com

May 12, 2006

Abstract

Cryptographic protocols often make use of nested cryptographic primitives, for example signed message digests, or encrypted signed messages. Gordon and Jeffrey's prior work on types for authenticity did not allow for such nested cryptography. In this work, we present the *pattern-matching spi-calculus*, which is an obvious extension of the spi-calculus to include pattern-matching as primitive. The novelty of the language is in the accompanying type system, which uses the same language of patterns to describe complex data dependencies which cannot be described using prior type systems. We show that any appropriately typed process is guaranteed to satisfy robust authenticity, secrecy and integrity properties.

1 Introduction

Background. Cryptographic protocols are prone to subtle errors, in spite of the fact that they are often relatively small, and so are a suitable target for formal and automated verification methods. One line of such research is the development of domain-specific languages and logics, such as BAN logic [15], strand spaces [39], CSP [37, 38], MSR [17] and the spi-calculus [7]. These languages are based on the Dolev–Yao model of cryptography [19], and often use Woo and Lam's correspondence assertions [40] to model authenticity. Techniques for proving correctness include rank functions [38, 30, 28], theorem provers [13, 36, 18], model checkers [31, 34] and type systems [1, 3, 16, 24, 25, 23].

Protocol verification by type-checking. Verification tools for cryptographic protocols range from fully automatic tools, for instance based on model checking (e.g. Casper [32]) or automatic theorem proving (e.g. ProVerif [11]), to interactive theorem provers (e.g. Isabelle [36]). The obvious advantage of fully automatic tools is that they do not require any human help beyond the specification of the protocol and its security properties. On the downside, such tools have theoretical limitations (see for instance [20, 9]). To deal with these

[*]This material is based upon work supported by the National Science Foundation under Grant No. 0208459. The article extends a shorter paper that was presented at the IFIP workshop on Formal Aspects of Security and Trust in Toulouse, August 2004.

limitations, they often restrict the Dolev–Yao model (for instance by assuming that each principal only runs a small number of sessions) or are not guaranteed to terminate. Interactive theorem provers, on the other hand, are not limited in this way but need much human help. In terms of the required human help, verification by type-checking is somewhere in between fully automatic verification and interactive theorem proving. By annotating variables and names with their types, the protocol specifier gives some hints to the automatic type-checker. Given these type annotations, the type-checker can then automatically (and quickly) construct a robust safety proof. Type-checking does not require any restrictions on the Dolev–Yao model. In particular, we can verify protocols even if each principal is allowed to run arbitrarily many sessions.

Gordon and Jeffrey's type systems for authenticity [24, 25, 23] use types to enforce crucial engineering principles for cryptographic protocols [8]. For instance, these type systems use dependent types to enforce the principle of always including all principal identities that are important for the semantics of a message (principle 3 from [8]). Or they use tagged union types to enforce that receivers can always tell the number of a message within a protocol run (principle 10 in [8]) in order to avoid type flaw attacks [29]. Gordon and Jeffrey prove that the engineering principles enforced by their type system are enough to guarantee robust safety of protocols. In other words, their type systems are sound. On the other hand, their type systems (like our type system for pattern-matching spi) are not complete. There are protocols that are robustly safe but do not type-check. Our type system for pattern-matching spi builds on Gordon and Jeffrey's earlier systems. It improves on them in so far as it allows the verification of additional protocols that could not be verified by the earlier type systems.

Towards more complete and realistic cryptographic type systems. Type systems for interesting languages are often incomplete, that is they fail to type-check some safe programs. Type systems usually are tailored to a particular idiom, for example [3] treats public encryption keys but not signing keys, and [25] covers full symmetric and asymmetric cryptography but not nested uses of cryptography. In this paper, we extend techniques from [24, 25, 23] to, in addition, reason about protocols making use of *nested cryptography*, *hashing* and *message authentication codes*. We also present new typing constructs for protocol-independent key types, which permit *reusable long-term keys*.

Nested cryptography. The process language of pattern-matching spi combines the suite of separate message destructors and equality checks from previous systems [24, 25, 23] into one pattern-matching construct. Patterns at the process level are convenient, and are similar to the communication techniques used in other specification languages [39, 17, 12]. Notably, our system uses patterns not only in processes but allows *patterns in types*, too. This permits types for nested use of cryptographic primitives, which would otherwise not be possible. For example, previous type systems [24, 25, 23] could express data dependencies such as

$$(\exists a : \mathsf{Princ}, \exists m : \mathsf{Msg}, \exists b : \mathsf{Princ}, [\,!\mathsf{begun}(a,b,m)])$$

where $!\mathsf{begun}(a,b,m)$ is an *effect* ensuring that principals a and b have agreed on message m. In this paper, we extend these systems to deal with more complex data dependencies such as

$$(\{\!|\#(\ell(\exists m : \mathsf{Msg}, \exists x))|\!\}_{\exists y^{-1}}, \exists a : \mathsf{Princ}, \exists b : \mathsf{Princ})[!\mathsf{begun}(m,a,b)]$$

where the effect $!\mathsf{begun}(a,b,m)$ makes use of a variable m, which is triply nested in the scope of a decryption $\{\!|\cdot|\!\}_{\exists y^{-1}}$, a hash function $\#(\cdot)$ and a message tag $\ell(\cdot, \exists x)$: such data dependencies were not previously allowed because the occurrence m in $!\mathsf{begun}(a,b,m)$ would be considered out of scope.

A form of nested cryptography are *sign-then-encrypt protocols*, where principal A digitally signs a secret message M for B and then encrypts it with B's public encryption key, resulting in the ciphertext $\{\!|\{\!|\ell(M,B)|\!\}_{esA}|\!\}_{epB}$. Sign-then-encrypt protocols were not typable in [25], because the type system permitted signing keys to only encrypt public messages. In order to allow such protocols, we refined the typing rules for encryption and decryption. Whereas in [25] encryption always results in ciphertexts of public types, in pattern-matching spi ciphertext types keep track of secrecy levels of encrypted plaintexts. For instance, if M's type is Secret then $\{\!|\ell(M,B)|\!\}_{esA}$'s type typically is $\mathsf{Secret}(B)$, meaning that it must be kept secret and may only be published after encryption with B's public key. In contrast, if M's type is Public then $\{\!|\ell(M,B)|\!\}_{esA}$'s type typically is Public, meaning that it may be published as is.

Another form of nested cryptography arises by *nesting of digital signatures*. If, for instance, principal D receives a message $\{\!|\{\!|\{\!|\ell(M,D)|\!\}_{esA}|\!\}_{esB}|\!\}_{esC}$ that has been digitally signed by principals A, B and C, then D knows that M has been authenticated by these three principals. This information was not expressible in the type system from [25]. In pattern-matching spi, the tagged message $\ell(M,D)$ may have an *authentication type* $\mathsf{Public}\langle A,B,C\rangle$ or $\mathsf{Secret}\langle A,B,C\rangle$. A message of this type requires authentication of principals A, B and C. This authentication can be acquired by nested digital signatures with signing keys of types $\mathsf{SigningEK}(A)$, $\mathsf{SigningEK}(B)$ and $\mathsf{SigningEK}(C)$. When principal D receives a message $\ell(M,D)$ of type $\mathsf{Public}\langle A,B,C\rangle$ (resp. $\mathsf{Secret}\langle A,B,C\rangle$), then he knows that it has been authenticated by A, B and C.

Somewhat related to nested digital signatures are *nonce challenges with multiple responders.* In challenge/response protocols, a single nonce challenge can have multiple responders resulting in authentication of each of the responders. Suppose, for instance, that A sends a nonce challenge to B, who forwards it together with additional data to a server S encrypted under a shared symmetric server key kBS, who then sends the nonce back to A together with additional data encrypted under the shared symmetric server key kAS. In such a situation, both B and S authenticate to A in reply to a single nonce challenge. In this paper, we have refined the nonce rules from [25] to allow authentication of multiple responders to a single nonce challenge.

Reusable long-term keys: tag types instead of tagged union types. Whereas in [25] a signing key k has type $\mathsf{EncryptKey}(T)$, where T is a tagged union type enumerating all message formats that k may sign, in pattern-matching spi Alice's signing key has type $\mathsf{SigningEK}(\mathrm{Alice})$. Our type system still enforces message tagging for avoiding type flaw attacks but, in contrast to the earlier systems, this is achieved by assigning types to tags instead of associating keys with tagged union types. Tag types are useful for composing two protocols that make use of the same key. This is safe provided each protocol requires ciphertexts to include a tag that identifies the protocol. Our type system enforces such tagging. It is, of course, also still possible to type-check single protocols that use the same key to encrypt different messages formats (provided that these messages are properly tagged).

Small core language. While increasing the completeness of a cryptographic type system, it is important to keep the system tractable, so that rigorous safety proofs are still feasible. For that reason, we chose to define a small core language and obtain the full language through derived forms. The core language is extremely parsimonious: its only constructs for messages are tupling, asymmetric encryption and those for asymmetric keys.We show that symmetric encryption, hashing, keyed hashing and message tagging can all be obtained by simple syntactic translations into this small core and that these translations yield sensible typing rules.

New type system architecture. Our primitive pattern-matching input operation, which replaces explicit message destructors from [24, 25, 23], lead us to a technically rather different type system architecture. Our type system is less syntax directed than the earlier systems, as it includes a set of typing rules that operate on type environments. These *left rules* are used to implicitly destruct messages that have been received as input. In contrast to the earlier typing rules for explicit message destructors, which operate at the process level, our left rules are message level rules. We have also replaced explicit syntactic operators for nonce casting and nonce checking by implicit typing rules. This new architecture provides a bit more flexibility, which turns out to be very useful for elegantly dealing with nested cryptography and hashing.

Authenticity, secrecy and integrity. We formalize authenticity in the same way as Gordon and Jeffrey in their earlier papers using Woo and Lam's correspondence assertions, and show that well-typed processes with public free names are robustly safe for authenticity. (In our terminology, *robust* safety means safety in the presence of adversaries.) In addition, we prove robust safety theorems for secrecy ("robust write-safety") and integrity ("robust read-safety"). Secrecy and integrity are formalized using the language of types, and the robust write- and read-safety results formally confirm the informal semantics of public and tainted types, as introduced in [25]. All three robust safety results are corollaries of the same type preservation theorem, which essentially states that the operational semantics preserves typings.

Outline. Section 2 presents an introductory example. Section 3 defines the syntax for messages, patterns and processes, and reviews the technique of specifying authenticity by correspondence assertions from [24, 25, 23, 40]. Section 4 explains the type system for the core language. Section 5 presents derived forms for key types, symmetric cryptography, message tagging, hashing and keyed hashing, and illustrates their use in simple examples. In Section 6, we analyze two standard protocols with our type system. In Section 7, we prove that the type system is robustly safe. Many proof details are postponed to the appendix. We conclude in Section 8 with a comparison with related work, a summary and some ideas on future work.

The examples omit complete type derivations, because these are tedious to do by hand. We have type-checked all examples from this paper (and many more) using our automatic type-checker [22], which is available on the web. The technical development, including proofs, is contained in Appendix B.

Acknowledgments. We thank Andy Gordon, Radha Jagadeesan, Corin Pitcher and James Riely for comments and discussions on this work at various stages of its development. Moreover, we thank the referees for their critique and numerous valuable suggestions how to improve the presentation.

Notational conventions. If the meta-variable x ranges over set S, then $\vec{x}$ ranges over finite sequences over S, and $\bar{x}$ ranges over finite multisets over S. We sometimes write sequences $\vec{x}$ in contexts that require multisets or sets, implicitly coercing by ignoring order and multiplicities.

2 An Introductory Example

Before the technical exposition, we want to convey a flavor of the type system by discussing a simple example. Consider the following simple sign-then-encrypt protocol:

$$\begin{array}{ll} A \text{ begins! } (M,A,B) & \\ A \to B & \{\!\!|\{\!\!|sec(M,B)|\!\!\}_{esA}|\!\!\}_{epB} \\ B \text{ ends } (M,A,B) & \end{array}$$

Alice wants to send Bob a secret message M. To this end, she first tags M together with Bob's name with a tag sec. Tagging is a prudent engineering practice for avoiding confusion between different protocol messages encrypted by the same key. Our type system enforces this practice and types of tags communicate important type information. Tagging is actually a derived construct defined as encryption with a key extracted from a public key pair: $\ell(M) \stackrel{\Delta}{=} \{\!\!|M|\!\!\}_{\mathsf{Enc}(\ell)}$ where the tag ℓ is a global, public name. Next, Alice encrypts the tagged message $sec(M,B)$ with her secret signing key esA to authenticate herself. Because M is to be kept secret, Alice finally encrypts the message with Bob's public encryption key. The begin- and end-statements are Woo-Lam correspondence assertions [40]. They specify that Alice begins a protocol session (M,A,B), which Bob ends after message reception.

Protocol specification in pattern-matching spi. Here are Alice's and Bob's side of this protocol expressed in pattern-matching spi-calculus:

$$\begin{array}{lcl} P_A & \stackrel{\Delta}{=} & \mathsf{new}\ m : \mathsf{Secret};\ \mathsf{begin!}(m,A,B);\ \mathsf{out}\ net\ \{\!\!|\{\!\!|sec(m,B)|\!\!\}_{esA}|\!\!\}_{epB} \\ P_B & \stackrel{\Delta}{=} & \mathsf{inp}\ net\ \{\!\!|\{\!\!|sec(\exists x,B)|\!\!\}_{dsA^{-1}}|\!\!\}_{dpB^{-1}};\ \mathsf{end}(x,A,B) \end{array}$$

The variable net represents an untrusted channel and dsA and dpB are the matching decryption keys for esA and epB. An output statement of the form $(\mathsf{out}\ net\ N)$ sends a message N out on channel net. A statement of the form $(\mathsf{inp}\ net\ X; P)$ attempts to input from channel *net* a message that matches pattern X. Existentials in patterns indicate which variables get bound as part of the pattern match. In the input pattern above, the variable x gets bound, whereas B, dsA and dpB are constants that must be matched exactly.

Type annotations. For a type-checker to verify the protocol's correctness (and also for us to better understand and document it), it is necessary that we annotate the protocol with types. For our example, the types for the free variables are:

$net : \mathsf{Un}$	*net* is an untrusted channel
$M : \mathsf{Secret}$	M will not be revealed to the opponent
$epB : \mathsf{PublicCryptoEK}(B)$	*epB* is B's public encryption key
$dpB : \mathsf{PublicCryptoDK}(B)$	*dpB* is B's matching decryption key
$esA : \mathsf{SigningEK}(A)$	*esA* is A's private signing key
$dsA : \mathsf{SigningDK}(A)$	*dsA* is A's matching signature verification key

No type annotations are necessary in P_A, because P_A does not have input statements. In P_B we add two type annotations. The input variable x is annotated with Secret. Moreover, we add an assertion $!\mathsf{begun}(x,A,B)$ to the input statement, meaning that a (x,A,B)-session has previously begun. The operational semantics ignores this assertion. However, the type-checker statically ensures that the assertion is met whenever a message matches the input pattern. Using this assertion as a precondition for the process continuation, the type-checker can verify that it is safe to end an (x,A,B)-session. Here is the annotated version of P_B:

$$P_B \ \stackrel{\Delta}{=}\ \mathsf{inp}\ net\ \{\!|\{\!|sec(\exists x : \mathsf{Secret}, B)|\!\}_{dsA^{-1}}|\!\}_{dpB^{-1}}[!\mathsf{begun}(x,A,B)];\ \mathsf{end}(x,A,B)$$

We have yet omitted the type of the tag *sec*, which gives the type-checker crucial hints for verifying the input assertion $!\mathsf{begun}(x,A,B)$:

$$sec\ :\ (\exists x : \mathsf{Secret}, \exists b : \mathsf{Public})[!\mathsf{begun}(x,a,b)] \ \rightarrow\ \mathsf{Auth}(\exists a : \mathsf{Public}, \exists b' : \mathsf{Public})$$

In this type, x, b, a and b' are binders whose scope is the entire tag type. Within tag types, existentials scope both to the left and the right: for instance, the occurrence of a on the left of the arrow is bound by the existential on the right. The pattern $(\exists x : \mathsf{Secret}, \exists b : \mathsf{Public})$ left of the arrow restricts the tag *sec* to only be used for tagging messages that match this pattern. The assertion $!\mathsf{begun}(x,a,b)$ further restricts the tag *sec* to only be used if a (x,a,b)-session has previously begun. The authentication type $\mathsf{Auth}(\exists a : \mathsf{Public}, \exists b' : \mathsf{Public})$ right of the arrow expresses that messages tagged by *sec* require further authentication by principal a (acquired by a's digital signature) and may then be encrypted by some other principal b'. Note that the binder b' is not mentioned in the assertion $!\mathsf{begun}(x,a,b)$. For this reason, the type-checker accepts a public encryption key for the outer encryption. Because the tagged message contains a secret, the type-checker requires an outer encryption key whose matching decryption key is secret.

These type annotations, together with the robust safety of our type system, are enough to ensure safety of this protocol in the presence of opponents.

3 A Spi-calculus with Pattern-matching

3.1 Messages

As usual in spi-calculi, messages are modeled as elements of an algebraic datatype. They may be built from atomic names and variables by pairing and asymmetric-key encryption. Moreover, there are two special symbolic operators Enc and Dec with the following meanings: if message M represents a key pair, then $\mathsf{Enc}(M)$ represents its encryption and $\mathsf{Dec}(M)$ its decryption part. This language of messages is extremely parsimonious; below we show how to introduce derived forms for constructs such as symmetric-key encryption, message tagging and hashing.

Messages:

x, y, z	variables
m, n, ℓ	names
$L, M, N ::=$	message
$\quad n$	name
$\quad x$	variable
$\quad ()$	empty message
$\quad (M, N)$	message pair
$\quad \{\!\|M\|\!\}_N$	M encrypted under encryption key N
$\quad \{\!\|M\|\!\}_{N^{-1}}$	M encrypted under inverse of decryption key N
$\quad \mathsf{Enc}\,(M)$	encryption part of key pair M
$\quad \mathsf{Dec}\,(M)$	decryption part of key pair M

Syntactic restriction: No subterms of the form $\{\!|M|\!\}_{\mathsf{Dec}(N)^{-1}}$.
Define: A message M is *implementable* if it contains no subterms $\{\!|M|\!\}_{N^{-1}}$.

We write $\langle M_1, \ldots, M_n \rangle$ as shorthand for $(M_1, (\ldots, (M_n, ()) \ldots))$ and $(M_1, \ldots, M_n)$ for $(M_1, (\ldots, (M_{n-1}, M_n) \ldots))$. $fn(M)$ and $fv(M)$ are the sets of free names and free variables of M.

In the presentation of messages, we include asymmetric-key encryption $\{\!|M|\!\}_N$, which encrypts plaintext M with encryption key N. We use such messages to model both public key encryption and digital signing: in the former case the encryption key N is public and in the latter case N is secret. We also allow messages $\{\!|M|\!\}_{N^{-1}}$, which represents the encryption of plaintext M with the encryption key that matches decryption key N. This is clearly not an implementable operation and non-implementable messages are disallowed in processes (as defined in section 3.3). Technically, this restriction is needed to rule out non-implementable opponents: our type system does not guarantee safety against non-implementable opponents who, for instance, are capable of digitally signing messages using signature verification keys. Non-implementable messages *are* allowed in patterns (as defined in section 3.2) and types (as defined in section 4.3), however.

The reason for the syntactic restriction disallowing subterms of the form $\{\!|M|\!\}_{\mathsf{Dec}(N)^{-1}}$ is technical: we could instead have worked with messages modulo the equation $\{\!|M|\!\}_{\mathsf{Enc}(L)} = \{\!|M|\!\}_{\mathsf{Dec}(L)^{-1}}$; however, we prefer working with syntactic equality and only permit messages that are in a certain normal form for this simple equational theory. Substitution is defined as usual by induction on the structure of messages (processes, types, etc.). The definition is mostly standard, but we have to be careful to not build messages $\{\!|M|\!\}_{\mathsf{Dec}(N)^{-1}}$. We display the case where we substitute into a term of the form $\{\!|M|\!\}_{N^{-1}}$; all other cases are as usual.

Substitution into Messages:

$$(\{\!|M|\!\}_{N^{-1}})\{\sigma\} \quad \triangleq \quad \begin{cases} \{\!|M\{\sigma\}|\!\}_{\mathsf{Enc}(L)} & \text{if } N\{\sigma\} = \mathsf{Dec}\,(L) \\ \{\!|M\{\sigma\}|\!\}_{(N\{\sigma\})^{-1}} & \text{otherwise} \end{cases}$$

The following display shows our definitions for tagging, symmetric encryption, hashing and keyed hashing. We believe that these definitions capture the properties that we care about in this abstract setting.

Derived Forms for Messages:

$L(M) \triangleq \{\!\|M\|\!\}_{\mathsf{Enc}(L)}$	M tagged by label L
$\{M\}_N \triangleq \{\!\|M\|\!\}_{\mathsf{Enc}(N)}$	M encrypted by symmetric key N
$\#(M) \triangleq \mathit{hashtag}(\{\!\|M\|\!\}_{\mathit{hashkey}})$	hash of M (*hashkey*, *hashtag* are fixed global names)
$\#_N(M) \triangleq \{\!\|\#(M)\|\!\}_N$	keyed hash of M with secret N

The definitions for tagging and symmetric encryption are identical. However, the types for tags will differ from those for symmetric keys: tags are public whereas symmetric keys are secret. Hashing is modeled as encryption under a public *hashkey* that has no matching decryption key. The encrypted message is then tagged by a special *hashtag*. The *hashtag* is needed to obtain good typing rules for hashing: it alerts the type-checker to treat the *hashtag*ged message in a special way. Keyed hashing is modeled as symmetric encryption of a hashed message.

3.2 Patterns and Assertions

Patterns are of the form $\exists \vec{x} \,.\, M[\overline{A}]$, where $\overline{A}$ is an *assertion set*. The variables $\vec{x}$ act as binders. A message N matches a pattern $\exists \vec{x} \,.\, M[\overline{A}]$ if it is of the form $N = M\{\vec{x} \leftarrow \vec{L}\}$ and, in addition, the assertions $\overline{A}\{\vec{x} \leftarrow \vec{L}\}$ are satisfied. Importantly, the operational semantics for pattern-matching input against $\exists \vec{x} \,.\, M[\overline{A}]$ only checks that N matches M and ignores the assertions $\overline{A}$. Our type system, however, ensures statically that in well-typed processes $\overline{A}$ is also satisfied. It is important that the operational semantics ignores these assertions, because we want to model standard security protocols rather than enhancing these protocols by additional dynamic type- and assertion-checks. The pattern body M may have multiple occurrences of the same variable and it may contain variables that are not mentioned in $\vec{x}$: such variables are regarded as constants and must be matched exactly. For instance, the pattern $\exists x \,.\, (x, \{\!| x |\!\}_y)[]$ is matched by messages of the form $(M, \{\!| M |\!\}_y)$, but not by messages $(M, \{\!| M |\!\}_z)$ or $(M, \{\!| N |\!\}_y)$.

Patterns:

$X, Y, Z ::=$	pattern
$\quad \exists \vec{x} \,.\, M[\overline{A}]$	pattern matching term M binding $\vec{x}$

Syntactic restrictions: $\vec{x} \subseteq fv(M)$ and $\vec{x}$ distinct.
Define: A pattern $\exists \vec{x} \,.\, M[\overline{A}]$ is *implementable* if $(fn(M), fv(M) - \vec{x}, M \Vdash \vec{x})$. ($\Vdash$ is defined below).

Assertions:

$A, B, C, D ::=$	assertions
$\quad M : T$	type assertion
$\quad \mathsf{begun}(M)$	begun-once assertion
$\quad !\mathsf{begun}(M)$	begun-many assertion
$\quad \mathsf{fresh}(N : T)$	fresh-once assertion
$\quad !\mathsf{fresh}(N : T)$	fresh-many assertions

Assertions. Here are informal interpretations of the different kinds of assertions:

- $M : T$ means M has type T.
- $!\mathsf{begun}(M)$ means an M-session has previously begun.
- $\mathsf{begun}(M)$ means an M-session has previously begun and has not been ended yet.
- $\mathsf{fresh}(N : T)$ means N has been generated with type T and has not been used as a nonce yet.
- $!\mathsf{fresh}(N : T)$ is always false.

Begun-many assertions are useful for verifying so-called non-injective agreement (a.k.a. one-to-many correspondences), where a single begin-statement may be ended by multiple end-statements. Begun-once assertions are useful for verifying injective agreement (a.k.a. one-to-one correspondences), where each begin-statement may be ended by at most one end-statement. Injective agreement is needed to avoid replay attacks. In addition to type and begun-assertions, there are freshness assertions. These are used for typing challenge/response protocols where nonces ensure injective agreement. Freshness assertions help ensuring that each nonce is used at most once. Fresh-many assertions are always false and are included for the technical reason that we want promotion, as defined below, to be a total function. Technically, begun-once and fresh-once assertions are the only assertions that are not mapped to themselves by promotion. An assertion A is called *copyable* whenever $!A = A$. Begun-once and fresh-once assertions are not copyable.

Promotion, $!E$, $!\overline{A}$, $!A$:

$$!(\bar{x};\overline{A}) \triangleq (\bar{x};!\overline{A}); \quad !(A_1,\ldots,A_n) \triangleq (!A_1,\ldots,!A_n); \quad !(M:T) \triangleq (M:T);$$
$$!(\mathsf{begun}(M)) \triangleq !\mathsf{begun}(M); \quad !(\mathsf{fresh}(M:T)) \triangleq !\mathsf{fresh}(M:T);$$
$$!(!\mathsf{begun}(M)) \triangleq !\mathsf{begun}(M); \quad !(!\mathsf{fresh}(M:T)) \triangleq !\mathsf{fresh}(M:T)$$

Implementable patterns. Importantly, not all patterns are implementable. For instance, the patterns $\exists x, dk\,.\,\{\!|x|\!\}_{dk^{-1}}[\overline{A}]$ and $\exists x\,.\,\{\!|x|\!\}_{ek}[\overline{A}]$ are not implementable, because they would allow access to the plaintext without knowing the decryption key. Similarly $\exists x\,.\,\#(x)[\overline{A}]$ is not implementable, because it would allow inverting a one-way hash function. On the other hand, $\exists x\,.\,\{\!|x|\!\}_{dk^{-1}}[\overline{A}]$, $\exists x\,.\,\{\!|x|\!\}_{\mathsf{Enc}(k)}[\overline{A}]$ and $\exists x\,.\,(x,\#(x))[\overline{A}]$ are implementable patterns. A syntactic restriction forbids non-implementable input patterns in processes (as defined in section 3.3). This restriction is needed to rule out non-implementable opponents: our type system does not guarantee safety against non-implementable opponents who, for instance, are capable of decrypting messages without knowing decryption keys. Non-implementable patterns *are* allowed in types (as defined in section 4.3), however. We formalize the notion of implementable pattern by making use of the Dolev–Yao 'derivable message' judgment $\overline{M} \Vdash \overline{N}$ meaning 'An agent who knows messages $\overline{M}$ can construct messages $\overline{N}$.'

Dolev–Yao Derivability, $\overline{M} \Vdash \overline{N}$:

(DY True)
$$\frac{}{\overline{M} \Vdash}$$

(DY Id)
$$\frac{\overline{M}, N \Vdash \overline{L}}{\overline{M}, N \Vdash N, \overline{L}}$$

(DY Copy)
$$\frac{\overline{M} \Vdash N, \overline{L}}{\overline{M} \Vdash N, N, \overline{L}}$$

(DY Nil)
$$\frac{\overline{M} \Vdash \overline{L}}{\overline{M} \Vdash (), \overline{L}}$$

(DY Pair)
$$\frac{\overline{M} \Vdash N, N', \overline{L}}{\overline{M} \Vdash (N,N'), \overline{L}}$$

(DY Split)
$$\frac{\overline{M}, N, N' \Vdash \overline{L}}{\overline{M}, (N,N') \Vdash \overline{L}}$$

(DY Key)
$$\frac{\overline{M} \Vdash N, \overline{L} \quad k \in \{\mathsf{Enc}, \mathsf{Dec}\}}{\overline{M} \Vdash k(N), \overline{L}}$$

(DY Encrypt)
$$\frac{\overline{M} \Vdash N, N', \overline{L}}{\overline{M} \Vdash \{\!|N'|\!\}_N, \overline{L}}$$

(DY Decrypt)
$$\frac{\overline{M} \Vdash N \quad \overline{M}, N' \Vdash \overline{L}}{\overline{M}, \{\!|N'|\!\}_{N^{-1}} \Vdash \overline{L}}$$

(DY Unencrypt)
$$\frac{\overline{M} \Vdash N \quad \overline{M}, N' \Vdash \overline{L}}{\overline{M}, \{\!|N'|\!\}_{\mathsf{Enc}(N)} \Vdash \overline{L}}$$

We use some convenient syntactic abbreviations that treat patterns as if they were messages containing binding existentials. These 'derived forms' for patterns are defined below. For example:

$$(x, \exists x : \mathsf{Public})[!\mathsf{begun}(x)] \;\equiv\; \exists x\,.\,(x,x)[x : \mathsf{Public}, !\mathsf{begun}(x)]$$

$$\begin{aligned}&\{\!|\{\!|sec(\exists x : \mathsf{Secret}, B)|\!\}_{dsA^{-1}}|\!\}_{dpB^{-1}}[!\mathsf{begun}(x,A,B)]\\ &\quad\equiv\; \exists x\,.\,\{\!|\{\!|sec(x,B)|\!\}_{dsA^{-1}}|\!\}_{dpB^{-1}}[x : \mathsf{Secret}, !\mathsf{begun}(x,A,B)]\end{aligned}$$

$$\begin{aligned}&(\{\!|\#(\ell(\exists m : \mathsf{Public}, \_))|\!\}_{\_^{-1}}, \exists a : \mathsf{Un}, \exists b : \mathsf{Un})[!\mathsf{begun}(m,a,b)]\\ &\quad\equiv\; \exists m,a,b,x,y\,.\,(\{\!|\#(\ell(m,x))|\!\}_{y^{-1}}, a, b)[m : \mathsf{Public}, a : \mathsf{Un}, b : \mathsf{Un}, !\mathsf{begun}(m,a,b)]\end{aligned}$$

In these derived forms, existentials scope both to the left and the right. The following display contains the complete definition of the derived forms. Note that no scoping ambiguities arise, by the side conditions on the clauses for tupling and encryption. For instance, $(\exists x : T, \exists x : U)[\mathsf{begun}(x)]$ is undefined because $\{x\} \cap \{x\} \neq \emptyset$.

Derived Forms for Patterns:

$$
\begin{array}{l}
M \triangleq \{M \mid \ \}; \quad T \triangleq \exists x \,.\, x[x : T] \text{ for fresh } x;\\
\exists x \triangleq \exists x \,.\, x[\,]; \quad \_ \triangleq (\exists x) \text{ for fresh } x;\\
X : T \triangleq \exists \vec{x} \,.\, M[\overline{A},\ M : T], \text{ if } X = \exists \vec{x} \,.\, M[\overline{A}];\\
\{\!|X|\!\}_Y \triangleq \exists \vec{x}, \vec{y} \,.\, \{\!|M|\!\}_N[\overline{A}, \overline{B}], \text{ if } X = \exists \vec{x} \,.\, M[\overline{A}], Y = \exists \vec{y} \,.\, N[\overline{B}], \{\vec{x}\} \cap \{\vec{y}\} = \emptyset;\\
\{\!|X|\!\}_{Y^{-1}} \triangleq \exists \vec{x}, \vec{y} \,.\, \{\!|M|\!\}_{N^{-1}}[\overline{A}, \overline{B}], \text{ if } X = \exists \vec{x} \,.\, M[\overline{A}], Y = \exists \vec{y} \,.\, N[\overline{B}], \{\vec{x}\} \cap \{\vec{y}\} = \emptyset;\\
X[\overline{B}] \triangleq \exists \vec{x} \,.\, M[\overline{A}, \overline{B}], \text{ if } X = \exists \vec{x} \,.\, M[\overline{A}];\\
(X_1, \ldots, X_n) \triangleq \exists \vec{x}_1, \ldots, \vec{x}_n \,.\, (M_1, \ldots, M_n)[\overline{A}_1, \ldots, \overline{A}_n], \text{ if } X_i = \exists \vec{x}_i \,.\, M_i[\overline{A}_i], i \neq j \Rightarrow \{\vec{x}_i\} \cap \{\vec{x}_j\} = \emptyset;\\
\langle X_1, \ldots, X_n \rangle \triangleq \exists \vec{x}_1, \ldots, \vec{x}_n \,.\, \langle M_1, \ldots, M_n \rangle[\overline{A}_1, \ldots, \overline{A}_n], \text{ if } X_i = \exists \vec{x}_i \,.\, M_i[\overline{A}_i], i \neq j \Rightarrow \{\vec{x}_i\} \cap \{\vec{x}_j\} = \emptyset;\\
Y(X) \triangleq \{\!|X|\!\}_{\mathsf{Enc}(Y)}; \quad \{X\}_Y \triangleq \{\!|X|\!\}_{\mathsf{Enc}(Y)}; \quad \#(X) \triangleq \mathit{hashtag}(\{\!|X|\!\}_{\mathit{hashkey}}); \quad \#_Y(X) \triangleq \{\#(X)\}_Y
\end{array}
$$

3.3 Processes

Processes:

$O, P, Q, R ::=$	process
$\mathsf{out}\ N\ M$	asynchronous output of M on N
$\mathsf{inp}\ N\ X; P$	input from N against pattern X
$\mathsf{new}\ n{:}T; P$	name generation
$P \mid Q$	parallel composition
$!P$	replication
$\mathbf{0}$	inactivity
$\mathsf{begin}(L);\ P$	begin L-session to be ended at most once
$\mathsf{begin!}(L);\ P$	begin L-session to be ended arbitrarily often
$\mathsf{end}(L);\ P$	end L-session

Syntactic restrictions:

- In $(\mathsf{out}\ N\ M)$, both N and M are implementable messages.
- In $(\mathsf{inp}\ N\ X; P)$, N is an implementable message and X is an implementable pattern.

Scope:

- The scope of $\vec{x}$ in $(\mathsf{inp}\ N\ \exists \vec{x} \,.\, M[\overline{A}]; P)$ is M, $\overline{A}$ and P.
- The scope of n in $(\mathsf{new}\ n{:}T; P)$ is P.

We often elide $\mathbf{0}$ from the end of processes and write $(\mathsf{out}\ N\ M; P)$ for $(\mathsf{out}\ N\ M \mid P)$. Note that the standard Dolev–Yao attacker can be expressed as an implementable process.

We impose some technical restrictions on *opponent processes*. These restrictions only concern type annotations and correspondence assertions, both of which are ignored at runtime. In other words, for every process P there is an opponent process O with the same "type- and correspondence-erasure" as P. Thus, the restrictions on opponents do not impose restrictions on type- and correspondence-free processes. As a consequence, our definition of opponents includes the Dolev–Yao attacker. The definition of opponent processes makes use of the type Un—the type of data that may flow to and from opponents.

Definition 3.1 (Opponent Processes) An *opponent process* is an implementable process that does not contain correspondence assertions, whose type annotations on new names are all Un, and whose input patterns are all of the form $\exists\vec{x}.M[M : \mathsf{Un}]$ for some $\vec{x}$, M.

3.4 Semantics

The operational semantics is a reduction semantics that operates on *computation states* of the form $\overline{A} ::: P$. The assertion set $\overline{A}$ keeps track of sessions that have begun. This bookkeeping is needed to recognize authenticity errors. Remember our convention that overbars indicate multisets, so $\overline{A}$ is a multiset of assertions. We interpret a $!\mathsf{begun}(M)$-assertion as infinitely many $\mathsf{begun}(M)$-assertions. This is formalized by the rules (Ass Copy One) and (Ass Copy Many) below.

Structural Equivalence of Assertion Sets, $\overline{A} \equiv \overline{B}$:

$\overline{A} \equiv \overline{A}$	(Ass Equiv)
$\overline{A} \equiv \overline{B} \Rightarrow \overline{B} \equiv \overline{A}$	(Ass Symm)
$\overline{A} \equiv \overline{B}, \overline{B} \equiv \overline{C} \Rightarrow \overline{A} \equiv \overline{C}$	(Ass Trans)
$(\overline{A}, !\mathsf{begun}(M)) \equiv (\overline{A}, !\mathsf{begun}(M), \mathsf{begun}(M))$	(Ass Copy One)
$(\overline{A}, !\mathsf{begun}(M)) \equiv (\overline{A}, !\mathsf{begun}(M), !\mathsf{begun}(M))$	(Ass Copy Many)

Structural Process Equivalence, $P \equiv Q$, and State Equivalence, $(\overline{A} ::: P) \equiv (\overline{B} ::: Q)$:

$P \equiv P$	(Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$Q \equiv R \Rightarrow P \mid Q \equiv P \mid R$	(Struct Par)
$P \mid \mathbf{0} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$\overline{A} \equiv \overline{A'}, P \equiv P' \to (\overline{A} ::: P) \equiv (\overline{A'} ::: P')$	(Struct State)

Our reduction semantics consists of a structural process equivalence relation and a state transition relation. The reduction rule (Redn IO) formalizes pattern-matching input. The reduction rules (Redn Begin One) and (Redn Begin Many) keep track of beginning sessions, and the rule (Redn End) checks that a session is only ended if it has previously begun.

State Transition, $(\overline{A} ::: P) \to (\overline{B} ::: Q)$:

(Redn Equiv)

$$\frac{(\overline{A} ::: P) \equiv (\overline{A'} ::: P') \to (\overline{B'} ::: Q') \equiv (\overline{B} ::: Q)}{(\overline{A} ::: P) \to (\overline{B} ::: Q)}$$

$n \notin fn(\overline{A}, Q) \Rightarrow (\overline{A} ::: (\mathsf{new}\, n : T; P) \mid Q) \to (\overline{A}, n : T, \mathsf{fresh}(n : T) ::: P \mid Q)$	(Redn New)
$(\overline{A} ::: (\mathsf{out}\, L\, M\{\vec{x}\leftarrow\vec{N}\} \mid \mathsf{inp}\, L\, \exists\vec{x}.M[\overline{A}]; P) \mid Q) \to (\overline{A} ::: P\{\vec{x}\leftarrow\vec{N}\} \mid Q)$	(Redn IO)
$(\overline{A} ::: (\mathsf{begin}(M); P) \mid Q) \to (\overline{A}, \mathsf{begun}(M) ::: P \mid Q)$	(Redn Begin One)
$(\overline{A} ::: (\mathsf{begin!}(M); P) \mid Q) \to (\overline{A}, !\mathsf{begun}(M) ::: P \mid Q)$	(Redn Begin Many)
$(\overline{A}, \mathsf{begun}(M) ::: (\mathsf{end}(M); P) \mid Q) \to (\overline{A} ::: P \mid Q)$	(Redn End)

Definition 3.2 (Safety for Authenticity) A computation state s is *safe for authenticity* iff $s \to^* \overline{A} ::: \mathsf{end}(M); P$ implies that $\overline{A} \equiv (\overline{B}, \mathsf{begun}(M))$ for some $\overline{B}$. A process P is *safe for authenticity* iff $(\vec{n} : \mathsf{Un} ::: P)$ is safe for authenticity, where $\vec{n} = fn(P)$ and $\vec{n}$ distinct.

As an example, consider the following process P:

$$P \triangleq !P_A \mid !P_B, \text{ where } \begin{array}{rcl} P_A & \triangleq & \mathsf{new}\ m : \mathsf{Public};\ \mathsf{begin!}(m,A,B);\ \mathsf{out}\ \mathit{net}\ (m,B) \\ P_B & \triangleq & \mathsf{inp}\ \mathit{net}\ (\exists x, B)[!\mathsf{begun}(x,A,B)];\ \mathsf{end}(x,A,B) \end{array}$$

Process P is safe, by Definition 3.2. However, what we are really interested in is safety in the presence of opponents.

Definition 3.3 (Closed Processes) A process P is *closed* iff $fv(P) = \emptyset$. Closed processes may contain free names.

Definition 3.4 (Robust Safety for Authenticity) A process P is called *robustly safe for authenticity* whenever $(O \mid P)$ is safe for authenticity for all closed opponent processes O.

Process P is not robustly safe, because $(\mathsf{out}\ \mathit{net}\ (N,B) \mid P)$ is not safe. The crucial property of our type system is that closed, well-typed processes with public free names are robustly safe. The following theorem captures this property formally. The judgment $(\vec{n} : \vec{T} \vdash P)$ may be read as 'P is a well-typed process assuming that its free names $\vec{n}$ have types $\vec{T}$'.

Theorem 3.1 (Robust Safety for Authenicity) *If $\vec{n}$ are distinct names and $(\vec{n} : \mathsf{Un} \vdash P)$, then P is robustly safe for authenticity.*

It follows from this theorem that the example process P does not type-check. We can ensure robust safety by adding encryption. As a prudent engineering practice, we also tag messages before encryption.

$$\begin{array}{rcl} Q & \triangleq & \mathsf{new}\ \ell : (\exists x : \mathsf{Public}, \exists b : \mathsf{Public})[!\mathsf{begun}(x,a,b)] \to \mathsf{Auth}\langle \exists a : \mathsf{Public}\rangle; \\ & & \mathsf{new}\ A : \mathsf{Public};\ \mathsf{new}\ B : \mathsf{Public};\ \mathsf{new}\ k : \mathsf{SigningKP}(A); \\ & & \quad (\ !\mathsf{out}\ \mathit{net}\ (\ell, A, B, \mathsf{Dec}(k))\ \mid\ !Q_A(\mathsf{Enc}(k))\ \mid\ !Q_B(\mathsf{Dec}(k))\ ) \end{array}$$

$$Q_A(ek : \mathsf{SigningEK}(A)) \triangleq \mathsf{new}\ m : \mathsf{Public};\ \mathsf{begin!}(m,A,B);\ \mathsf{out}\ \mathit{net}\ \{\!|\ell(m,B)|\!\}_{ek}$$

$$Q_B(dk : \mathsf{SigningDK}(A)) \triangleq \mathsf{inp}\ \mathit{net}\ \{\!|\ell(\exists x : \mathsf{Public}, B)|\!\}_{dk^{-1}}[!\mathsf{begun}(x,A,B)];\ \mathsf{end}(x,A,B)$$

We publish the tag ℓ, the principal names A and B, and the public signature verification key $\mathsf{Dec}(k)$, in order to model that opponents have access to this data. The signing key $\mathsf{Enc}(k)$, of course, is not published. Process Q type-checks, i.e., $\mathit{net} : \mathsf{Un} \vdash Q$. By robust safety of the type system, it follows that Q is robustly safe. The types of tag ℓ and key pair k constrain the sender A to only tag-and-encrypt if an (m,A,B)-session has previously begun. As a consequence, the receiver B obtains this guarantee after decrypting-and-untagging. In comparison, in protocol P without encryption the receiver obtains no such guarantee, because the received message may be fake coming from an opponent.

4 The Type System

4.1 Overview of Judgments

As is usual in most type systems, we give our judgments relative to an *environment*. Environments in our system are pairs of the form $(\bar{x};\overline{A})$. The component $\bar{x}$ lists free variables and $\overline{A}$ lists assertions that may be used as assumptions.

Environments:

$E, F, G ::=$	environments
$\quad \bar{x};\overline{A}$	$\quad$ environment
$dom(\bar{x};\overline{A}) \triangleq \bar{x}$	environment domain

For instance, this is an environment:

$$dk, x;\ dk : \mathsf{SigningDK}(A),\ \{\!|x,B|\!\}_{dk^{-1}} : \mathsf{Un},\ !\mathsf{begun}(x,A,B)$$

This environment permits processes with free variables *dk* and *x*, and allows the type-checker to use the assumptions $dk : \mathsf{SigningDK}(A)$ (meaning that *dk* is the decryption part of *A*'s signing key), $\{\!|x,B|\!\}_{dk^{-1}} : \mathsf{Un}$ (meaning that $\{\!|x,B|\!\}_{dk^{-1}}$ may come from an opponent) and $!\mathsf{begun}(x,A,B)$ (meaning that an (x,A,B)-session has previously begun). A difference to previous type systems for authenticity [24, 25, 23] is that we are unifying the notions of *variable environment* and *process effect* into a common language of environments. In these previous papers, the above environment would be split into the variable environment $dk : \mathsf{SigningDK}(A)$, and the process effect $\mathsf{trust}(\{\!|x,B|\!\}_{dk^{-1}} : \mathsf{Un}), !\mathsf{begun}(x,A,B)$.

Here are the judgments of our type system:

Judgments:

$T :: K$	type T has kind K
$K \leq H$	K is a subkind of H
$T \leq U$	T is a subtype of U
$E \vdash \diamond$	E is a good environment
$E \vdash \overline{A}$	assertions $\overline{A}$ are valid in E
$E \vdash P$	process P is well-typed in E

Meta-variable $\mathcal{R}$ ranges over *right-hand sides*, i.e., $\diamond$, $\overline{A}$ and P.

The good-environment judgment is simple:

Good Environment, $E \vdash \diamond$:

(Good Env)

$$\frac{fv(\overline{A}) \subseteq \tilde{x}}{\tilde{x}; \overline{A} \vdash \diamond}$$

In addition to these judgments, there is an auxiliary judgment for pattern-matching.

Auxiliary Judgments for Pattern-Matching:

$E \vdash M \in X$	M matches pattern X in E

This judgment is auxiliary in the sense that it is a shorthand for a formula built from basic judgments:

Pattern-Matching (where $X = \exists \vec{x}\,.\,N[\overline{A}]$):

$$E \vdash M \in X \ \stackrel{\Delta}{=}\ (\exists \vec{L})(M = N\{\vec{x} \leftarrow \vec{L}\} \ \wedge\ E \vdash M : \mathsf{Top}, \overline{A}\{\vec{x} \leftarrow \vec{L}\})$$

For example:

- $E \vdash (M,B) \in (\exists x : \mathsf{Public}, B)[!\mathsf{begun}(x,A,B)]$ is defined to be
  $E \vdash (M,B) : \mathsf{Top}, M : \mathsf{Public}, !\mathsf{begun}(M,A,B)$.

Finally, we also allow assertions $M \in X$ to occur in environments of auxiliary judgments. We could resolve such occurrences in the same way as on the right of $\vdash$:

$$E, M \in X \vdash \mathcal{R} \ \stackrel{\Delta}{=}\ (\exists \vec{L})(M = N\{\vec{x} \leftarrow \vec{L}\} \ \wedge\ E, M : \mathsf{Top}, \overline{A}\{\vec{x} \leftarrow \vec{L}\} \vdash \mathcal{R}) \qquad \text{(a possible definition)}$$

This definition is safe and good enough to type-check many protocols. However, there are some protocols that require a more liberal definition of pattern-matching on the left: Assume, for instance, the type-checker is faced with a proof goal of the form $(E, x \in (\exists y, \exists z) \vdash P)$. Such a goal may arise after parsing an input statement of the form $(\mathsf{inp}\ net\ \{\!|\exists x|\!\}_{\mathsf{Enc}(k)}; P)$ where k's type constrains $\mathsf{Enc}(k)$ to only encrypt messages that match $(\exists y, \exists z)$. In such a situation, the type-checker knows that at runtime x must refer to a pair. With the above definition of pattern-matching on the left, however, the assertion $x \in (\exists y, \exists z)$ cannot be resolved: x does not match $(\exists y, \exists z)$. A more liberal definition would substitute the pair (y,z) for x in the entire proof goal, resulting in the new proof goal $(y,z,E\{x \leftarrow (y,z)\} \vdash P\{x \leftarrow (y,z)\})$ (assuming that y and z are fresh variables).

Our definition of pattern-matching on the left uses unification. The function $mgu(\bar{x}, M, N)$ takes two messages M and N where $fv(M,N) \subseteq \bar{x}$, and returns a most general unifier $\sigma : \bar{x} \to \bar{y}$ of M and N, where $\bar{x}$ is the *domain* of σ and $\bar{y} \supseteq \cup\{fv(\sigma(x)) \mid x \in \bar{x}\}$ is its *range*. If M and N are not unifiable, then $mgu(\bar{x}, M, N) = \bot$. As a shorthand, we write $mgu(E, M, N)$ for $mgu(dom(E), M, N)$. If $dom(E) = \bar{x}$, the application of substitution $\sigma : \bar{x} \to \bar{y}$ to environment E is defined by $(\bar{x}; \overline{A})\{\sigma\} \triangleq (\bar{y}; \overline{A}\{\sigma\})$. (See Appendix A for more detailed definitions of typed substitutions and most general unifiers.)

Pattern-Matching on the Left (where $X = \exists \vec{x} \,.\, N[\overline{A}]$ and $\vec{x} \cap dom(E) = \emptyset$):

$$E, M \in X \vdash \mathcal{R} \;\triangleq\; \begin{cases} \text{false}, & \text{if } fv(\mathcal{R}) \not\subseteq dom(E) \\ \text{true}, & \text{if } fv(\mathcal{R}) \subseteq dom(E) \text{ and } mgu((\vec{x}, E), M, N) = \bot \\ (\vec{x}, E, M : \mathsf{Top}, \overline{A})\{\sigma\} \vdash \mathcal{R}\{\sigma\}, & \\ \quad & \text{if } fv(\mathcal{R}) \subseteq dom(E) \text{ and } mgu((\vec{x}, E), M, N) = \sigma \end{cases}$$

For example:

- $E, (M,B) \in (\exists x : \mathsf{Public}, B)[!\mathsf{begun}(x,A,B)] \vdash \mathcal{R}$ is defined to be
  $E, (M,B) : \mathsf{Top}, M : \mathsf{Public}, !\mathsf{begun}(M,A,B) \vdash \mathcal{R}$.
- $y, E, y \in (\exists x : \mathsf{Public}, B)[!\mathsf{begun}(x,A,B)] \vdash \mathcal{R}$ is defined to be
  $x, E\{y \leftarrow (x,B)\}, (x,B) : \mathsf{Top}, x : \mathsf{Public}, !\mathsf{begun}(x,A,B) \vdash \mathcal{R}\{y \leftarrow (x,B)\}$.
- $() \in ((),()) \vdash \mathcal{R}$ is defined to be true, because $()$ and $((),())$ are not unifiable.

4.2 Kinds

A message is *publishable* if it may be sent to an untrusted target. For instance, ciphertext $\{\!|M|\!\}_{ek}$ is publishable if the decryption key for ek is secret. If ek is a signing key, whose corresponding decryption key is public, then $\{\!|M|\!\}_{ek}$ is only publishable if M is already publishable.

A message is *untainted* if it has been received from a trusted source. For instance, if dk's matching encryption key is a trusted agent's signing key, then M is untainted even if $\{\!|M|\!\}_{dk^{-1}}$ is tainted. If dk's matching encryption key is public, then M is only untainted if $\{\!|M|\!\}_{dk^{-1}}$ is already untainted.

An important part of the type system is a *kinding relation* $(T :: K)$ that assigns kinds K to types T. Kinding is actually a function, so every type will have exactly one kind. Kinds are subsets of the two-element set $\{\mathsf{Public}, \mathsf{Tainted}\}$. The type system is designed so that the following statements hold:

- If $(T :: K)$ and $\mathsf{Public} \in K$, then members of type T are publishable.
- If $(T :: K)$ and $\mathsf{Tainted} \notin K$, then members of type T are untainted.

We say that type T is *public* (respectively *tainted*) if $T :: K \ni \mathsf{Public}$ (respectively $T :: K \ni \mathsf{Tainted}$).

We next define a subkinding relation, which we will use in our definition of subtyping. An important link between subtyping and subkinding is that the kinding function is monotone with respect to these orders, i.e., if T is a subtype of U, $T :: K$ and $U :: H$, then K is a subkind of H.

Subkinding:

$$\frac{(\mathsf{Public} \in H) \Rightarrow (\mathsf{Public} \in K) \quad (\mathsf{Tainted} \in K) \Rightarrow (\mathsf{Tainted} \in H)}{K \leq H}$$

$$\begin{tikzcd}
& \{\mathsf{Tainted}\} \arrow[dl, dash] \arrow[dr, dash] & \\
\{\mathsf{Public},\mathsf{Tainted}\} & & \{\} \\
& \{\mathsf{Public}\} \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

4.3 Types

We next define types and the kinding relation:

Types:

$T,U,V ::=$	types
Top	top type
$K\,\mathsf{Top}$	top type for generative types of kind K
$K\,\mathsf{Auth}(L)$	authentication types of kind K
$(K,H)\,KT(X)$	key type whith encryption kind K and decryption kind H
$(K,H)\,NT(\overline{A})$	valid nonce type with challenge kind K and response kind H
Stale	stale nonce type
$KT ::=$	key qualifiers
EK	encryption key
DK	decryption key
KP	key pair
$NT ::=$	qualifiers for valid nonces
Chall	nonce challenge
Resp	nonce response

Syntactic restrictions:

- In $(K,H)\,KT(\exists\vec{x}.M[\overline{A}])$: $!\overline{A} = \overline{A}$.
- In $(K,H)\,\mathsf{Chall}(\overline{A})$: $\overline{A} = \emptyset$ or $\mathsf{Public} \notin K$.
- In $(K,H)\,\mathsf{Resp}(\overline{A})$: $\overline{A} = \emptyset$ or $\mathsf{Tainted} \notin H$ or $\mathsf{Public} \notin K \cup H$.

Define: A type is *generative* if it is not of the form Top or $(K,H)\,\mathsf{Resp}(\overline{A})$ for some K, H, $\overline{A}$.

Kinding, $T :: K$:

$\mathsf{Top} :: \{\mathsf{Tainted}\}$; $\quad K\,\mathsf{Top} :: K$; $\quad K\,\mathsf{Auth}(L) :: K$;

$(K,H)\,\mathsf{EK}(X) :: K$; $\quad (K,H)\,\mathsf{DK}(X) :: H$; $\quad (K,H)\,\mathsf{KP}(X) :: K \cap H$;

$(K,H)\,\mathsf{Chall}(\overline{A}) :: K$; $\quad (K,H)\,\mathsf{Resp}(\overline{A}) :: H$; $\quad \mathsf{Stale} :: \{\mathsf{Public}\}$

Note that for most types their kinds are given by kind annotations. Key types have two kind annotations: the first annotation is the kind of the encryption part and the second one the kind of the decryption part. The kind of a key pair is the intersection of these two kind annotations. Nonce types also have two kind annotations: the challenge kind and the response kind.

Generative types. Our typed treatment of challenge/response protocols requires that nonces get generated with challenge types. We therefore disallow response types as generative types. (Only generative types may be used as type annotations on new-generated names.) We define Top to be non-generative in order to avoid equating modulo subtyping the types Top and $K\,\mathsf{Top}$.

Top types. Top is the largest type of the type hierarchy. In addition, there are four types $K\,\mathsf{Top}$ that are the largest generative types of kind K. The following derived forms are convenient:

Derived Forms for Generative Top Types:

$$\mathsf{Secret} \triangleq \emptyset\,\mathsf{Top}; \qquad \mathsf{Tainted} \triangleq \{\mathsf{Tainted}\}\,\mathsf{Top};$$
$$\mathsf{Public} \triangleq \{\mathsf{Public}\}\,\mathsf{Top}; \qquad \mathsf{Un} \triangleq \{\mathsf{Public},\mathsf{Tainted}\}\,\mathsf{Top}$$

Let's spell out the semantics of generative top types:

- Members of type Secret are untainted but not publishable.
- Members of type Public are untainted and publishable.
- Members of type Tainted are neither untainted nor publishable.
- Members of type Un are publishable but not untainted.

Subtyping. Here is the subtyping relation:

Subtyping, $T \leq U$:

(Subty Refl) $$\frac{}{T \leq T}$$

(Subty Top) $$\frac{}{T \leq \mathsf{Top}}$$

(Subty Top Gen) $$\frac{T :: K \quad K \leq H \quad T \text{ generative}}{T \leq H\,\mathsf{Top}}$$

(Subty Auth) $$\frac{K \leq H}{K\,\mathsf{Auth}(L) \leq H\,\mathsf{Auth}(L)}$$

(Subty Public Tainted) $$\frac{T :: K \cup \{\mathsf{Public}\} \quad U :: H \cup \{\mathsf{Tainted}\}}{T \leq U}$$

(Subty Stale Nonce) $$\frac{}{\mathsf{Stale} \leq (K,H)\,NT(\bar{A})}$$

The rules (Subty Top) and (Subty Top Gen) formalize our previous informal description of top types. It is instructive to picture the subtyping relation on generative top types:

$$\begin{tikzcd}
 & \mathsf{Tainted} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathsf{Un} \arrow[dr, dash] & & \mathsf{Secret} \arrow[dl, dash] \\
 & \mathsf{Public} &
\end{tikzcd}$$

The rule (Subty Auth) says that authentication types $K\,\mathsf{Auth}(L)$ with identical arguments L inherit their ordering from the subkinding ordering on the kind annotations. The rule (Subty Public Tainted) states that any message of public type also has any tainted type, as in [25]. Technically, this rule is crucial for showing that opponent processes are well-typed in environments that assign public types to their free names—a property we call *opponent typability*. For instance, the rule (Subty Public Tainted) allows us to upcast public types to tainted key types. As a result, well-typed processes may use members of any public type as (tainted) encryption keys. Opponent typability is a property that our system has in common with many other type systems for secrecy or authenticity in the spi-calculus. Opponent typability allows us to easily obtain robust safety from safety (see Section 7.2). The rule (Subty Stale Nonce) says that it is safe to upcast stale nonce types to valid nonce types. This is safe because there are other means, namely freshness assertions, for avoiding that stale nonces are reused (see section 4.7).

Key types: kind annotations. Key types are used as types for symmetric keys, public encryption keys, digital signing keys and message tags. The differences are in the kind annotations. The following table summarizes how different kinds of key types represent various forms of cryptography:

$(\{\mathsf{Secret}\},\{\mathsf{Secret}\})\,KT(X)$	symmetric key types
$(\{\mathsf{Public}\},\{\})\,KT(X)$	public encryption key types
$(\{\},\{\mathsf{Public}\})\,KT(X)$	signing key types
$(\{\mathsf{Public}\},\{\mathsf{Public}\})\,KT(X)$	tag types
$\mathsf{Tainted} \in K \cup H,\ (K,H)\,KT(X)$	needed for opponent typability

Tainted key types are needed for obtaining opponent typability: they allow us to use any public message as a tainted key.

Authentication Types. Here is an informal generic reading for authentication types:

- Members of $K\,\mathsf{Auth}(L)$ require encryption by keys associated with names mentioned in L. A little abbreviated: *Members of* $K\,\mathsf{Auth}(L)$ *require authentication by L.*

Typically, L is a list of principal names and authentication is acquired by encryption with their signing keys. We use the following shorthands for authentication types, similar to those for generative top types:

Derived Forms for Authentication Types:

$$\mathsf{Secret}(L) \triangleq \emptyset\,\mathsf{Auth}(L); \qquad \mathsf{Tainted}(L) \triangleq \{\mathsf{Tainted}\}\,\mathsf{Auth}(L);$$
$$\mathsf{Public}(L) \triangleq \{\mathsf{Public}\}\,\mathsf{Auth}(L); \qquad \mathsf{Un}(L) \triangleq \{\mathsf{Public},\mathsf{Tainted}\}\,\mathsf{Auth}(L)$$

Key and Authentication Types. Key and authentication types are closely related. In useful key types $(K,H)\,KT(X)$, the pattern X is always of the form $X = (X_{plain}, X_{auth})$. If k is an encryption key of type $(K,H)\,\mathsf{EK}(X_{plain}, X_{auth})$, it can be used to encrypt messages M that match X_{plain} and the resulting ciphertext $\{\!|M|\!\}_k$ is of type $J\,\mathsf{Auth}(L)$ where L matches X_{auth}.[1] The type $J\,\mathsf{Auth}(L)$ expresses that $\{\!|M|\!\}_k$ requires further authentication by L, which can be acquired by encryption with a key that is associated with L.

In order to show how the the slot X_{auth} is used, let us first "switch it off" by setting it to the constant pattern for the empty message: $X_{auth} = ()$. Consider the following example, which has X_{auth} "switched off":

$$\begin{aligned}
&\mathsf{Signing}KT \ \triangleq\ (\{\},\{\mathsf{Public}\})\,KT(\mathsf{Public}(),())\\
&P \ \triangleq\ \mathsf{new}\ k : \mathsf{SigningKP};\ (!\mathsf{out}\ \mathit{net}\ (\mathsf{Dec}\,(k))\ |!Q_A(\mathsf{Enc}\,(k))\ |!Q_B(\mathsf{Dec}\,(k)))\\
&P_A(\mathit{esA} : \mathsf{SigningEK}) \ \triangleq\ \mathsf{new}\ m : \mathsf{Public};\ \mathsf{begin!}(m,A,B);\ \mathsf{out}\ \mathit{net}\ \{\!|\mathit{snd}(m,A,B)|\!\}_{\mathit{esA}}\\
&P_B(\mathit{dsA} : \mathsf{SigningDK}) \ \triangleq\ \mathsf{inp}\ \mathit{net}\ \{\!|\mathit{snd}(\exists x : \mathsf{Public}, A, B)|\!\}_{\mathit{dsA}^{-1}}[!\mathsf{begun}(x,A,B)];\ \mathsf{end}(x,A,B)
\end{aligned}$$

In this protocol, A sends to B the signed, public message m. The sender A also includes both her own and the intended receiver's identity. This protocol type-checks, if we assign the following type to the tag snd:

$$\begin{aligned}
&\mathit{snd} : (\{\mathsf{Public}\},\{\mathsf{Public}\})\,\mathsf{KP}(X_{plain},())\\
&\text{where } X_{plain} \triangleq (\exists x{:}\mathsf{Public}, \exists a{:}\mathsf{Public}, \exists b{:}\mathsf{Public})[!\mathsf{begun}(x,a,b)]
\end{aligned}$$

With this tag type and assuming that A and B have type Public both the tagged message $\mathit{snd}(m,A,B)$ and the signed message $\{\!|\mathit{snd}(m,A,B)|\!\}_{\mathit{esA}}$ have type $\mathsf{Public}()$. Note that this protocol is robustly safe even if the sender's identity is omitted ($\{\!|\mathit{snd}(m,B)|\!\}_{\mathit{esA}}$ instead of $\{\!|\mathit{snd}(m,A,B)|\!\}_{\mathit{esA}}$), because the sender's identity is

[1] The kind J of $\{\!|M|\!\}_k$'s authentication type depends on the kinds (K,H) of the key type and the kind of the plaintext type; see Section 4.5 for the precise rule how to determine J.

associated with her signing key and a receiver of a signed message can infer her identity from the signature. An optimized protocol omits A's identity:

$$\begin{aligned}
&\mathsf{Signing}KT(A) \triangleq (\{\},\{\mathsf{Public}\})\,KT(\mathsf{Public}(A),())\\
&Q \triangleq \mathsf{new}\;k:\mathsf{SigningKP}(A);\;(!\mathsf{out}\;net\;(\mathsf{Dec}\,(k))\;|!Q_A(\mathsf{Enc}\,(k))!\mid Q_B(\mathsf{Dec}\,(k)))\\
&Q_A(esA:\mathsf{SigningEK}(A)) \triangleq \mathsf{new}\;m:\mathsf{Public};\;\mathsf{begin!}(m,A,B);\;\mathsf{out}\;net\;\{\!|snd(m,B)|\!\}_{esA}\\
&Q_B(dsA:\mathsf{SigningDK}(A)) \triangleq \mathsf{inp}\;net\;\{\!|snd(\exists x:\mathsf{Public},B)|\!\}_{dsA^{-1}}[!\mathsf{begun}(x,A,B)];\;\mathsf{end}(x,A,B)
\end{aligned}$$

Note that we have modified the type of A's signing key. The new key type expresses that the key may be used for encrypting messages of type $\mathsf{Public}(A)$, i.e., messages that require authentication by A. We have to modify the type of snd correspondingly and make use of the slot X_{auth}:

$$\begin{aligned}
&snd:(\{\mathsf{Public}\},\{\mathsf{Public}\})\,\mathsf{KP}(X_{plain},X_{auth})\\
&\text{where } X_{plain} \triangleq (\exists x{:}\mathsf{Public},\exists b{:}\mathsf{Public})[!\mathsf{begun}(x,a,b)] \text{ and } X_{auth} \triangleq \exists a
\end{aligned}$$

Note that the scope of the existential binder for a in X_{auth} includes the occurrence of a in X_{plain}. Now, $snd(m,B)$ has type $\mathsf{Public}(A)$ and $\{\!|snd(m,B)|\!\}_{esA}$ has type $\mathsf{Public}()$.

4.4 Output and Input

The interesting typing rules for processes are for input and output. We assume that every channel is untrusted. Accordingly, in our typing rules for I/O we require that channels are of type Un.[2] Untrusted channels may transmit messages of type Un. In the output rule (Proc Out), message M has to be of type Un in order to be sent out on the untrusted channel N. Note that M may also be sent out if M's type is any other public type, because each public type is a subtype of Un. In the input rule (Proc In), the received message M is assumed to be of type Un. This assumption is used to establish the assertions $\overline{B}$, which are then used as assumptions to check the process continuation P. So $\overline{B}$ is a *post-condition for input* and a *pre-condition for the process continuation*. Note that in (Proc In) we split the non-copyable part of the environment into $\overline{A}_1$ and $\overline{A}_2$. This avoids that the type-checker uses a single non-copyable assertion multiple times.

(Proc Out)

$$\frac{E \vdash N:\mathsf{Un},\, M:\mathsf{Un}}{E \vdash \mathsf{out}\;N\;M}$$

(Proc In)

$$\frac{\vec{x}\cap dom(!E)=\emptyset \quad fv(M)\subseteq dom(\vec{x},!E) \qquad !E,\overline{A}_1,\overline{A}_2 \vdash N:\mathsf{Un} \quad \vec{x},!E,\overline{A}_1,M:\mathsf{Un}\vdash\overline{B} \quad \vec{x},!E,\overline{A}_2,\overline{B}\vdash P}{!E,\overline{A}_1,\overline{A}_2 \vdash \mathsf{inp}\;N\;\exists\vec{x}.\,M[\overline{B}];P}$$

4.5 Encryption

There are two typing rules for encryption. Depending on the kind attributes of the encryption key type, exactly one of these is applicable. The rule (Encrypt Trusted) applies when the encryption key is untainted and the matching decryption key is not public. In this situation, it is always safe to publish the ciphertext, hence, the ciphertext has a public type:

(Encrypt Trusted)

$$\frac{\mathsf{Tainted}\notin K\cup H^{-1} \qquad E\vdash N:(K,H)\,\mathsf{EK}(X),\;(M,L)\in X,\;\overline{B}}{E\vdash\{\!|M|\!\}_N:\mathsf{Public}(L),\;\overline{B}}$$

$$\begin{aligned}
\mathsf{Public}^{-1} &\triangleq \mathsf{Tainted}\\
\mathsf{Tainted}^{-1} &\triangleq \mathsf{Public}\\
K^{-1} &\triangleq \{k^{-1}\mid k\in K\}
\end{aligned}$$

Otherwise, the rule (Encrypt Untrusted) is used for encryption:

[2] In order to model protocols with trusted channels, one could encode these using symmetric encryption.

(Encrypt Untrusted)

$$\frac{\begin{array}{l}\mathsf{Tainted} \in K \cup H^{-1} \quad J = (J' - \{\mathsf{Tainted}\}) \cup (K - \{\mathsf{Public}\}) \\ E \vdash N : (K,H)\,\mathsf{EK}(X),\ (M,L) \in X,\ M : J'\,\mathsf{Top},\ \overline{B}\end{array}}{E \vdash \{\!|M|\!\}_N : J\,\mathsf{Auth}(L),\ \overline{B}}$$

Note that here the ciphertext type $J\,\mathsf{Auth}(L)$ is only public if the plaintext type $J'\,\mathsf{Top}$ is already public, and is tainted if the encryption key is tainted.

4.6 Decryption

There are two typing rules for decryption. Depending on the kind attribute of the ciphertext type, at most one of these is applicable. The rule (Decrypt Trusted) applies if both the ciphertext and the decryption key are untainted. It is the "inverse" of the encryption rule (Encrypt Trusted):

(Decrypt Trusted)

$$\frac{\begin{array}{l}\mathsf{Tainted} \notin H \cup J \\ E \vdash N : (K,H)\,\mathsf{DK}(X) \quad E, (M,L) \in X \vdash \overline{B}\end{array}}{E, \{\!|M|\!\}_{N^{-1}} : J\,\mathsf{Auth}(L) \vdash \overline{B}}$$

The rule (Decrypt Untrusted) applies if the ciphertext is tainted. In this case, we do not know who has encrypted the ciphertext. We therefore instantiate the authentication slot of the decryption key type's pattern parameter X by a fresh eigenvariable x, which acts a placeholder for the "real" authenticator.

(Decrypt Untrusted)

$$\frac{\begin{array}{l}\mathsf{Tainted} \in J \quad x \notin dom(E) \cup fv(\overline{B}) \qquad E \vdash N : (K,H)\,\mathsf{DK}(X) \\ (x, E, (M,x) \in X \vdash \overline{B}) \ \vee \ (\mathsf{Public}, \mathsf{Tainted} \in K \cup H^{-1}) \\ (E, M : J\,\mathsf{Top} \vdash \overline{B}) \ \vee \ (\mathsf{Public} \notin K \cup H^{-1})\end{array}}{E, \{\!|M|\!\}_{N^{-1}} : J\,\mathsf{Top} \vdash \overline{B}}$$

The rule sometimes requires to prove the assertion set $\overline{B}$ in two different environments. This is necessary if $K = \{\mathsf{Public}\}$ and $H = \emptyset$, i.e. if the ciphertext has been encrypted by a public encryption key. Intuitively, the environment $(x, E, (M,x) \in X)$ covers the case when the ciphertext has been formed by an honest agent, and the environment $(E, M : J\,\mathsf{Top})$ when it has been formed by an opponent. If $K = \emptyset$ and $H \subseteq \{\mathsf{Public}\}$ (i.e. the encryption key is a symmetric key or signing key), it is enough to show $\overline{B}$ in environment $(x, E, (M,x) \in X)$. If $K = H = \{\mathsf{Public}\}$ (i.e. the "encryption key" is a tag), it is enough to show $\overline{B}$ in environment $(E, M : J\,\mathsf{Top})$.

4.7 Nonce Types

There are four important typing rules for nonces, which can be classified into responder's and challenger's rules. The rules (Weaken Chall), (Nonce Cast) and (Strengthen Resp) are usefully applied by responders in order to turn challenge types into response types: A responder will first use the rule (Weaken Chall) to turn the type of a nonce challenge into an empty challenge type. In order to do that, he has to be able to offer all assertions that the challenger requested. Then the responder uses the (Nonce Cast) rule to turn the empty challenge type into an empty response type before adding additional offers into the response type's assertion set using (Strengthen Resp).

Responder's Nonce Rules:

(Weaken Chall) (Nonce Cast) (Strengthen Resp)

$$\frac{E \vdash N : (K,H)\,\mathsf{Chall}(\overline{A},B),\ B,\ \overline{C}}{E \vdash N : (K,H)\,\mathsf{Chall}(\overline{A}),\ \overline{C}} \qquad \frac{E \vdash N : (K,H)\,\mathsf{Chall}(),\ \overline{B}}{E \vdash N : (K,H)\,\mathsf{Resp}(),\ \overline{B}} \qquad \frac{E \vdash N : (K,H)\,\mathsf{Resp}(\overline{A}),\ B,\ \overline{C}}{E \vdash N : (K,H)\,\mathsf{Resp}(\overline{A},B),\ \overline{C}}$$

The rule (Nonce Use) is usefully applied by challengers after receiving a response: A challenger may add the assertions from the challenge and response types into his environment. To do that, he has to remove the nonce's freshness assertion from the environment and replace it by an assertion that the nonce has now become stale.

Challenger's Nonce Rule:

(Nonce Use)

$$\frac{E, N : \mathsf{Stale}, \overline{A}, \overline{B} \vdash \overline{C}}{E, \mathsf{fresh}(N : (K,H)\,\mathsf{Chall}(\overline{A})), N : (K,H)\,\mathsf{Resp}(\overline{B}) \vdash \overline{C}}$$

Let's look back at the syntactic restrictions on types: Note first that, by syntactic restriction, the pattern argument of key types may contain copyable assertions only. Thus, without nonce types it is impossible to communicate non-copyable assertions and establish one-to-one correspondences. Note, furthermore, that the syntactic restrictions on challenge and response types prohibit unsafe nonce handshakes. An example of an unsafe handshake is one where both the challenge and response are sent in the clear. In this case, the challenge and response types must both have kind $\{\mathsf{Public}, \mathsf{Tainted}\}$ and then, by syntactic restriction, both types must have empty assertion sets. Therefore, an unsafe nonce handshake of this kind is useless.

Following Gordon and Jeffrey's article [25], we distinguish between POSH ("Public Out Signed Home"), SOPH ("Secret Out Public Home") and SOSH ("Secret Out Secret Home") nonces. Note that, by syntactic restriction, POSH challenge types and SOPH response types must have empty assertion sets.

Derived Forms for Nonce Types:

$$\begin{array}{rcl}
\mathsf{Posh}NT(\overline{A}) & \stackrel{\Delta}{=} & (\{\mathsf{Public}, \mathsf{Tainted}\}, \{\mathsf{Public}\})\,NT(\overline{A}); \\
\mathsf{Soph}NT(\overline{A}) & \stackrel{\Delta}{=} & (\{\mathsf{Tainted}\}, \{\mathsf{Public}, \mathsf{Tainted}\})\,NT(\overline{A}); \\
\mathsf{Sosh}NT(\overline{A}) & \stackrel{\Delta}{=} & (\{\mathsf{Tainted}\}, \{\mathsf{Tainted}\})\,NT(\overline{A})
\end{array}$$

Our treatment of challenge and response types for pattern-matching spi builds on Gordon and Jeffrey's treatment from [25]. We have modified their nonce rules to make nonce casts implicit and to allow the verification of additional protocols:

Implicit nonce casts and nonce checks. Nonce casting and nonce checking in pattern-matching spi is achieved implicitly, rather than by explicit syntactic operators as in [25].

Encrypting SOPH challenges with public keys. In pattern-matching spi, we can verify SOPH challenges that are encrypted by public encryption keys, whereas, in [25], SOPH challenges had to be encrypted by symmetric keys. Technically, we achieve this by additional kind annotations on nonce types and by having freshness assertions keep track of the original nonce challenge types.

Multiple responders. We have split the single nonce cast rule from [25] into the three rules (Nonce Cast), (Weaken Chall) and (Strengthen Resp). As a result, a single nonce challenge may be passed through several responders authenticating each of them. As an example, consider the following fragment of the

Yahalom protocol:[3]

$$\begin{array}{ll}
A \to B & A, N_a \\
B \text{ begins} & \text{“}B \text{ acknowledging receipt of } N_a \text{ to } A\text{”} \\
B \to S & B, \{msg_2(A, N_a)\}_{Kbs} \\
S \text{ begins} & \text{“}S \text{ providing } K_{ab} \text{ to } A \text{ shared with } B\text{”} \\
S \to A & \{msg_3(B, K_{ab}, N_a)\}_{Kas} \\
A \text{ ends} & \text{“}B \text{ acknowledging receipt of } N_a \text{ to } A\text{”} \\
A \text{ ends} & \text{“}S \text{ providing } K_{ab} \text{ to } A \text{ shared with } B\text{”}
\end{array}$$

Here, Alice generates a POSH challenge N_a of type $\mathsf{PoshChall}()$ and sends it to Bob. Bob casts the challenge to an empty response type, strengthens the response by the begun-assertion that he has to offer, and then forwards N_a to server S. When the server receives N_a, it has the following type where $ack(B, N_a, A) =$ “B acknowledging receipt of N_a to A”:

$$\mathsf{PoshResp}(\mathsf{begun}(ack(B, N_a, A)))$$

The server then further strengthens the response, before forwarding it on to Alice. When Alice receives N_a, it has the following type where $prvd(S, K_{ab}, A, B) =$ “S providing K_{ab} to A shared with B”:

$$\mathsf{PoshResp}(\mathsf{begun}(ack(B, N_a, A)), \mathsf{begun}(prvd(S, K_{ab}, A, B)))$$

Now, Alice can safely end both correspondences.

4.8 Secrecy and Integrity

In addition to robust safety for authenticity, our type system also satisfies robust safety theorems for secrecy (“robust write-safety”) and integrity (“robust read-safety”). These theorems formally confirm our informal semantics of public and tainted types from Section 4.2. Like robust safety for authenticity, robust write- and read-safety are simple corollaries of a type preservation theorem. Proving these theorems therefore hardly requires any additional work. They are a byproduct of the proof work that we had to do anyways in order to show robust safety for authenticity.

The robust write-safety theorem says that well-typed processes never leak names that are meant to be kept secret. A name is meant to be kept secret, if at name generation it is annotated by a non-public type that is not a challenge type.[4] For instance, in the following protocol fragment the name n is meant to be kept secret:

$$P_A \ \stackrel{\Delta}{=}\ \mathsf{new}\ n : \mathsf{Secret};\ \mathsf{out}\ net\ \{\!\!\{ \{\!\!\{ sec(n, B) \}\!\!\}_{esA} \}\!\!\}_{epB}$$

Definition 4.1 (Write-Safety) A computation state s is *write-safe* iff $s \to^* \overline{A}, n : T\ :::\ \mathsf{out}\ L\ n \mid P$ implies that T is public or a challenge type. A process P is *write-safe* iff $(\vec{n} : \mathsf{Un}\ :::\ P)$ is write-safe, where $\vec{n} = fn(P)$ and $\vec{n}$ distinct.

Definition 4.2 (Robust Write-Safety) A process P is *robustly write-safe* iff $(P \mid O)$ is write-safe for every closed opponent process O.

Theorem 4.1 (Robust Write-Safety) *If $\vec{n}$ are distinct names and $(\vec{n} : \mathsf{Un} \vdash P)$, then P is robustly write-safe.*

The robust read-safety says that input variables that are annotated by untainted types that are not response types never receive names that have been generated by opponents. Remember that opponents only generate

[3]See Section 6.2 for a typed specification of the BAN–Yahalom protocol.

[4]Challenge types are excluded, because the system is not designed to preserve their secrecy. The rule (Nonce Cast) changes the kind of challenge types. For instance, SOPH nonces are generated as secret challenges and turn into public responses by (Nonce Cast).

names of type Un (by our definition of opponent processes in Section 3.3). For instance, in the following protocol fragment the type annotation Secret on variable x specifies that names received through x have not been generated by opponents:

$$P_B \;\stackrel{\Delta}{=}\; \mathsf{inp}\ \{\!|\{\!|sec(\exists x : \mathsf{Secret}, B)|\!\}_{dsA^{-1}}|\!\}_{dpB^{-1}}\ ;$$

Definition 4.3 (Read-Safety) A computation state s is *read-safe* iff $(s \to^* \overline{A}, n : \mathsf{Un}\ :::\ \mathsf{out}\, L\, M\{x{\leftarrow}n, \vec{y}{\leftarrow}\vec{N}\} \mid (\mathsf{inp}\, L\ \exists x, \vec{y}\,.\, M[x : T, \overline{A}];\ P) \mid Q)$ implies that T is tainted or a response type. A process P is *read-safe* iff $(\vec{n} : \mathsf{Un}\ :::\ P)$ is read-safe, where $\vec{n} = fn(P)$ and $\vec{n}$ distinct.

Definition 4.4 (Robust Read-Safety) A process P is *robustly read-safe* iff $(P \mid O)$ is read-safe for every closed opponent process O.

Theorem 4.2 (Robust Read-Safety) *If $\vec{n}$ are distinct names and $(\vec{n} : \mathsf{Un} \vdash P)$, then P is robustly read-safe.*

4.9 All Typing Rules

After having discussed the most interesting rules we now display the complete set of typing rules:

Right Rules, $E \vdash \overline{A}$:

(True)
$$\frac{}{E \vdash}$$

(Id)
$$\frac{E, !A \vdash \overline{B}}{E, !A \vdash !A, \overline{B}}$$

(Lift)
$$\frac{E \vdash \overline{B}}{E, A \vdash A, \overline{B}}$$

(Derelict)
$$\frac{E \vdash !A, \overline{B}}{E \vdash A, \overline{B}}$$

(Copy)
$$\frac{E \vdash !A, \overline{B}}{E \vdash !A, !A, \overline{B}}$$

(Sub)
$$\frac{E \vdash M : T, \overline{B} \quad T \leq U \quad fv(U) \subseteq dom(E)}{E \vdash M : U, \overline{B}}$$

(Empty)
$$\frac{E \vdash \overline{B}}{E \vdash () : K\,\mathsf{Top}, \overline{B}}$$

(Pair)
$$\frac{E \vdash M : K\,\mathsf{Top}, N : K\,\mathsf{Top}, \overline{B}}{E \vdash (M, N) : K\,\mathsf{Top}, \overline{B}}$$

(Enc Part)
$$\frac{E \vdash M : (K, H)\,\mathsf{KP}(X), \overline{B}}{E \vdash \mathsf{Enc}\,(M) : (K, H)\,\mathsf{EK}(X), \overline{B}}$$

(Encrypt Trusted)
$$\frac{\begin{array}{l}\mathsf{Tainted} \notin K \cup H^{-1}\\ E \vdash N : (K, H)\,\mathsf{EK}(X),\ (M, L) \in X, \overline{B}\end{array}}{E \vdash \{\!|M|\!\}_N : \mathsf{Public}(L), \overline{B}}$$

(Dec Part)
$$\frac{E \vdash M : (K, H)\,\mathsf{KP}(X), \overline{B}}{E \vdash \mathsf{Dec}\,(M) : (K, H)\,\mathsf{DK}(X), \overline{B}}$$

(Encrypt Untrusted)
$$\frac{\begin{array}{l}\mathsf{Tainted} \in K \cup H^{-1} \quad J = (J' - \{\mathsf{Tainted}\}) \cup (K - \{\mathsf{Public}\})\\ E \vdash N : (K, H)\,\mathsf{EK}(X),\ (M, L) \in X,\ M : J'\,\mathsf{Top}, \overline{B}\end{array}}{E \vdash \{\!|M|\!\}_N : J\,\mathsf{Auth}(L), \overline{B}}$$

(Nonce Cast)
$$\frac{E \vdash N : (K, H)\,\mathsf{Chall}(), \overline{B}}{E \vdash N : (K, H)\,\mathsf{Resp}(), \overline{B}}$$

(Weaken Chall)
$$\frac{E \vdash N : (K, H)\,\mathsf{Chall}(\overline{A}, B), B, \overline{C}}{E \vdash N : (K, H)\,\mathsf{Chall}(\overline{A}), \overline{C}}$$

(Strengthen Resp)
$$\frac{E \vdash N : (K, H)\,\mathsf{Resp}(\overline{A}), B, \overline{C}}{E \vdash N : (K, H)\,\mathsf{Resp}(\overline{A}, B), \overline{C}}$$

Left Rules, $E, \overline{A} \vdash \overline{B}$:

(Unsub)
$$\frac{E, M : U \vdash \overline{B} \quad T \leq U \quad fv(U) \subseteq dom(E)}{E, M : T \vdash \overline{B}}$$

(Split)
$$\frac{E, M : K\,\mathsf{Top}, N : K\,\mathsf{Top} \vdash \overline{B}}{E, (M, N) : K\,\mathsf{Top} \vdash \overline{B}}$$

(Decrypt Untrusted)
$$\frac{\begin{array}{l}\mathsf{Tainted} \in J \quad x \notin dom(E) \cup fv(\overline{B}) \qquad E \vdash N : (K, H)\,\mathsf{DK}(X)\\ (x, E, (M, x) \in X \vdash \overline{B}) \ \vee\ (\mathsf{Public}, \mathsf{Tainted} \in K \cup H^{-1})\\ (E, M : J\,\mathsf{Top} \vdash \overline{B}) \ \vee\ (\mathsf{Public} \notin K \cup H^{-1})\end{array}}{E, decrypt(M, N) : J\,\mathsf{Top} \vdash \overline{B}}$$

(Decrypt Trusted)
$$\frac{\begin{array}{l}\mathsf{Tainted} \notin H \cup J\\ E \vdash N : (K, H)\,\mathsf{DK}(X) \quad E, (M, L) \in X \vdash \overline{B}\end{array}}{E, decrypt(M, N) : J\,\mathsf{Auth}(L) \vdash \overline{B}}$$

$$\text{where } decrypt(M,N) \triangleq \begin{cases} \{\!|M|\!\}_{\mathsf{Enc}\,(L)} & \text{if } N = \mathsf{Dec}\,(L) \\ \{\!|M|\!\}_{N^{-1}} & \text{otherwise} \end{cases}$$

(Nonce Use)
$$\frac{E, N : \mathsf{Stale}, \overline{A}, \overline{B} \vdash \overline{C}}{E, \mathsf{fresh}(N : (K,H)\,\mathsf{Chall}(\overline{A})), N : (K,H)\,\mathsf{Resp}(\overline{B}) \vdash \overline{C}}$$

(Discard Chall)
$$\frac{E, N : \mathsf{Stale} \vdash \overline{B}}{E, \mathsf{fresh}(N : (K,H)\,\mathsf{Chall}(\overline{A})) \vdash \overline{B}}$$

Well-typed Processes, $E \vdash P$:

(Proc Out)
$$\frac{E \vdash N : \mathsf{Un}, M : \mathsf{Un}}{E \vdash \mathsf{out}\ N\ M}$$

(Proc In)
$$\frac{\begin{array}{c}\vec{x} \cap dom(!E) = \emptyset \quad fv(M) \subseteq dom(\vec{x}, !E) \\ !E, \overline{A}_1, \overline{A}_2 \vdash N : \mathsf{Un} \quad \vec{x}, !E, \overline{A}_1, M : \mathsf{Un} \vdash \overline{B} \quad \vec{x}, !E, \overline{A}_2, \overline{B} \vdash P\end{array}}{!E, \overline{A}_1, \overline{A}_2 \vdash \mathsf{inp}\ N\ \exists \vec{x}.\, M[\overline{B}]; P}$$

(Proc New)
$$\frac{E, n : T, \mathsf{fresh}(n : T) \vdash P \quad n \notin fn(E),\ T \text{ generative}}{E \vdash \mathsf{new}\ n{:}T;\ P}$$

(Proc Par)
$$\frac{!E, \overline{A}_1 \vdash P \quad !E, \overline{A}_2 \vdash Q}{!E, \overline{A}_1, \overline{A}_2 \vdash P \mid Q}$$

(Proc Repl)
$$\frac{!E \vdash P}{!E, \overline{A} \vdash\ !P}$$

(Proc Stop)
$$\frac{}{E \vdash \mathbf{0}}$$

(Proc Begin)
$$\frac{E, \mathsf{begun}(M) \vdash P}{E \vdash \mathsf{begin}(M);\ P}$$

(Proc Begin Many)
$$\frac{E, !\mathsf{begun}(M) \vdash P}{E \vdash \mathsf{begin!}(M);\ P}$$

(Proc End)
$$\frac{!E, \overline{A}_1 \vdash \mathsf{begun}(M) \quad !E, \overline{A}_2 \vdash P}{!E, \overline{A}_1, \overline{A}_2 \vdash \mathsf{end}(M);\ P}$$

5 Derived Forms

In the previous section, we have introduced the core language and type system for pattern-matching spi. In this section, we define the customized derived forms that we work with when specifying and checking protocols. We accompany each of the derived forms by derived typing rules that are easier to work with than the core language rules. These derived rules are sometimes a bit weaker than the core language rules, but we think that they are sufficient to type-check most practical protocols that can be checked with the core language rules directly.

5.1 Tagging

In previous type systems for cryptographic protocols [24, 25, 23], message tags were introduced using *tagged union types*. These types are sound, and they allow a key to be used in more than one protocol, but they require the protocol suite to be known *before* the key is generated, since the plaintext type of the key is given as the tagged union of all the messages in the protocol suite. In this paper, we adopt a variant of *dynamic types* to allow a key to be generated with no knowledge of the protocol suite it will be used for. In our system, we give message tags a type of the form $\ell : X \rightarrow \mathsf{Auth}(Y)$, which can be used to tag messages M of kind $(J \cup \mathsf{Tainted})$ to get tagged messages $\ell(M) : J\,\mathsf{Auth}(L)$, if (M, L) matches the pattern (X, Y). For example:

Example 5.1 (Simple Signing)

$$snd : (\exists x : \mathsf{Public}, \exists b : \mathsf{Public})[!\mathsf{begun}(x,a,b)] \rightarrow \mathsf{Auth}(\exists a : \mathsf{Public}, \_)$$
$$P \triangleq \mathsf{new}\ k : \mathsf{SigningKP}(A);\ (!\mathsf{out}\ net\ (\mathsf{Dec}\,(k)) \mid !P_A(\mathsf{Enc}\,(k)) \mid !P_B(\mathsf{Dec}\,(k)))$$
$$P_A(esA : \mathsf{SigningEK}(A)) \triangleq \mathsf{new}\ m : \mathsf{Public};\ \mathsf{begin!}(m,A,B);\ \mathsf{out}\ net\ \{\!|snd(m,B)|\!\}_{esA}$$
$$P_B(dsA : \mathsf{SigningDK}(A)) \triangleq \mathsf{inp}\ net\ \{\!|snd(\exists x : \mathsf{Public}, B)|\!\}_{dsA^{-1}}[!\mathsf{begun}(x,A,B)];\ \mathsf{end}(x,A,B)$$

In the type of *snd*, the scope of $\exists a$ includes the entire tag type. Thus, the a inside the begun-assertion is bound by the $\exists a$ on the right.

Remember that in pattern-matching spi-calculus, tagging is not primitive but defined as symmetric encryption with public keys, i.e., $\ell(M) \triangleq \{\!|M|\!\}_{\mathsf{Enc}(\ell)}$. We define tag types by the following translation to our core language of types.

$$X \to \mathsf{Auth}(Y) \ \triangleq\ (\{\mathsf{Public}\}, \{\mathsf{Public}\})\,\mathsf{KP}(X,Y)$$

We obtain the following derived typing rules for tagging:

(Tag)
$$\frac{E \vdash \ell : X \to \mathsf{Auth}(Y), M : (J \cup \{\mathsf{Tainted}\})\,\mathsf{Top}, (M,L) \in (X,Y), \overline{B}}{E \vdash \ell(M) : J\,\mathsf{Auth}(L), \overline{B}}$$

(Untag Untainted)
$$\frac{\mathsf{Tainted} \notin J \quad E \vdash \ell : X \to \mathsf{Auth}(Y) \quad E, (M,L) \in (X,Y) \vdash \overline{B}}{E, \ell(M) : J\,\mathsf{Auth}(L) \vdash \overline{B}}$$

(Untag Tainted)
$$\frac{\mathsf{Tainted} \in J \quad E \vdash \ell : X \to \mathsf{Auth}(Y) \quad E, M : J\,\mathsf{Top} \vdash \overline{B}}{E, \ell(M) : J\,\mathsf{Top} \vdash \overline{B}}$$

5.2 Signing Keys

A goal of this type system is to allow principals to have just one signing key, which can be used for any protocol, rather than requiring different signing key types for different protocols. Message tags are then used to ensure the correctness of each protocol.

The type for a signing key is designed to support nested signatures, for example $\{\!|\{\!|M|\!\}_{esA}|\!\}_{esB}$ is a message M signed by A (using her signing key $esA : \mathsf{SigningEK}(A)$) and B (using his signing key $esB : \mathsf{SigningEK}(B)$). This message can be given type $\{\!|\{\!|M|\!\}_{esA}|\!\}_{esB} : \mathsf{Secret}$ as long as $M : \mathsf{Secret}(A,B,y)$ for some y, and type $\{\!|\{\!|M|\!\}_{esA}|\!\}_{esB} : \mathsf{Public}$ as long as $M : \mathsf{Public}(A,B,y)$ for some y. This form of nested signing was not supported by [24, 25, 23].

$$\mathsf{Signing}KT(L) \ \triangleq\ (\emptyset, \{\mathsf{Public}\})\,KT(\mathsf{Secret}(L,y), \exists y)$$

The derived type rules for signing keys are:

(Sign)
$$\frac{\mathsf{Tainted} \notin J \quad E \vdash N : \mathsf{SigningEK}(L), M : J\,\mathsf{Auth}(L,L'), \overline{B}}{E \vdash \{\!|M|\!\}_N : J\,\mathsf{Auth}(L'), \overline{B}}$$

(Unsign Untainted)
$$\frac{\mathsf{Tainted} \notin J \quad E \vdash N : \mathsf{SigningDK}(L) \quad E, M : \mathsf{Secret}(L,L') \vdash \overline{B}}{E, \{\!|M|\!\}_{N^{-1}} : J\,\mathsf{Auth}(L') \vdash \overline{B}}$$

(Unsign Tainted)
$$\frac{\mathsf{Tainted} \in J \quad E \vdash N : \mathsf{SigningDK}(L) \quad x, E, M : \mathsf{Secret}(L,x) \vdash \overline{B}}{E, \{\!|M|\!\}_{N^{-1}} : J\,\mathsf{Top} \vdash \overline{B}}$$

These type rules are enough to verify Example 5.1: The critical bit of the sender's type derivation is the proof that $\{\!|snd(m,B)|\!\}_{esA}$ has type Un. The critical bit of the receiver's type derivation is the proof that the post-condition $!\mathsf{begun}(x,A,B)$ for the pattern-matching input holds. These parts of the type derivation are sketched in Figure 1. The (shortcut) in this figure merges applications of the rule for pattern-matching, the pair formation rule (Pair) and the subsumption rule (Sub).

$X \triangleq (\exists x : \mathsf{Public}, \exists b : \mathsf{Public})[!\mathsf{begun}(x,a,b)]$, $Y \triangleq (\exists a : \mathsf{Public}, \_)$, $E_0 \triangleq (\mathit{net} : \mathsf{Un}, A : \mathsf{Public}, B : \mathsf{Public}, \mathit{snd} : X \to \mathsf{Auth}(Y))$, $E_{snd} \triangleq (E_0, \mathit{esA} : \mathsf{SigningEK}(A), m : \mathsf{Public}, !\mathsf{begun}(m,A,B))$, $E_{rcv} \triangleq (E_0, \mathit{dsA} : \mathsf{SigningDK}(A))$, $\bar{C}(x) \triangleq (x : \mathsf{Public}, !\mathsf{begun}(x,A,B))$

Checking that $\{\!|snd(m,B)|\!\}_{esA}$ is publishable:

$$\dfrac{\dfrac{}{E_{snd} \vdash \mathit{esA} : \mathsf{SigningEK}(A)}\,(\text{Id}) \qquad \dfrac{\dfrac{\dfrac{}{E_{snd} \vdash m,B,A : \mathsf{Public}, !\mathsf{begun}(m,A,B)}\,(\text{Id})}{E_{snd} \vdash (m,B) : \mathsf{Un}, ((m,B),(A,())) \in (X,Y)}\,(\text{shortcut})}{E_{snd} \vdash snd(m,B) : \mathsf{Public}(A,())}\,(\text{Tag})}{\dfrac{E_{snd} \vdash \{\!|snd(m,B)|\!\}_{esA} : \mathsf{Public}()}{E_{snd} \vdash \{\!|snd(m,B)|\!\}_{esA} : \mathsf{Un}}\,(\text{Sub})}\,(\text{Sign})$$

Checking the input post-condition $\bar{C}(x)$:

$$\dfrac{\dfrac{}{x, E_{rcv} \vdash \mathit{dsA} : \mathsf{SigningDK}(A)}\,(\text{Id}) \qquad \dfrac{\dfrac{\dfrac{}{y,x,E_{rcv}, x,B,A : \mathsf{Public}, !\mathsf{begun}(x,A,B) \vdash \bar{C}(x)}\,(\text{Id})}{y,x,E_{rcv}, ((x,B),(A,y)) \in (X,Y) \vdash \bar{C}(x)}\,(\text{pattern match})}{y,x,E_{rcv}, snd(x,B) : \mathsf{Secret}(A,y) \vdash \bar{C}(x)}\,(\text{Untag Untainted})}{x, E_{rcv}, \{\!|snd(x,B)|\!\}_{dsA^{-1}} : \mathsf{Un} \vdash \bar{C}(x)}\,(\text{Unsign Tainted})$$

Figure 1: Important parts of the type derivation for Example 5.1

5.3 Public Encryption Keys

Public key encryption is dual to signing: the encryption key is public, and the decryption key is kept secret. One crucial difference is that although our type system supports nested uses of signatures, it does not support similar nested uses of public-key encryption. As a result, the type system does not support encrypt-then-sign applications. It is well-known that encrypt-then-sign protocols are problematic, because a receiver of an encrypted-then-signed message cannot be sure that the signature is the original—a compromised principal may have removed the original signature and replaced it by his own. This is a well-known problem that results in security flaws for some encrypt-then-sign applications (see, for instance, the analysis of the CCITT X.509 protocol in [15]).

$$\mathsf{PublicCrypto}KT(L) \triangleq (\{\mathsf{Public}\}, \emptyset)\,KT(\mathsf{Secret}(L), \_)$$

The derived type rules for public encryption keys are:

(Pub Encrypt)
$$\dfrac{E \vdash N : \mathsf{PublicCryptoEK}(L),\ M : \mathsf{Secret}(L),\ \bar{B}}{E \vdash \{\!|M|\!\}_N : \mathsf{Public},\ \bar{B}}$$

(Prv Decrypt Tainted)
$$\dfrac{\mathsf{Tainted} \in J \quad E \vdash N : \mathsf{PublicCryptoDK}(L) \quad E, M : J\,\mathsf{Top} \vdash \bar{B} \quad E, M : \mathsf{Secret}(L) \vdash \bar{B}}{E, \{\!|M|\!\}_{N^{-1}} : J\,\mathsf{Top} \vdash \bar{B}}$$

(Prv Decrypt Untainted)
$$\dfrac{\mathsf{Tainted} \notin J \quad E \vdash N : \mathsf{PublicCryptoDK}(L) \quad E, M : \mathsf{Secret}(L) \vdash \bar{B}}{E, \{\!|M|\!\}_{N^{-1}} : J\,\mathsf{Auth}(L') \vdash \bar{B}}$$

The rule (Prv Decrypt Tainted) requires to prove assertions $\bar{B}$ in two different environments. This is necessary, because a tainted ciphertext under a public encryption key may have been encrypted either by an opponent or by an honest agent. The first premise of (Prv Decrypt Tainted) accounts for the former possibility and the second premise for the latter. We can now verify the sign-then-encrypt protocol:

$X \triangleq (\exists x : \mathsf{Secret}, \exists b : \mathsf{Public})[!\mathsf{begun}(x,a,b)]$, $Y \triangleq (\exists a : \mathsf{Public}, \_)$, $SEK(L) \triangleq \mathsf{SigningEK}(L)$, $SDK(L) \triangleq \mathsf{SigningDK}(L)$, $PEK(L) \triangleq \mathsf{PublicCryptoEK}(L)$, $PDK(L) \triangleq \mathsf{PublicCryptoDK}(L)$, $E_0 \triangleq (net : \mathsf{Un}, A, B : \mathsf{Public}, sec : X \to \mathsf{Auth}(Y))$, $E_{snd} \triangleq (E_0, esA : SEK(A), epB : PEK(B), m : \mathsf{Secret}, !\mathsf{begun}(m,A,B))$, $E_{rcv} \triangleq (E_0, dsA : SDK(A), dpB : PDK(B))$, $\bar{C}(x) \triangleq (x : \mathsf{Secret}, !\mathsf{begun}(x,A,B))$

Checking that $\{\!|\{\!|sec(m,B)|\!\}_{esA}|\!\}_{epB}$ is publishable:

$$\dfrac{\dfrac{}{E_{snd} \vdash epB : PEK(B)} \qquad \dfrac{\dfrac{}{E_{snd} \vdash esA : SEK(A)} \qquad \dfrac{\dfrac{\dfrac{}{E_{snd} \vdash m : \mathsf{Secret}, B,A : \mathsf{Public}, !\mathsf{begun}(m,A,B)}}{E_{snd} \vdash (m,B) : \mathsf{Tainted}, ((m,B),(A,B)) \in (X,Y)}\text{(shortcut)}}{E_{snd} \vdash sec(m,B) : \mathsf{Secret}(A,B)}\text{(Tag)}}{E_{snd} \vdash \{\!|sec(m,B)|\!\}_{esA} : \mathsf{Secret}(B)}\text{(Sign)}}{E_{snd} \vdash \{\!|\{\!|sec(m,B)|\!\}_{esA}|\!\}_{epB} : \mathsf{Un}}\text{(Sub),(Pub Encrypt)}$$

Checking the input post-condition $\bar{C}(x)$:

$$\dfrac{\dfrac{}{x, E_{rcv} \vdash dpB : PDK(B)} \qquad \mathcal{D}_{untrusted} \qquad \mathcal{D}_{trusted}}{x, E_{rcv}, \{\!|\{\!|sec(x,B)|\!\}_{dsA^{-1}}|\!\}_{dpB^{-1}} : \mathsf{Un} \vdash \bar{C}(x)}\text{(Prv Decrypt Tainted)}$$

Subderivation $\mathcal{D}_{untrusted}$:

$$\dfrac{\dfrac{}{x, E_{rcv} \vdash dsA : SDK(A)} \qquad \dfrac{\dfrac{\dfrac{}{y, x, E_{rcv}, x : \mathsf{Secret}, B, A : \mathsf{Public}, !\mathsf{begun}(x,A,B) \vdash \bar{C}(x)}\text{(Id)}}{y, x, E_{rcv}, ((x,B),(A,y)) \in (X,Y) \vdash \bar{C}(x)}\text{(pattern match)}}{y, x, E_{rcv}, sec(x,B) : \mathsf{Secret}(A,y) \vdash \bar{C}(x)}\text{(Untag Untainted)}}{x, E_{rcv}, \{\!|sec(x,B)|\!\}_{dsA^{-1}} : \mathsf{Un} \vdash \bar{C}(x)}\text{(Unsign Tainted)}$$

Subderivation $\mathcal{D}_{trusted}$:

$$\dfrac{\dfrac{}{x, E_{rcv} \vdash dsA : SDK(A)} \qquad \dfrac{\dfrac{\dfrac{}{x, E_{rcv}, x : \mathsf{Secret}, B, A : \mathsf{Public}, !\mathsf{begun}(x,A,B) \vdash \bar{C}(x)}\text{(Id)}}{x, E_{rcv}, ((x,B),(A,B)) \in (X,Y) \vdash \bar{C}(x)}\text{(pattern match)}}{x, E_{rcv}, sec(x,B) : \mathsf{Secret}(A,B) \vdash \bar{C}(x)}\text{(Untag Untainted)}}{x, E_{rcv}, \{\!|sec(x,B)|\!\}_{dsA^{-1}} : \mathsf{Secret}(B) \vdash \bar{C}(x)}\text{(Unsign Untainted)}$$

Figure 2: Important parts of the type derivation for Example 5.2

Example 5.2 (Sign-Then-Encrypt)

$$\begin{aligned}
&sec : (\exists x : \mathsf{Secret}, \exists b : \mathsf{Public})[!\mathsf{begun}(x,a,b)] \to \mathsf{Auth}(\exists a : \mathsf{Public}, \_)\\
&P \;\triangleq\; \mathsf{new}\ k : \mathsf{SigningKP}(A);\ \mathsf{new}\ h : \mathsf{PublicCryptoKP}(B);\\
&\quad (!\mathsf{out}\ net\ (\mathsf{Dec}(k), \mathsf{Enc}(h)) \mid !P_A(\mathsf{Enc}(k)) \mid !P_B(\mathsf{Dec}(k)))\\
&P_A(esA : \mathsf{SigningEK}(A), epB : \mathsf{PublicCryptoEK}(B)) \;\triangleq\\
&\quad \mathsf{new}\ m : \mathsf{Secret};\ \mathsf{begin}!(m,A,B);\ \mathsf{out}\ net\ \{\!|\{\!|sec(m,B)|\!\}_{esA}|\!\}_{epB}\\
&P_B(dsA : \mathsf{SigningDK}(A), dpB : \mathsf{PublicCryptoDK}(B)) \;\triangleq\\
&\quad \mathsf{inp}\ net\ \{\!|\{\!|sec(\exists x : \mathsf{Secret}, B)|\!\}_{dsA^{-1}}|\!\}_{dpB^{-1}}[!\mathsf{begun}(x,A,B)];\ \mathsf{end}(x,A,B)
\end{aligned}$$

The most important parts of the type derivation are sketched in Figure 2. In the derivation of $E_{snd} \vdash \{\!|\{\!|sec(m,B)|\!\}_{esA}|\!\}_{epB} : \mathsf{Un}$, note that $\{\!|sec(m,B)|\!\}_{esA}$ has type $\mathsf{Secret}(B)$. This was impossible in [24, 25, 23], where all ciphertexts have type Un. This limitation of [24, 25, 23] is the reason why sign-then-encrypt fails to type-check in these earlier systems.

5.4 Symmetric Cryptography

Recall that symmetric encryption is not a primitive but defined as asymmetric encryption with a shared, secret key pair, i.e., $\{M\}_N \triangleq \{\!|M|\!\}_{\mathsf{Enc}(N)}$. We define a type for symmetric keys as follows:

$$\mathsf{SymK}(L) \triangleq (\emptyset, \emptyset)\,\mathsf{KP}(\mathsf{Secret}(L), \_)$$

The derived type rules are:

$$\text{(Sym Encrypt)}\quad \frac{E \vdash N : \mathsf{SymK}(L), M : \mathsf{Secret}(L), \overline{B}}{E \vdash \{M\}_N : \mathsf{Public}, \overline{B}} \qquad \text{(Sym Decrypt)}\quad \frac{E \vdash N : \mathsf{SymK}(L) \quad E, M : \mathsf{Secret}(L) \vdash \overline{B}}{E, \{M\}_N : \mathsf{Un} \vdash \overline{B}}$$

The key type parameter L may be instantiated by the tuple of all principals that share the symmetric key. It is also legal to include the symmetric key itself in L. (Name declarations of the form $n : T(n)$ are legal.) Including a symmetric key in the parameter of its own key type is sometimes useful, for instance, when the reception of a new session key is acknowledged by encrypting a nonce with the session key itself. In Section 6.2, we present the types for the BAN–Yahalom protocol. These types instantiate the parameter L in that way.

5.5 Hashing

Recall that we define hashing as encryption with a public encryption key that has no matching decryption key. The encrypted message is then tagged by a special *hashtag*, which is needed to alert the type-checker to treat the *hashtag*ed message in a special way. Our definition of hashing was $\#(M) \triangleq \mathit{hashtag}(\{\!|M|\!\}_{\mathit{hashkey}})$. Here are our types for *hashkey* and *hashtag*:

$$\mathit{hashkey} : (\{\mathsf{Public}\}, \emptyset)\,\mathsf{EK}(\_); \quad \mathit{hashtag} : \{\!|\mathsf{Secret}(y)|\!\}_{\mathit{hashkey}} \to \mathsf{Auth}(\exists y)$$

The derived type rules for hashing are:

$$\text{(Hash)}\quad \frac{E \vdash M : \mathsf{Secret}(L), \overline{B}}{E \vdash \#(M) : \mathsf{Public}(L), \overline{B}} \qquad \text{(Unhash)}\quad \frac{E, M : \mathsf{Secret}(L) \vdash \overline{B}}{E, \#(M) : \mathsf{Secret}(L) \vdash \overline{B}}$$

We assume that each environment E is implicitly extended by the above type assertions for the special global names *hashkey* and *hashtag*. We can then adapt Example 5.1 to allow A to sign the message digest of M rather than signing the entire message:

Example 5.3 (Message Digest)

$$\begin{array}{ll} A \text{ begins! } (M,A,B) & \\ A \to B & M, \{\!|\#(\mathit{snd}(M,B))|\!\}_{\mathit{esA}} \\ B \text{ ends } (M,A,B) & \end{array}$$

This example uses the same types as Example 5.1.

5.6 Keyed Hashing

Keyed hashing is commonly used in message authentication codes. Recall that we encode keyed hashing as a composite of hashing and symmetric cryptography, i.e., $\#_N(M) \triangleq \{\#(M)\}_N$. Here is our definition of types for the hash keys:

$$\mathsf{HashKey}(L) \;\triangleq\; (\emptyset,\emptyset)\,\mathsf{KP}(\#(\mathsf{Secret}(L)),\_)$$

The derived typing rules are:

$$\text{(Keyed Hash)}\quad \frac{E \vdash N : \mathsf{HashKey}(L), M : \mathsf{Secret}(L), \overline{B}}{E \vdash \#_N(M) : \mathsf{Public}, \overline{B}} \qquad \text{(Keyed Unhash)}\quad \frac{E \vdash N : \mathsf{HashKey}(L) \quad E, M : \mathsf{Secret}(L) \vdash \overline{B}}{E, \#_N(M) : \mathsf{Un} \vdash \overline{B}}$$

We can now type-check the following example:

Example 5.4 (Keyed Hashing)

$$\begin{array}{ll} A \text{ begins! } (M,A,B) & \\ A \to B & M, \#_k(snd(M)) \\ B \text{ ends } (M,A,B) & \end{array}$$

Consider the types $k : \mathsf{HashKey}(A,B)$ and $snd : (\exists x : \mathsf{Public})[!\mathsf{begun}(x,a,b)] \to \mathsf{Auth}(\exists a : \mathsf{Public}, \exists b : \mathsf{Public})$. With these types, we can type-check a system that restricts each principal to act in only one of the two roles. A bit more precisely: If $Snd_1(a,b,k)$ and $Rcv_1(b,a,k)$ are the obvious spi-calculus implementations of the sender process on behalf of a and the receiver process on behalf of b, then the following system type-checks:

$$\mathsf{new}\ k : \mathsf{HashKey}(A,B); (!Snd_1(A,B,k) \mid !Rcv_1(B,A,k))$$

If we want to allow each principal to act in both roles, the above protocol is unsafe. An opponent can simply play a message M from A straight back to her, making her believe M originated from B. Note that it is impossible to type-check $\mathsf{new}\ k : \mathsf{HashKey}(A,B); (!Rcv_1(B,A,k) \mid !Rcv_1(A,B,k))$.

We can modify the protocol by including the principal identities in the message:

$$\begin{array}{ll} A \text{ begins! } (M,A,B) & \\ A \to B & M, \#_k(snd(M,A,B)) \\ B \text{ ends } (M,A,B) & \end{array}$$

Now, if $Snd_2(a,b,k)$ and $Rcv_2(b,a,k)$ are the spi-calculus implementations for the modified sender and receiver processes, we can type-check the following system where each principal acts both as a sender and a receiver.

$$\mathsf{new}\ k : \mathsf{HashKey}(); (!Snd_2(A,B,k) \mid !Snd_2(B,A,k) \mid !Rcv_2(A,B,k) \mid !Rcv_2(B,A,k))$$

The type for the snd-tag is $(\exists x : \mathsf{Public}, \exists a : \mathsf{Public}, \exists b : \mathsf{Public})[!\mathsf{begun}(x,a,b)] \to \mathsf{Auth}()$.

5.7 Matching

In pattern-matching spi, we have replaced multiple destructors from other spi-calculi by a single pattern-matching input primitive. This primitive can be used to directly destruct a message as part of an input statement. Sometimes, however, pattern-matching at the initial input is not possible, because constants needed for the pattern-match only become known later in the protocol. Examples are protocols where a receiver first obtains a ciphertext and only later gets to know the decryption key. In such a case, decryption of the ciphertext cannot be part of the initial pattern-matching input. It is also sometimes necessary to bind a ciphertext to a variable before destroying the ciphertext. This is necessary if the ciphertext is still needed later, like for instance in the following protocol.

Example 5.5 (Signed Message Digest with Acknowledgment) *Suppose esA and esB are A and B's signing keys.*

$$\begin{array}{ll} A \text{ begins! } (snd,M,A,B) & \\ A \to B & A,\ M,\ \{\!|\#(snd(M,B))|\!\}_{esA} \\ B \text{ ends } (snd,M,A,B) & \\ B \text{ begins! } (ack,M,A,B) & \\ B \to A & \{\!|ack(\{\!|\#(snd(M,B))|\!\}_{esA},\ A)|\!\}_{esB} \\ A \text{ ends } (ack,M,A,B) & \end{array}$$

Note that B's acknowledgment contains A's signed message digest. If B does not save A's signed message digest before decrypting it, he will not be able to construct his acknowledgment.

It is not hard to encode general pattern-matching, using pattern-matching input and symmetric encryption. The derived form $(\mathsf{match}\ M\ \mathsf{is}\ X;\, P)$ attempts to match message M against pattern X and then executes P under the resulting substitution:

$$\mathsf{match}\ M\ \mathsf{is}\ \exists\vec{x}\,.\,N[\overline{A}];\, P \quad \overset{\Delta}{=} \quad \begin{array}{l}\mathsf{new}\ c : \mathsf{Un};\ \mathsf{new}\ k : (\emptyset,\emptyset)\,\mathsf{KP}(M);\\ \quad \mathsf{out}\ c\ \{M\}_k;\ \mathsf{inp}\ c\ \exists\vec{x}\,.\,\{N\}_k[\overline{A},\, \{N\}_k : \mathsf{Un}];\ P\end{array}$$

We obtain the following derived typing rule:

$$\frac{\begin{array}{c}\vec{x} \cap dom(!E) = \emptyset \quad fv(N) \subseteq dom(\vec{x}, !E)\\ \vec{x}, !E, \overline{B}_1, N \in M \vdash \overline{A} \quad \vec{x}, !E, \overline{B}_2, \overline{A}, N \in M \vdash P\end{array}}{!E, \overline{B}_1, \overline{B}_2 \vdash \mathsf{match}\ M\ \mathsf{is}\ \exists\vec{x}\,.\,N[\overline{A}];\, P}$$

We can use this derived form to express B's side of Example 5.5:

$$P_B(esB : \mathsf{SigningEK}(B),\, dsA : \mathsf{SigningDK}(A)) \overset{\Delta}{=} \begin{array}{l}\mathsf{inp}\ net\ (a : \mathsf{Un}, x : \mathsf{Un}, ctext : \mathsf{Un});\\ \mathsf{match}\ ctext\ \mathsf{is}\ \{\!|\#(snd(x,B))|\!\}_{dsA^{-1}};\\ \mathsf{end}(snd, x, a, B);\\ \mathsf{begin!}(ack, x, a, B);\\ \mathsf{out}\ net\ \{\!|ack(ctext, a)|\!\}_{esB}\end{array}$$

This protocol is interesting because the type-checker needs our general definition of pattern-matching on the left (see Section 4.1). Under the less general alternative definition presented in Section 4.1, which uses matching instead of unification, this protocol does not type-check. We will explain why the more general definition is needed here: The protocol type-checks with the following tag type.

$$\begin{array}{l} ack : X(b) \rightarrow \mathsf{Auth}\langle \exists b : \mathsf{Public}\rangle\\ \text{where } X(b) \ \overset{\Delta}{=}\ (\{\!|\#(snd(\exists x : \mathsf{Public},\, \_))|\!\}_{\_^{-1}},\, \exists a : \mathsf{Public})[!\mathsf{begun}(ack, x, a, b)]\end{array}$$

In order to type-check the output statement in P_B's last line, it is necessary to prove a judgment of the following form.

$$E,\, ctext : \mathsf{Un},\, ctext \in \{\!|\#(snd(x,B))|\!\}_{dsA^{-1}},\, !\mathsf{begun}(ack, x, a, B) \vdash \{\!|ack(ctext, a)|\!\}_{esB} : \mathsf{Un}$$

Using (Sign) and (Tag), this proof obligation gives rise to the following subgoal.

$$E,\, ctext : \mathsf{Un},\, ctext \in \{\!|\#(snd(x,B))|\!\}_{dsA^{-1}},\, !\mathsf{begun}(ack, x, a, B) \vdash \langle (ctext, a),\, B\rangle \in \langle X(b),\, \exists b : \mathsf{Public}\rangle$$

At this point, we would be stuck if our left pattern-matching rule did not use unification, because $ctext$ does not match the constant pattern $\{\!|\#(snd(x,B))|\!\}_{dsA^{-1}}$. Fortunately, our left pattern-matching rule allows us to unify $ctext$ and $\{\!|\#(snd(x,B))|\!\}_{dsA^{-1}}$ and then attempt to prove the judgment under the most general unifier. Obviously, the most general unifier is $(ctext \leftarrow \{\!|\#(snd(x,B))|\!\}_{dsA^{-1}})$. Applying this unifier to the above proof goal yields the following new goal:

$$E,\, \{\!|\#(snd(x,B))|\!\}_{dsA^{-1}} : \mathsf{Un},\, !\mathsf{begun}(ack, x, a, B) \vdash \langle (\{\!|\#(snd(x,B))|\!\}_{dsA^{-1}},\, a),\, B\rangle \in \langle X(b),\, \exists b : \mathsf{Public}\rangle$$

This new goal is derivable, assuming that E contains the type assertion $B : \mathsf{Public}$.

6 More Examples

In the previous section, we have seen typing rules for checking sign-then-encrypt, nested signing, hashing and keyed hashing. None of these constructs could be verified by Gordon and Jeffrey's earlier type systems, nor (to the best of our knowledge) by other type systems for cryptographic protocols. In particular, Examples 5.2, 5.3, 5.4 and 5.5 could not be verified using the type systems from [24, 25, 23]. Although these examples are small, it is clear that constructs like hashing, keyed hashing and sign-then-encrypt are common ingredients of realistic protocols.

In this section, we analyze two additional protocols: the (three-message version of the) Needham–Schroeder–Lowe protocol (NSL) and the BAN–Yahalom protocol. NSL could already be type-checked by Gordon and Jeffrey's system [25]. However, our new type system allows us to verify optimizations of NSL that avoid encryption of some nonces, if the secrecy of these nonces is not a security goal. These optimizations could not be verified by [25]. NSL is a good example to show how, in our system, public key encryption interacts with nonce types. For the BAN–Yahalom protocol our new type system allows us to verify additional correspondence assertions that Gordon and Jeffrey's [24, 25, 23] could not verify.

It is tedious to construct type derivations by hand and we have therefore implemented the automatic type-checker Cryptyc, which is available from [22]. We will present complete typed specifications for the NSL and BAN–Yahalom protocols using a language that is almost identical to the Cryptyc input language. (We have slightly modified the Cryptyc input by hand, so that it is consistent with the syntax used in this paper.)

6.1 The Needham-Schroeder-Lowe Protocol

We first analyze a variant of NSL, which includes in the second message a secret m generated by principal B. This variant of NSL is convenient for explaining several aspects of the type system. The protocol uses A and B's public encryption keys epA and epB.

$$
\begin{array}{ll}
A \to B & \{\!|\mathit{msg1}(na, A)|\!\}_{epB} \\
\multicolumn{2}{l}{B \text{ begins } (B \text{ authenticating } m \text{ to } A)} \\
B \to A & \{\!|\mathit{msg2}(na, nb, m, B)|\!\}_{epA} \\
\multicolumn{2}{l}{A \text{ ends } (B \text{ authenticating } m \text{ to } A)} \\
\multicolumn{2}{l}{A \text{ begins } (A \text{ authenticating to } B)} \\
A \to B & \{\!|\mathit{msg3}(nb)|\!\}_{epB} \\
\multicolumn{2}{l}{B \text{ ends } (A \text{ authenticating to } B)}
\end{array}
$$

Figure 3 shows our typed specification of NSL, using syntax that closely resembles the input language for our automatic type-checker Cryptyc [22]. It contains a series of type definitions (introduced by the keyword `type`), pattern definitions (introduced by the keyword `pattern`) and tag declarations (introduced by the keyword `tag`), followed by the process definitions for the initiator and the responder (introduced by the keyword `process`).

The nonce `na` is a SOSH nonce and accounts both for the authentication of `(b,m)` to `a` and the secrecy of `m`. Note, in particular, that the type assertion `m:Secret` is contained in the response type for `na`. This links the secrecy of `m` to the nonce `na`. The fact that `m`'s secrecy is associated with the nonce is no surprise, because without this nonce principal B would have no way of knowing that `m` comes from an honest agent. The nonce `nb` is a SOPH nonce that accounts for `a`'s authentication to `b`.

Figure 4 shows an interesting bit of the type derivation for NSL, namely, the proof of the initiator's input post-condition. We invite you to read the type derivation upwards in a goal-directed fashion. Here are a few remarks: In order to prove the input post-condition $\overline{A}_{post}$, we need to construct two subderivations $\mathcal{D}_{untrusted}$ and $\mathcal{D}_{trusted}$. Intuitively, $\mathcal{D}_{untrusted}$ covers the case where the ciphertext comes from the opponent, whereas $\mathcal{D}_{trusted}$ covers the case where the ciphertext comes from an honest agent. $\mathcal{D}_{trusted}$ and $\mathcal{D}_{untrusted}$ share a common subderivation $\mathcal{D}_{use-na}$, which applies the (Nonce Use) rule to na. The (Nonce Use) rule unwraps the assertions $\overline{A}_{na}$ wrapped inside na's response type. Note that this is only allowed if the environment contains a freshness assertion for na. The derivation $\mathcal{D}_{untrusted}$ contains two interesting uses of subtyping: Firstly, the type Un of nb is upcast to the challenge type $\mathsf{SophChall}(\overline{A}_{nb})$. Intuitively, such a type cast is safe because assertions wrapped

```
/* Nonce types for nA. */
type ChallengeAToB = SoshChall ();
type ResponseBToA (a:Public, b:Public, m:Top)
  = SoshResp ( m:Secret, begun(b authenticating m to a) );

/* Nonce types for nB. */
type ChallengeBToA (a:Public, b:Public)
  = SophChall ( begun(a authenticating to b) );
type ResponseAToB = SophResp ();

/* Message patterns. */
pattern Msg1 = (ChallengeAToB, Public);
pattern Msg2 (a:Public)
  = ( na : ResponseBToA(a,b,m),
      nb : ChallengeBToA(a,b),
      m : Top,
      b : Public );
pattern Msg3 = ResponseAToB;

/* Message tags. */
tag msg1 : Msg1 -> Auth(b:Public);
tag msg2 : Msg2(a) -> Auth(a:Public);
tag msg3 : Msg3 -> Auth(b:Public);

process Initiator (a:Public, dpA:PublicCryptoDK(a), b:Public, epB:PublicCryptoEK(b)) {
  new (na:ChallengeAToB);
  out net {| msg1(na,a) |}epB;
  inp net {| msg2(na, nb:ChallengeBToA(a,b), m:Secret, b) |}dpA^-1
    [ begun(b authenticating m to a) ];
  end(b authenticating m to a);
  begin(a authenticating to b);
  out net {| msg3(nb) |}epB;
}

process Responder (b:Public, dpB:PublicCryptoDK(b), a:Public, epA:PublicCryptoEK(a)) {
  inp net is {| msg1(na:ChallengeAToB, a) |}dpB^-1;
  new (m:Secret);
  new (nb:ChallengeBToA(a,b));
  begin(b authenticating m to a);
  out net {| msg2(na,nb,m,b) |}epA;
  inp net {| msg3(nb) |}dpB^-1
    [ begun(a authenticating to b) ];
  end(a authenticating to b);
}
```

Figure 3: The Needham–Schroeder–Lowe protocol

$E_0 \triangleq$ some environment such that $a, b \in dom(E_0)$ and $E_0 \vdash dpa : \mathsf{PublicCryptoDK}(a)$
$E \triangleq (E_0, \mathsf{fresh}(na : \mathsf{SoshChall}()))$
$\overline{A}_{na} \triangleq (\ m : \mathsf{Secret}, \mathsf{begun}(auth(b,m,a))\ )$ $\qquad \overline{A}_{nb} \triangleq \mathsf{begun}(auth(a,b))$ $\qquad \overline{A}_{post} \triangleq (\ nb : \mathsf{SophChall}(\overline{A}_{nb}), \overline{A}_{na}\ )$

Checking the initiator's input post-condition:

$$\dfrac{\dfrac{}{nb,m,E \vdash dpa : \mathsf{PublicCryptoDK}(a)} \qquad \mathcal{D}_{untrusted} \qquad \mathcal{D}_{trusted}}{nb,m,E,\ \{\!|msg2(na,nb,m,b)|\!\}_{dsA^{-1}} : \mathsf{Un} \vdash \overline{A}_{post}}\ \text{(Prv Decrypt Tainted)}$$

Subderivation $\mathcal{D}_{trusted}$:

$$\dfrac{\dfrac{\mathcal{D}_{use-na}}{nb,m,E,\ na : \mathsf{SoshResp}(\overline{A}_{na}),\ nb : \mathsf{SophChall}(\overline{A}_{nb}), m : \mathsf{Top}, b : \mathsf{Public} \vdash \overline{A}_{post}}\ \text{(Id),(Weaken)}}{nb,m,E,\ msg2(na,nb,m,b) : \mathsf{Secret}(a) \vdash \overline{A}_{post}}\ \text{(Untag Untainted)}$$

Subderivation $\mathcal{D}_{untrusted}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}_{use-na}}{nb,m,E,\ na : \mathsf{Un} \vdash \overline{A}_{na}}\ \text{(Unsub)}}{nb,m,E,\ na : \mathsf{Un}, nb : \mathsf{SophChall}(\overline{A}_{nb}) \vdash \overline{A}_{post}}\ \text{(Id),(Weaken)}}{nb,m,E,\ na : \mathsf{Un}, nb : \mathsf{Un}, m : \mathsf{Un}, b : \mathsf{Un} \vdash \overline{A}_{post}}\ \text{(Unsub),(Weaken)}}{nb,m,E,\ msg2(na,nb,m,b) : \mathsf{Un} \vdash \overline{A}_{post}}\ \text{(Untag Tainted),(Split)}$$

Subderivation $\mathcal{D}_{use-na}$:

$$\dfrac{\dfrac{}{nb,m,E_0,\ na : \mathsf{Stale}, \overline{A}_{na} \vdash \overline{A}_{na}}\ \text{(Lift),(Id)}}{nb,m,E_0, \mathsf{fresh}(na : \mathsf{SoshChall}()), na : \mathsf{SoshResp}(\overline{A}_{na}) \vdash \overline{A}_{na}}\ \text{(Nonce Use)}$$

Figure 4: Checking the initiator's input post-condition for NSL

inside a challenge type represent *obligations*. The upcast introduces additional obligations, and adding obligations is harmless. Assertions wrapped inside response types, on the other hand, represent *benefits*. The second use of (Unsub) in $\mathcal{D}_{untrusted}$ upcasts Un to $\mathsf{SoshResp}(\overline{A}_{na})$. On first sight, this seems unsafe because adding benefits is certainly not safe in general. However, names that are generated as SOSH challenges never become public. Consequently, a type environment that contains both an assertion $\mathsf{fresh}(na : \mathsf{SoshChall}(\overline{B}))$ and a type assertion $na : \mathsf{Un}$ does not correspond to a feasible runtime configuration. Therefore, if the type environment $(nb, m, E,\ na : \mathsf{Un})$ represents a feasible runtime configuration, then the (Nonce Use) rule is not applicable to $(nb, m, E,\ na : \mathsf{SoshResp}(\overline{A}_{na}))$. In this sense, $\overline{A}_{na}$ is a *useless benefit*. The upcast from Un to $\mathsf{SoshResp}(\overline{A}_{na})$ introduces a useless benefit, and adding useless benefits is harmless.

Note that *nb* has a SOPH nonce type. For SOPH nonces the response may be public. We can therefore modify the protocol and avoid the encryption of the third message. The resulting protocol still type-checks and is thus robustly safe. This optimized protocol does not type-check in Gordon and Jeffrey's earlier system [25], because that system requires SOPH challenges to be encrypted by symmetric keys. If we omit the secret message *m* from the protocol, we can also turn *na* into a SOPH nonce and omit the encryption of *na*'s response. So the following two variations of NSL now type-check:

$A \to B \qquad \{\!|msg1(na, A)|\!\}_{epB}$
B begins (B authenticating m to A)
$B \to A \qquad \{\!|msg2(na, nb, m, B)|\!\}_{epA}$
A ends (B authenticating m to A)
A begins (A authenticating to B)
$A \to B \qquad nb$
B ends (A authenticating to B)

$A \to B \qquad \{\!|msg1(na, A)|\!\}_{epB}$
B begins (B authenticating to A)
$B \to A \qquad na, \{\!|msg2(nb, B)|\!\}_{epA}$
A ends (B authenticating to A)
A begins (A authenticating to B)
$A \to B \qquad nb$
B ends (A authenticating to B)

Note that, for a simple scoping reason, we cannot omit principal name B from *msg2*. An attempt to type-check this modified protocol would force us to also modify the definition of the message pattern for the second message:

```
pattern Msg2 (a:Public)
  = ( na : ResponseBToA(a,b,m),
      nb : ChallengeBToA(a,b),    // scoping error: b is not bound
      m : Top );
```

This means that the original Needham–Schroeder Public Key protocol (NSPK) does not type-check. Moreover, our type system guides us towards the fix that protects against Lowe's attack [31]. The obvious way to fix the scoping error is the inclusion of *B*'s identity in *msg2*, which is exactly Lowe's fix. Like Gordon and Jeffrey's previous systems, our pattern-matching variant of dependent types enforces Abadi and Needham's engineering principle 3 [8]:

> If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message.

It is interesting that Lowe's attack is not captured by our attacker model, which does not include attacks by compromised principals. We currently do not know of any protocol that passes our type-checker but is vulnerable to an attack by a compromised principal. It is conceivable that our type system guarantees robust safety even in the presence of compromised principals. Unfortunately, we do not currently know whether or not this is the case.

6.2 The BAN–Yahalom Protocol

In their BAN logic article [15], Burrows, Abadi and Needham propose the following slightly modified version of the Yahalom protocol. The goal of this protocol is that *A* and *B* establish a session key *kab* using the long-term keys *kas* and *kbs* they share with a server *S*.[5]

$$\begin{array}{ll}
A \rightarrow B & A, na \\
B \rightarrow S & B, nb, \{msg2(A,na)\}_{kbs} \\
S \rightarrow A & nb, \{msg3(B,kab,na)\}_{kas}, \{msg4a(A,kab,nb)\}_{kbs} \\
A \rightarrow B & \{msg4a(A,kab,nb)\}_{kbs}, \{msg4b(nb)\}_{kab}
\end{array}$$

Figure 5 shows the type, pattern and tag definitions for BAN–Yahalom and Figure 6 the process definitions. They contain four correspondence specifications:

(a) The server begins a one-one correspondence `begin(s providing kab to a for b)`.

(b) The server begins a one-one correspondence `begin(s providing to kab b for a)`.

(c) The initiator begins a one-one correspondence `begin!(a acknowledging receipt of kab to b)`.

(d) The responder begins a one-many correspondence `begin(b acknowledging receipt of na to a)`.

Correspondences (a) and (b) say that upon successful completion of a protocol run both principals know that the server has issued `kab` to be shared with the other principal. These two correspondences do not guarantee that the other principal has actually received `kab`. The responder gets this guarantee when receiving *msg4b* encrypted by `kab`. This is formally captured by correspondence (c). The initiator only gets a weaker guarantee. She never knows that the responder has received the session key, but when receiving *msg3* she does know that

[5]Our model does not distinguish between long- and short-term keys. Key distribution protocols are usually designed to be safe, even if distributed session keys can be cracked given enough time. This can for instance be verifi ed using a type system for a timed spi-calculus (see [27]).

```
/* Challenge type. */
type Challenge = PoshChall ();

/* Response types for na. */
type ResponseBToA (a:Public, b:Public, na:Top)
  = PoshResp ( begun(b acknowledging receipt of na to a) );
type ResponseSToA (a:Public, b:Public, s:Public, kab:SymK(a,b,kab), na:Top)
  = PoshResp ( begun(s providing kab to a for b),
               begun(b acknowledging receipt of na to a) );

/* Response type for nb. */
type ResponseSToB (a:Public, b:Public, s:Public, kab:SymK(a,b,kab))
  = PoshResp ( begun(s providing kab to b for a) );

/* Message patterns. */
pattern Msg2 (b:Public)
  = ( a:Public,
      nA:ResponseBToA(a,b,nA) );
pattern Msg3 (a:Public, s:Public)
  = ( b:Public,
      kab:SymK(a,b,kab),
      nA:ResponseSToA(a,b,s,kab,nA) );
pattern Msg4a (b:Public, s:Public)
  = ( a:Public,
      kab:SymK(a,b,kab),
      nB:ResponseSToB(a,b,s,kab) );
pattern Msg4b (a:Public, b:Public, kab:SymK(a,b,kab))
  = Top [ !begun(a acknowledging receipt of kab to b) ];

/* Message tags. */
tag msg2 : Msg2(b) -> Auth(b:Public, s:Public, kbs:SymK(b,s,kbs));
tag msg3 : Msg3(a,s) -> Auth(a:Public, s:Public, kas:SymK(a,s,kas));
tag msg4a : Msg4a(b,s) -> Auth(b:Public, s:Public, kbs:SymK(b,s,kbs));
tag msg4b : Msg4b(a,b,kab) -> Auth(a:Public, b:Public, kab:SymK(a,b,kab));
```

Figure 5: Types, patterns and tags for BAN–Yahalom

```
process Initiator (a:Public, b:Public, s:Public, kas:SymK(a,s,kas)) {
  new (nA:Challenge);
  out net (a,nA);
  inp net (nB:Challenge, { msg3(b, kab:SymK(a,b,kab), nA) }kas, ticket:Un)
    [ begun(s providing kab to a for b),
      begun(b acknowledging receipt of nA to a) ];
  end(s providing kab to a for b);
  end(b acknowledging receipt of nA to a);
  begin!(a acknowledging receipt of kab to b);
  out net (ticket, { msg4b(nB) }kab);
}

process Responder (a:Public, b:Public, s:Public, kbs:SymK(b,s,kbs)) {
  inp net (a, nA:Challenge);
  new (nB:Challenge);
  begin(b acknowledging receipt of nA to a);
  out net (b, nB, { msg2(a,nA) }kbs);
  inp net ( { msg4a(a, kab:SymK(a,b,kab), nB) }kbs, { msg4b(nB) }kab )
    [ begun(s providing kab to b for a),
      !begun(a acknowledging receipt of kab to b) ];
  end(s providing kab to b for a);
  end(a acknowledging receipt of kab to b);
}

process Server (s:Public, a:Public, kas:SymK(a,s,kas), b:Public, kbs:SymK(b,s,kbs)) {
  inp net (b, nB:Challenge, { msg2(a, nA:ResponseBToA(a,b,nA)) }kbs);
  new (kab:SymK(a,b,kab));
  begin(s providing kab to a for b);
  begin(s providing kab to b for a);
  out net ( { msg3(b,kab,nA) }kas, { msg4a(a,kab,nB) }kbs );
}
```

Figure 6: Initiator, responder and server for BAN–Yahalom

the responder is "in the protocol loop". This is formally captured by correspondence (d). Correspondences (a) and (b) could already be verified by Gordon and Jeffrey's systems [24, 23], but correspondences (c) and (d) could not be verified by these systems.

For verifying correspondence (c), it is important that the type of session key `kab:SymKey(a,b,kab)` refers to `kab` itself. This is needed because encryption with `kab` proves a correspondence that refers to `kab`. In Gordon and Jeffrey's earlier systems, the type of a name n could not refer to n itself. Note that correspondence (c) even type-checks if the tag *msg4b* tags the empty message instead of *nb*, i.e., $\{msg4b()\}_{kab}$ instead of $\{msg4b(nb)\}_{kab}$. On the other hand, it is not so hard to see that the presence of nonce *nb* in *msg4b* allows to strengthen the one-to-many correspondence (c) to a one-to-one correspondence. This, however, cannot currently be verified by our type system—an evidence of its incompleteness.[6]

The nonce *na* serves to prove two correspondences, namely, correspondences (a) and (d). Correspondence (d) is begun by the responder and correspondence (a) is begun by the server. Correspondingly, *na* changes its type twice: the responder casts *na*'s type from `Challenge` to `ResponseBToA(a,b,na)` (using the

[6]The problem is that our system does not allow the same nonce to be used twice in the same protocol run, even if the two nonce uses account for distinct one-one correspondences.

typing rules (Nonce Cast) and (Strengthen Resp)), and the server casts *na*'s type from `ResponseBToA(a,b,na)` to `ResponseSToA(a,b,s,kab,na)` (using the typing rule (Strengthen Resp)). As we already discussed in Section 4.7, Gordon and Jeffrey's earlier type systems do not support incremental strengthening of response types by multiple responders. This is why a type-checker based on these systems could either verify correspondence (a) or correspondence (d), but not both.

7 The Robust Safety Proof

In this section, we give an overview of the most interesting formal properties of the system and explain how to put these properties together to show robust safety. In order to make the forest behind the trees visible, we omit many proof details and instead deliver those in Appendix B. Here is a coarse map of the overall proof architecture towards robust safety:

$$\begin{array}{rcl} \text{safety + opponent typability} & \Rightarrow & \text{robust safety} \\ \text{type preservation + cut} & \Rightarrow & \text{safety} \\ \text{substitutivity + cut} & \Rightarrow & \text{type preservation} \\ \text{inversion lemmas + key uniqueness + nonce safety} & \Rightarrow & \text{cut} \end{array}$$

7.1 Opponent Typability

Safety theorems for type systems are often corollaries of type preservation theorems (a.k.a. subject reduction theorems). A type preservation theorem says that the reduction rules of the operational semantics preserve well-typedness. In type systems for cryptographic protocols, we are interested in the safety of processes $P \mid O$, where P models the cryptographic protocol and O is an opponent. In order to be able to obtain robust safety as a corollary of type preservation, it is therefore important that not only P but also the opponent O is well-typed. For this reason, type system for cryptographic protocols are typically designed so that opponent processes that have access to public names are well-typed. We call this property *opponent typability*.

Theorem 7.1 (Opponent Typability) *If O is an opponent process, $fn(O) \subseteq \vec{n}$ and $fv(O) \subseteq \vec{x}$, then $(\vec{x}; \vec{n} : \mathsf{Un}, \vec{x} : \mathsf{Un} \vdash O)$.*

For opponent typability to hold, it is important that opponent processes are Dolev–Yao implementable. For instance, the following non-implementable process P, which decrypts a message without knowing the decryption key, is not well-typed, i.e., $net : \mathsf{Un} \nvdash P$.

$$P \;\triangleq\; \mathsf{inp}\; net\; \exists x, y.\, \{\!|x|\!\}_{y^{-1}}[\{\!|x|\!\}_{y^{-1}} : \mathsf{Un}]; \mathsf{out}\; net\; x$$

The proof of Theorem 7.1 makes use of the following lemma:

Lemma 7.1 (Dolev–Yao $\Rightarrow$ Typability) *If $(\vec{M} \Vdash \vec{N})$, then $(\vec{M} : \mathsf{Un} \vdash \vec{N} : \mathsf{Un})$.*

The proofs of Theorem 7.1 and Lemma 7.1 are pretty direct and can be found in Appendix B.3.

7.2 Safety + Opponent Typability $\Rightarrow$ Robust Safety

Given that opponent processes are typable, robust safety of the type system follows easily from the fact that well-typed processes are safe in isolation and the fact that the type system works compositionally. Lemma 7.2 below states that well-typed processes are safe in isolation. We postpone its proof to Section 7.6. We show in this section how to prove robust safety of the type system from its safety and opponent typability. To follow the robust safety proof, you merely need to recall that P is defined to be robustly safe iff $P \mid O$ is safe for all closed opponent processes O.

Definition 7.1 (Closed Types) A type T is *closed* iff $fv(T) = \emptyset$ (but not necessarily $fn(T) = \emptyset$).

Lemma 7.2 (Safety for Authenticity) *If $\vec{n}$ are distinct names, $\vec{T}$ are closed types and $(\vec{n} : \vec{T} \vdash P)$, then $(\vec{n} : \vec{T} \ ::: \ P)$ is safe for authenticity.*

Proof of Theorem 3.1 (Robust Safety for Authenticity) *If $\vec{n}$ are distinct names and $(\vec{n} : \mathsf{Un} \vdash P)$, then P is robustly safe for authenticity.*

Proof Suppose $\vec{n}$ distinct and $(\vec{n} : \vec{T} \vdash P)$. Let O be a closed opponent process. Let $\vec{m}$ be a vector of names that occur free in $(P \mid O)$ but are not already contained in $\vec{n}$. Then $(\vec{m} : \mathsf{Un}, \vec{n} : \mathsf{Un} \vdash O)$, by opponent typability (Theorem 7.1) and weakening. Then $(\vec{m} : \mathsf{Un}, \vec{n} : \mathsf{Un} \vdash P \mid O)$, by (Proc Par). Then $(\vec{m} : \mathsf{Un}, \vec{n} : \mathsf{Un} \ ::: \ P \mid O)$ is safe for authenticity, by Lemma 7.2. □

Robust write- and read-safety follow from safety lemmas for write- and read-safety in essentially the same way (see Appendices B.8 and B.9).

7.3 Substitutivity

Substitutivity is a crucial property of many type systems and is needed to show that substituting reduction rules, like our (Redn IO), preserve well-typedness. Usually, substitutivity roughly says that applying *well-typed* substitutions to typing judgments preserves derivability. Our type system is a bit peculiar in that it allows composite terms on the left hand side of typing judgments. For this reason, we do not even need substitutions to be well-typed. In our system, applying *arbitrary* substitutions to typing judgments preserves derivability. Our substitutivity lemma even says a bit more, namely that applying a substitution does not increase the height of a type derivation. This fact is needed in the (Cut) proof, in order to deal with the placeholder x for the unknown authenticator in typing rule (Decrypt Untrusted).

To conveniently state the substitutivity lemma, we introduce some notation: Let $\mathcal{J}$ range over *judgments*. Let $\mathcal{D}$ range over type derivations. If $\mathcal{D}$ is a derivation, we write $\mathcal{D} \rhd \mathcal{J}$ to indicate that $\mathcal{D}$ is a derivation for judgment $\mathcal{J}$. If $\mathcal{D}$ is a derivation, we write $\mathcal{D} = ((\mathcal{D}_1, \ldots, \mathcal{D}_n), \mathcal{J}, (Rule))$ to indicate that $\mathcal{D} \rhd \mathcal{J}$, $\mathcal{D}$'s immediate subderivations are $\mathcal{D}_1, \ldots, \mathcal{D}_n$, and $\mathcal{D}$'s last rule is $(Rule)$. For a derivation $\mathcal{D} = ((\mathcal{D}_1, \ldots, \mathcal{D}_n), \mathcal{J}, (Rule))$ we inductively define $\mathcal{D}$'s *height* as follows:

$$\mathsf{height}(\mathcal{D}) \ \stackrel{\Delta}{=} \ 1 + \bigvee_{i=1}^{n} \mathsf{height}(\mathcal{D}_i);$$

where $\bigvee$ is the least upper bound function on non-negative integers.

Lemma 7.3 (Substitutivity) *If $E \vdash \diamond$, $dom(E) = dom(\sigma)$ and $\mathcal{D} \rhd (E \vdash \mathcal{R})$, then there exists $\mathcal{D}'$ such that $\mathcal{D}' \rhd (E\{\sigma\} \vdash \mathcal{R}\{\sigma\})$ and $\mathsf{height}(\mathcal{D}') \leq \mathsf{height}(\mathcal{D})$.*

Proof This lemma holds because applying a substitution to an instance of a typing rule results in an instance of the same typing rule. Note, in particular, that this is also true for the axiom (Id), because we permit environments to contain non-atomic type assertions of the form $(M : T)$ (unlike in many other type systems where environments are functions from variables to types). Formally, the proof is by induction on the structure of $\mathcal{D}$. The only cases that require some care are where $\mathcal{D}$ ends in a rule that involves unification, i.e., (Decrypt Trusted) or (Decrypt Untrusted). In these cases, one factorizes σ by the most general unifier and then applies the resulting mediating substitution to the subderivation. This application of the mediating substitution preserves derivability, by induction hypothesis. We omit the details. □

7.4 Cut

Just as importantly as substitutivity, our type system is designed to satisfy (Cut):

Theorem 7.2 (Cut) *If E is nominal, $(E \vdash !\overline{B}_0, \overline{B}_1, \overline{B}_2)$ and $(!\overline{B}_0, \overline{B}_1 \vdash \overline{C})$, then $(E \vdash !\overline{B}_0, \overline{B}_2, \overline{C})$.*

We will discuss the proof of (Cut) in Section 7.7 and present the proof details in Appendix B.7. Note that (Cut) requires nominality of environment E. Nominal environments satisfy a number of sanity conditions. Most prominently, every type assertion in a nominal environment has to be of the form $n : T$, i.e., type assertions of the forms $M : T$ or $x : T$ are not allowed in nominal environments.

Definition 7.2 (Nominal Environments) An environment E is called *nominal* iff all of the following statements hold:

- *Variable freeness: $dom(E) = \emptyset$* and $E \vdash \diamond$.
- *Nominal type assertions:* If $E = (E', M : T)$ or $E = (E', \mathsf{fresh}(M : T))$, then M is a name.
- *Weak functionality:* If $E = (E', n : T, n : U)$, then $\mathsf{Stale} \leq T$.
- *Fresh-linearity:* Neither $E = (E', \mathsf{fresh}(n : T), \mathsf{fresh}(n : U))$ nor $E = (E', !\mathsf{fresh}(n : T))$.
- *Type-consistency for fresh-assertions:* If $E = (E', \mathsf{fresh}(n : T), n : U)$, then $T \leq U$.

The following lemma collects closure properties of nominal environments. These closure properties imply that all typing rules (when read upwards) preserve nominality of the environment. This is needed for inductively proving statements of the form "If E is nominal and $(E \vdash \mathcal{R})$, then ...".

Lemma 7.4 (Closure Properties of Nominal Environments)

(a) *If (E, A) is nominal, then so is E.*

(b) *If $(E, \overline{A})$ is nominal and $\overline{A} \equiv \overline{B}$, then $(E, \overline{B})$ is nominal.*

(c) *If $(E, n : T)$ is nominal, $(T \leq U)$ and $fv(U) = \emptyset$, then $(E, n : U)$ is nominal.*

(d) *If $(E, \mathsf{fresh}(n : (K,H)\,\mathsf{Chall}(\overline{A})), n : (K,H)\,\mathsf{Resp}(\overline{B}))$ is nominal, then $\overline{A} = \overline{B} = \emptyset$.*

(e) *If $(E, \mathsf{fresh}(n : (K,H)\,\mathsf{Chall}(\overline{A})))$ is nominal, then so is $(E, n : \mathsf{Stale})$.*

Proof Parts (a), (b) (c), and (e) are easily checked. To prove part (d), let $T = (K,H)\,\mathsf{Chall}(\overline{A})$ and $U = (K,H)\,\mathsf{Resp}(\overline{B})$ and assume that $(E, \mathsf{fresh}(n : T), n : U)$ is nominal. Then $(T \leq U)$, by type consistency for fresh-assertions. This is only possible if T is public and U is tainted. Then $\overline{A} = \overline{B} = \emptyset$, by the syntactic restrictions on challenge and response types. $\square$

7.5 Type Preservation

Well-typed computation states are defined by the following rule:

Well-typed Computation States, $\vdash \overline{A} ::: P$:

(State)

$$\frac{\overline{A}\ \text{nominal} \quad \overline{A} \vdash \overline{B} \quad \overline{B} \vdash P}{\vdash \overline{A} ::: P}$$

Using (Cut) and substitutivity, it is now not hard to show that our state transition rules preserve well-typedness.

Lemma 7.5 (Structural Equivalence Preserves Typing)

(a) *If* $P \equiv Q$*, then* $(E \vdash P)$ *iff* $(E \vdash Q)$.

(b) *If* $\overline{A} \equiv \overline{B}$*, then* $(E, \overline{A} \vdash \mathcal{R})$ *iff* $(E, \overline{B} \vdash \mathcal{R})$.

(c) *If* $(\overline{A} ::: P) \equiv (\overline{B} ::: Q)$*, then* $(\vdash \overline{A} ::: P)$ *iff* $(\vdash \overline{B} ::: Q)$.

Proof Part (a) is proved by induction on $(P \equiv Q)$'s derivation. One uses that the typing rules for processes are invertible. To prove part (b), one first proves the following two auxiliary statements, separately by induction on $\mathsf{height}(\mathcal{D})$.

(a) *If* $\mathcal{D} \rhd (E, !\mathsf{begun}(M), \mathsf{begun}(M) \vdash \mathcal{R})$*, then* $(E, !\mathsf{begun}(M) \vdash \mathcal{R})$.

(b) *If* $\mathcal{D} \rhd (E, !\mathsf{begun}(M), !\mathsf{begun}(M) \vdash \mathcal{R})$*, then* $(E, !\mathsf{begun}(M) \vdash \mathcal{R})$.

Given these two auxiliary statements, it is straightforward to prove part (b) of the lemma by induction on $(\overline{A} \equiv \overline{B})$'s derivation. Part (c) is a simple consequence of parts (a) and (b), and the fact that structural equivalence of assertion sets preserves nominality. $\square$

Theorem 7.3 (Type Preservation) *If* $\vdash \overline{A} ::: P$ *and* $(\overline{A} ::: P) \rightarrow (\overline{B} ::: Q)$*, then* $\vdash \overline{B} ::: Q$.

Proof By induction on $(\overline{A} ::: P) \rightarrow (\overline{B} ::: Q)$'s derivation. Suppose $\vdash \overline{A} ::: P$ and $\mathcal{D} \rhd (\overline{A} ::: P) \rightarrow (\overline{B} ::: Q)$. ($\mathcal{D}$ is a derivation tree whose nodes are applications of the (Redn Equiv) rule.) By inverting the typing judgment $\vdash \overline{A} ::: P$, we obtain that $\overline{A}$ is nominal and $(\overline{A} \vdash \overline{A'})$ and $(\overline{A'} \vdash P)$ for some $\overline{A'}$.

Suppose $\mathcal{D}$ ends in (Redn Equiv). Then $(\overline{A} ::: P) \equiv (\overline{A'} ::: P') \rightarrow (\overline{B'} ::: Q') \equiv (\overline{B} ::: Q)$ for some $\overline{A'}$, P', $\overline{B'}$, Q'. By Lemma 7.5, it is the case that $\vdash \overline{A'} ::: P'$. Then, by induction hypothesis, $\vdash \overline{B'} ::: Q'$. Then $\vdash \overline{B} ::: Q$, by Lemma 7.5.

Suppose in the remainder that $\mathcal{D}$ does not end in (Redn Equiv). Then $P = (P_1 \mid P_2)$ and $Q = (P'_1 \mid P_2)$ for some P_1, P_2, P'_1, by inspection of the reduction rules.

Suppose $\mathcal{D}$ ends in (Redn New). Then $P_1 = (\mathsf{new}\, n{:}T; P'_1)$, $n \notin fn(\overline{A}, Q)$ and $\overline{B} = (\overline{A}, n : T, \mathsf{fresh}(n : T))$ for some n, T. Let $\overline{B'} = (\overline{A'}, n : T, \mathsf{fresh}(n : T))$. By inverting the last typing rules of $(\overline{A'} \vdash P)$'s derivation, we get that $\overline{A'} = (!\overline{A'_0}, \overline{A'_1}, \overline{A'_2})$, $(!\overline{A_0}, \overline{A'_1}, n : T, \mathsf{fresh}(n : T) \vdash P'_1)$ and $(!\overline{A_0}, \overline{A'_2} \vdash P_2)$ for some $!\overline{A'_0}$, $\overline{A'_1}$, $\overline{A'_2}$. By (Proc Par), we get that $(\overline{B'} \vdash Q)$. Applying (Lift) twice to $(\overline{A} \vdash \overline{A'})$, we get $(\overline{B} \vdash \overline{B'})$. From $(\overline{B} \vdash \overline{B'})$ and $(\overline{B'} \vdash Q)$ we get $\vdash \overline{B} ::: Q$, by (State).

Suppose $\mathcal{D}$ ends in (Redn Begin One). Then $P_1 = (\mathsf{begin}(M); P'_1)$ and $\overline{B} = (\overline{A}, \mathsf{begun}(M))$. Let $\overline{B'} = (\overline{A'}, \mathsf{begun}(M))$ for some M. By inverting the last typing rules of $(\overline{A'} \vdash P)$'s derivation, we get that $\overline{A'} = (!\overline{A'_0}, \overline{A'_1}, \overline{A'_2})$, $(!\overline{A_0}, \overline{A'_1}, \mathsf{begun}(M) \vdash P'_1)$ and $(!\overline{A_0}, \overline{A'_2} \vdash P_2)$ for some $!\overline{A'_0}$, $\overline{A'_1}$, $\overline{A'_2}$. Then $(\overline{B'} \vdash Q)$, by (Proc Par). Applying (Lift) to $(\overline{A} \vdash \overline{A'})$, we get that $(\overline{B} \vdash \overline{B'})$. From $(\overline{B} \vdash \overline{B'})$ and $(\overline{B'} \vdash Q)$ we get $\vdash \overline{B} ::: Q$, by (State).

The proof case for (Redn Begin Many) is essentially identical to the previous proof case.

Suppose $\mathcal{D}$ ends in (Proc End). Then $P_1 = (\mathsf{end}(M); P'_1)$ and $\overline{A} = (\overline{B}, \mathsf{begun}(M))$ for some M. By inverting $(\overline{A'} \vdash P)$'s last rules, we get that $\overline{A'} = (!\overline{A'_0}, \overline{A'_1}, \overline{A'_2}, \overline{A'_3})$, $(!\overline{A'_0}, \overline{A'_1} \vdash \mathsf{begun}(M))$, $(!\overline{A'_0}, \overline{A'_2} \vdash P'_1)$ and $(!\overline{A'_0}, \overline{A'_3} \vdash P_2)$ for some $!\overline{A'_0}$, $\overline{A'_1}$, $\overline{A'_2}$, $\overline{A'_3}$. Let $\overline{B'} = (!\overline{A'_0}, \overline{A'_2}, \overline{A'_3})$. From $(!\overline{A'_0}, \overline{A'_2} \vdash P'_1)$ and $(!\overline{A'_0}, \overline{A'_3} \vdash P_2)$, we obtain $(\overline{B'} \vdash Q)$, by (Proc Par). On the other hand, cutting $(\overline{A} \vdash \overline{A'})$ with $(!\overline{A'_0}, \overline{A_1} \vdash \mathsf{begun}(M))$ results in $(\overline{A} \vdash \overline{B'}, \mathsf{begun}(M))$. Then, by Lemma B.16, either $(\overline{B} \vdash \overline{B'})$ or $!\mathsf{begun}(M) \in \overline{B}$. In the second case $\overline{B} \equiv \overline{A}$, and we obtain $(\overline{B} \vdash \overline{B'})$ from $(\overline{A} \vdash \overline{A'})$ by Lemma 7.5 and weakening. Thus, we have established that $(\overline{B} \vdash \overline{B'})$ and $(\overline{B'} \vdash Q)$. Then $\vdash \overline{B} ::: Q$, by (State).

Suppose $\mathcal{D}$ ends in (Redn IO). Then $P_1 = (\mathsf{out}\ L\ M\{\vec{x} \leftarrow \vec{N}\} \mid \mathsf{inp}\ L\ \exists \vec{x} \,.\, M[\overline{C}]; P''_1)$ and $P'_1 = P''_1\{\vec{x} \leftarrow \vec{N}\}$ and $\overline{B} = \overline{A}$. Inverting the last rules of $(\overline{A'} \vdash P)$, we get that there exist $!\overline{A'_0}$, $\overline{A'_1}$, $\overline{A'_2}$, $\overline{A'_3}$, $\overline{A'_4}$, $\overline{A'_5}$, $\vec{x}$ such that

$\overline{A}' = (!\overline{A}'_0, \overline{A}'_1, \overline{A}'_2, \overline{A}'_3, \overline{A}'_4, \overline{A}'_5)$ and the following statements hold:

(1) $!\overline{A}'_0, \overline{A}'_1 \vdash L : \mathsf{Un}, M\{\vec{x} \leftarrow \vec{N}\} : \mathsf{Un}$
(2) $!\overline{A}'_0, \overline{A}'_2 \vdash L : \mathsf{Un}$
(3) $\vec{x}, !\overline{A}'_0, \overline{A}'_3, M : \mathsf{Un} \vdash \overline{C}$
(4) $\vec{x}, !\overline{A}'_0, \overline{A}'_4, \overline{C} \vdash P''_1$
(5) $!\overline{A}'_0, \overline{A}'_5 \vdash P_2$

Let $\overline{B}' = (!\overline{A}'_0, \overline{A}'_4, \overline{A}'_5, \overline{C}\{\vec{x} \leftarrow \vec{N}\})$. Applying the substitution $\{\vec{x} \leftarrow \vec{N}\}$ to judgment (4) results in $(!\overline{A}'_0, \overline{A}'_4, \overline{C}\{\vec{x} \leftarrow \vec{N}\} \vdash P'_1)$. From this judgment and (5), we obtain $(\overline{B}' \vdash Q)$, by (Proc Par). On the other hand, applying substitution $\{\vec{x} \leftarrow \vec{N}\}$ to judgment (3) results in $(!\overline{A}'_0, \overline{A}'_3, M\{\vec{x} \leftarrow \vec{N}\} : \mathsf{Un} \vdash \overline{C}\{\vec{x} \leftarrow \vec{N}\})$. Cutting $(\overline{A} \vdash \overline{A}')$ with this judgment and then applying weakening results in $(\overline{A} \vdash \overline{B}')$. We have, thus, established $(\overline{B} = \overline{A} \vdash \overline{B}')$ and $(\overline{B}' \vdash Q)$. Therefore, $\vdash \overline{B} ::: Q$, by (State). □

7.6 Safety

Recall Definition 3.2 of safety for authenticity. This definition roughly says that a process is safe if it can never reach a state where the next process instruction ends a session but the computation state contains no record that such a session has previously begun. Such a state is *not* a well-typed computation state. By the type preservation theorem, a well-typed process can therefore never reach such a state. This observation is essentially the safety proof for authenticity. Here is the precise proof:

Proof of Lemma 7.2 (Safety for Authenticity) *If $\vec{n}$ are distinct names, $\vec{T}$ are closed types and $(\vec{n} : \vec{T} \vdash P)$, then $(\vec{n} : \vec{T} ::: P)$ is safe for authenticity.*

Proof Suppose $\vec{n}$ are distinct, $\vec{T}$ are closed types and $(\vec{n} : \vec{T} \vdash P)$. Suppose that $(\vec{n} : \vec{T} ::: P) \to^* (\overline{A} ::: (\mathsf{end}(M); Q) \mid Q')$. Because $(\vec{n} : \vec{T} \vdash P)$, it is the case that $\vdash \vec{n} : \vec{T} ::: P$, by (State). Then $\vdash \overline{A} ::: (\mathsf{end}(M); Q) \mid Q'$, by type preservation (Theorem 7.3). By inverting the last rules of this judgment, we obtain $(\overline{A} \vdash \overline{A}')$ and $(\overline{A}' \vdash \mathsf{begun}(M))$ for some $\overline{A}'$. Cutting $(\overline{A} \vdash \overline{A}')$ with $(\overline{A}' \vdash \mathsf{begun}(M))$ results in $(\overline{A} \vdash \mathsf{begun}(M))$. Then $\mathsf{begun}(M) \in \overline{A}$ or $!\mathsf{begun}(M) \in \overline{A}$, by inverting this judgment (as justified by Lemma B.16). □

7.7 Inversion Lemmas + Key Uniqueness + Nonce Safety ⇒ Cut

In this section, we explain how the (Cut) proof works and tell you what lemmas are needed. The full proof can be found in Appendix B.7.

Recall (Cut):

If E is nominal, $(E \vdash !\overline{B}_0, \overline{B}_1, \overline{B}_2)$ and $\mathcal{D} \rhd (!\overline{B}_0, \overline{B}_1 \vdash \overline{C})$, then $(E \vdash !\overline{B}_0, \overline{B}_2, \overline{C})$.

We prove (Cut) by induction on $\mathsf{height}(\mathcal{D})$. To this end, we need to make sure that, for every possible last rule of $\mathcal{D}$, the left judgment $(E \vdash !\overline{B}_0, \overline{B}_1, \overline{B}_2)$ can be transformed to match $\mathcal{D}$'s last rule's premise. Suppose, for instance, $\mathcal{D}$ ends in (Split) as shown in this picture:

$$E \vdash !\overline{B}'_0, (M,N) : K\,\mathsf{Top}, \overline{B}_1, \overline{B}_2 \qquad \dfrac{\dfrac{\vdots}{!\overline{B}'_0, M : K\,\mathsf{Top}, N : K\,\mathsf{Top}, \overline{B}_1, \overline{B}_2 \vdash \overline{C}}}{!\overline{B}'_0, (M,N) : K\,\mathsf{Top}, \overline{B}_1, \overline{B}_2 \vdash \overline{C}}\ (\text{Split})$$

In this situation, we need to know that the we can transform the left judgment into $(E \vdash !\overline{B}'_0, M : K\,\mathsf{Top}, N : K\,\mathsf{Top}, \overline{B}_1, \overline{B}_2)$. In other words, we need to know that we can invert the (Pair) rule. This will allow us to match the premise of (Split) and apply the induction hypothesis.

Inversion lemmas. Appendix B.4 states and proves a number of inversion properties. As we have just explained, these are needed to prove (Cut). For example, here are special cases of the inversion properties for pairing and encryption. They hold under the assumption that E is nominal.

- *Suppose* $(E \vdash \overline{B},\ (M,N) : K\,\mathsf{Top})$.
  Then $(E \vdash \overline{B}, M : K\,\mathsf{Top}, N : K\,\mathsf{Top})$.
- *Suppose* $(E \vdash \overline{B},\ \{\!|M|\!\}_N : \mathsf{Secret}(L))$.
  Then there exist K,H,X *such that* $(E \vdash \overline{B}, N : (K,H)\,\mathsf{EK}(X), (M,L) \in X)$ *and* $\mathsf{Tainted} \notin K$.
- *Suppose* $(E \vdash \overline{B},\ \{\!|M|\!\}_N : \mathsf{Un})$.
  Then there exist K,H,X,L *such that* $(E \vdash \overline{B}, N : (K,H)\,\mathsf{EK}(X), (M,L) \in X)$.
  Moreover, if $\mathsf{Tainted} \in K$ *or* $\mathsf{Public} \in H$ *then* $(E \vdash \overline{B}, M : \mathsf{Un})$.

Note that some of these properties do not hold if E is allowed to contain type assertions for composite terms. For instance, if $E = (n : \mathsf{Top}, k : \mathsf{Top}, \{\!|n|\!\}_k : \mathsf{Un})$, then $E \vdash \{\!|n|\!\}_k : \mathsf{Un}$ by (Id), but $E \not\vdash k : (K,H)\,\mathsf{EK}(X)$. Thus, the nominality of environment E is a crucial assumption for these inversion lemmas to hold.

Key Uniqueness. Suppose, in the (Cut) proof, we want to handle the case where the derivation of the right judgment ends in (Decrypt Trusted). In particular, consider the following situation (where $\overline{B} \triangleq !\overline{B}'_0, \overline{B}_1, \overline{B}_2$):

$$E \vdash \overline{B}, \{\!|M|\!\}_{\mathsf{Enc}(N)} : \mathsf{Secret}(L) \qquad \dfrac{\dfrac{\vdots}{\overline{B} \vdash \mathsf{Dec}(N) : (K',H')\,\mathsf{DK}(X')} \qquad \dfrac{\vdots}{\overline{B}, (M,L) \in X' \vdash \overline{C}}}{\overline{B}, \{\!|M|\!\}_{\mathsf{Enc}(N)} : \mathsf{Secret}(L) \vdash \overline{C}}\ \begin{array}{l}\text{(Decrypt Trusted)}\\ \mathsf{Tainted} \notin H'\end{array}$$

By the inversion property from above, we can transform the left judgment into $(E \vdash \overline{B}, \mathsf{Enc}(N) : (K,H)\,\mathsf{EK}(X), (M,L) \in X)$ for some K,H,X such that $\mathsf{Tainted} \notin K$. By weakening this judgment in three ways, we obtain $(E \vdash \mathsf{Enc}(N) : (K,H)\,\mathsf{EK}(X))$, $(E \vdash \overline{B})$ and $(E \vdash \overline{B}, (M,L) \in X)$. Cutting $(E \vdash \overline{B})$ with the first premise of (Decrypt Trusted), namely $(\overline{B} \vdash \mathsf{Dec}(N) : (K',H')\,\mathsf{DK}(X'))$, results in $(E \vdash \mathsf{Dec}(N) : (K',H')\,\mathsf{DK}(X'))$. Now, we would like to cut $(E \vdash \overline{B}, (M,L) \in X)$ with the second premise of (Decrypt Trusted), namely $(\overline{B}, (M,L) \in X' \vdash \overline{C})$. In order to be able to do this, we need to know that $X = X'$.

The following lemma is what is needed. It says that in nominal environments different types of the same key pair are closely related. In particular, untainted key types are unique.

Lemma 7.6 (Key Uniqueness) *If E is nominal, $E \vdash \mathsf{Enc}(N) : (K,H)\,\mathsf{EK}(X)$ and $E \vdash \mathsf{Dec}(N) : (K',H')\,\mathsf{DK}(X')$, then $(K,H,X) = (K',H',X')$ or* $\mathsf{Tainted} \in (K \cap K'^{-1}) \cup (H^{-1} \cap H') \cup (K \cap H')$

Not surprisingly, the proof of this lemma makes crucial use of the weak functionality condition on nominal environments. Without weak functionality, the environment could assign two distinct untainted key types to the same name, which would lead to a violation of key uniqueness. The proof of the key uniqueness lemma is given in Appendix B.5.

Nonce Safety. Suppose, in the (Cut) proof, we want to handle the case where the right judgment ends in (Nonce Use). So we are in the following situation (where $Ch(\overline{A}_{ch}) \triangleq (K,H)\,\mathsf{Chall}(\overline{A}_{ch})$ and $Rp(\overline{A}_{rp}) \triangleq (K,H)\,\mathsf{Resp}(\overline{A}_{rp})$ and $\overline{B} \triangleq !\overline{B}'_0, \overline{B}'_1, \overline{B}_2$):

$$E \vdash \overline{B}, \mathsf{fresh}(N : Ch(\overline{A}_{ch})), N : Rp(\overline{A}_{rp}) \qquad \dfrac{\dfrac{\vdots}{\overline{B}, N : \mathsf{Stale}, \overline{A}_{ch}, \overline{A}_{rp} \vdash \overline{C}}}{\overline{B}, \mathsf{fresh}(N : Ch(\overline{A}_{ch})), N : Rp(\overline{A}_{rp}) \vdash \overline{C}}\ \text{(Nonce Use)}$$

We need the following lemma:

Lemma 7.7 (Nonce Safety) *If E is nominal and* $(E \vdash \overline{B}, \mathsf{fresh}(N : (K,H)\,\mathsf{Chall}(\overline{A}_{ch})), N : (K,H)\,\mathsf{Resp}(\overline{A}_{rp}))$, *then* $(E \vdash \overline{B}, N : \mathsf{Stale}, \overline{A}_{ch}, \overline{A}_{rp})$.

The nonce safety lemma is proven in Appendix B.6. For it to hold, the typing rules (Discard Chall) and (Subty Stale Nonce) are needed. Without either of these rules the following judgment would not be derivable:

$$\mathsf{fresh}(n : \mathit{Ch}()), n : \mathit{Ch}(), \mathsf{begun}() \vdash \mathsf{begun}(), n : \mathit{Rp}(\mathsf{begun}())$$

By the nonce safety lemma, this judgment is a consequence of the following judgment:

$$n : \mathit{Ch}(), \mathsf{begun}() \vdash n : \mathit{Rp}(\mathsf{begun}()), n : \mathit{Rp}(\mathsf{begun}())$$

To derive the former judgment from the latter, first apply (Lift) to introduce the fresh-assertion on both sides, then apply the nonce safety lemma, and then apply weakening to drop the stale-assertion from the right. To derive the latter judgment, use (Nonce Cast), (Strengthen Resp) and then (Copy) to duplicate the type assertion $n : \mathit{Rp}(\mathsf{begun}())$. Note that, whereas the bare assertion $\mathsf{begun}()$ is not copyable, the type assertion $n : \mathit{Rp}(\mathsf{begun}())$ is.

8 Conclusion

8.1 Related Work

Gordon and Jeffrey's type systems for authenticity verify one-to-one correspondences for symmetric [24] and asymmetric cryptography [25] and one-to-many correspondences for symmetric cryptography [23]. Our type system builds on these systems and the differences are summarized in the introduction (Section 1) and the summary (Section 8.2).

Abadi's seminal type system [1] deals with symmetric cryptography protocols and verifies secrecy. The notion of secrecy in [1] is different from ours: whereas [1] considers secrecy based on noninterference, we consider secrecy based on the Dolev–Yao intruder (see [2] for a comparison of these two notions of secrecy). In [3], Abadi and Blanchet present a secrecy type system for public key cryptography. Their system deals with the half of asymmetric cryptography where the encryption key is public and the decryption key is private, but not with the other half. Their notion of secrecy is based on the Dolev–Yao intruder and similar to ours. A difference is that Abadi and Blanchet's system does not have explicit type annotations. Therefore, it is not obvious how to transform their type system into a type-checking algorithm. It is not straightforward to enhance their system with explicitly type annotations either, because their typing rule for destructing ciphertexts under public encryption keys checks the process continuation twice in different environments. In contrast, our corresponding typing rule only checks the input post-condition twice (instead of the entire process continuation). Here pattern-matching input helps. Because our type system processes an entire input pattern in one batch, it can essentially look ahead a few destructors, which would be messy in a process calculus with destructors. In [4], Abadi and Blanchet present a generic process calculus with constructors and destructors and a secrecy type system (without explicit type annotations) of which their system [3] is an instance of. Our pattern-matching spi-calculus is not an instance of their generic process calculus, because it uses pattern-matching input instead of destructors. Moreover, our type system is explicitly typed, whereas Abadi and Blanchet's generic system is not and cannot be easily turned into an explicitly typed system for the same reason as [4]. They describe how to translate processes into verification condition sets and prove that their verification condition generator is relatively complete with respect to the type system: if secrecy is provable by the type system, then it is also provable from the generated verification conditions. The verification conditions are Horn clauses and can be solved by resolution provers. Abadi and Blanchet report that, although resolution provers do not always terminate, in practice Blanchet's tool [11] terminates on numerous example protocols.

Bugliesi, Focardi and Maffei present an explicitly typed system for authenticity [14]. One of their goals is to replace dependent types from Gordon and Jeffrey's type systems by additional dynamically checked tags in order to simplify type annotations. In their system, not only the plaintext inside an encryption must be tagged, but each component of this plaintext must be tagged separately. These additional tags correspond to dynamic checks that are unnecessary for protocol security, as one tag inside each ciphertext suffices to avoid type flaw attacks. We deliberately avoided such spurious dynamic checks. It was not Bugliesi, Focardi and Maffei's goal to verify additional protocols beyond what Gordon and Jeffrey's type system could already verify. For instance, their system cannot verify sign-then-encrypt protocols, as our new system can. In their system, nonces only account for public messages. As a result, their system cannot verify message authentication for key establishment protocols, which is possible with both Gordon and Jeffrey's and our system. Overall, Bugliesi/Focardi/Maffei's system perhaps requires simpler type annotations than Gordon/Jeffrey's and ours, but looses some degree of completeness that way. A nice feature is that they show robust safety against internal attackers with an associated identity. In that respect, their robust safety result is stronger than Abadi/Blanchet's, Gordon/Jeffrey's and ours, but see [26] for a recent secrecy type system that allows to model compromised hosts.

Bodei, Buchholz, Degano, Nielson and Nielson present a control flow analysis for verifying one-to-many entity authentication and secrecy [12]. They use their process calculus Lysa, which is a variant of the spi-calculus. Lysa has pattern-matching input and pattern-matching decryption and is tailored to make their static analysis convenient. Their analysis tool is fully automatic; it only requires the specification of security properties but no additional protocol annotations beyond that. In contrast, our language requires type annotations to specify a type discipline and make automatic checking simple. In particular, in our system the protocol specifier is required to explicitly provide the exact format of plaintexts and the semantic guarantees that are obtained by successfully decrypting messages. Both approaches have advantages: control flow analysis is fully automatic, whereas type annotations provide tighter guidance for protocol developers and documentation why protocols are correct.

The applied pi calculus by Abadi and Fournet [6] is a generic process calculus similar to the one by Abadi and Blanchet [4], but allows arbitrary equational theories on messages and does not distinguish between constructors and destructors. We are not aware of secrecy or authenticity type systems for the applied pi calculus. Marchignoli and Martinelli's Crypto-CCS [33, 35] is another generic process calculus for cryptographic protocols. Like [6] and [4], it is parametric in the cryptographic primitives. Crypto-CCS has an inference operator that has a similar effect as destructors in the spi-calculus and as pattern-matching in our calculus. Martinelli presents a partial model-checking algorithm that can verify security properties, including secrecy, for a bounded number of sessions and bounded message size. In comparison to [6], [4] and [35], our pattern-matching spi-calculus seems less generic on first sight, because asymmetric cryptography is its only interesting primitive and is hardwired into the calculus. However, we achieve some degree of genericity by encoding other cryptographic primitives (like symmetric cryptography, hashing and keyed hashing) in terms of asymmetric cryptography. These encodings seem legitimate in a model that assumes perfect cryptography. We cannot, however, encode primitives with associated algebraic equalities, like Diffie-Hellman key agreement, which is possible in the applied pi calculus and to some extent perhaps in Abadi and Blanchet's calculus.

8.2 Summary and Future Work

In this paper, we have shown how pattern-matching types can be used to express complex data dependencies, in particular how they can be used to provide authenticity typings for nested uses of cryptography and hashing. We have shown how a combination of tag types and authentication types can be used to obtain protocol-independent key types and reusable long-term keys. We have refined the typing rules for encryption and decryption from Gordon and Jeffrey's earlier systems, so that sign-then-encrypt protocols and nested digital signatures are now typable. We have also refined the nonce rules to now also permit SOPH challenges that are encrypted with public encryption keys (instead of symmetric keys) and nonce challenges with multiple responders. Our technical approach was to define a small core language with only asymmetric cryptography, while obtaining other cryptographic constructions, like symmetric cryptography, message tagging, hashing and keyed hashing by translation to the core language. Interestingly, the translations that are operationally sensible also yield sensible derived typing rules. We have used the type system from this paper as the basis for the new version of the Cryptyc type-checker [22], which extends the previous Cryptyc implementation for symmetric cryptography to also handle asymmetric cryptography, nested cryptography and hashing. Our pattern-matching spi-calculus is explicitly typed, and so the implementation of an automatic type checker is not so difficult, although a number of decisions had to be made, for instance about the order in which to apply typing rules.

In the future, we want to try to develop a type system for cryptographic primitives that have associated algebraic equations. In particular, we would like to verify protocols that make use of Diffie-Hellman key establishment. To this end, we will have to modify our type system to allow composite keys. (With our current type system, all interesting keys in well-typed protocols are atomic names). An interesting question to answer is whether or not the system presented in this paper guarantees robust safety against internal compromised principals. This would strengthen our robust safety theorem, which guarantees robust safety against external opponents with no associated identity. See the end of Section 6.1 on the Needham–Schroeder–Lowe protocol for a short discussion on this subject.

Appendices

A Unification

A.1 Typed Substitution

We view substitutions as “typed” entities with explicit domain and range, as proposed, for instance, in [21]. A *substitution* is a triple of the form $\sigma = (\sigma_{map} : \sigma_{dom} \rightarrow \sigma_{ran})$ where σ_{dom} and σ_{ran} are finite sets of variables, σ_{map} is a function from σ_{dom} to the set of messages and $fv(\sigma_{map}(x)) \subseteq \sigma_{ran}$ for all x in σ_{dom}. The meta-variables σ, ρ range over substitutions. We say that σ is a substitution *from* σ_{dom} *to* σ_{ran}, and write $dom(\sigma)$ for σ_{dom} and $ran(\sigma)$ for σ_{ran}. The *application* $M\{\sigma\}$ of substitution σ to message M is defined iff $fv(M) \subseteq dom(\sigma)$. In this case, $M\{\sigma\}$ is defined inductively in the obvious way, with the case for $M = \{\!|M'|\!\}_{N^{-1}}$ as in Section 3.1. If $\vec{x} = (x_1,\dots,x_n)$ are distinct and $\vec{M} = (M_1,\dots,M_n)$, the *updated substitution* $\sigma[\vec{x}\leftarrow\vec{M}]$ is defined as follows: $(\sigma[\vec{x}\leftarrow\vec{M}])_{dom} \triangleq \sigma_{dom} \cup \vec{x}$, $(\sigma[\vec{x}\leftarrow\vec{M}])_{ran} \triangleq \sigma_{ran} \cup fv(\vec{M})$, $(\sigma[\vec{x}\leftarrow\vec{M}])_{map}(y) \triangleq M_i$, if $y = x_i$ and $1 \le i \le n$, and $(\sigma[\vec{x}\leftarrow\vec{M}])_{map}(y) \triangleq \sigma(y)$, if $y \in (\sigma_{dom} - \vec{x})$. Whenever we write $M\{\sigma\}$ with $fv(M) \not\subseteq dom(\sigma)$, this is to be interpreted as $M\{\sigma[\vec{x}\leftarrow\vec{x}]\}$ where $\vec{x} = fv(M) - dom(\sigma)$. We inductively extend the definition of substitution application to types, patterns, assertion sets and processes. The cases for the binding constructs involve variable renaming and domain extension: For instance, $\exists\vec{x}\,.\,M[\bar{A}]\{\sigma\} \triangleq \exists\vec{y}\,.\,M\{\sigma'\}[\bar{A}\{\sigma'\}]$, where $\sigma' = \sigma[\vec{x}\leftarrow\vec{y}]$ and $\vec{y}\cap\sigma_{ran} = \emptyset$. The application of substitution σ to environment $(\bar{x};\bar{A})$ is defined iff $dom(\sigma) = \bar{x}$: $(\bar{x};\bar{A})\{\sigma\} \triangleq (\sigma_{ran};\bar{A}\{\sigma\})$. The *empty substitution* $\emptyset$ has empty domain, empty range and empty map. For $n \geq 0$, $\vec{x} = (x_1,\dots,x_n)$ distinct and $\vec{M} = (M_1,\dots,M_n)$, we define the *enumerated substitution* $\vec{x}\leftarrow\vec{M} \triangleq \emptyset[\vec{x}\leftarrow\vec{M}]$. For $ran(\sigma) = dom(\rho)$, we define the *substitution composition* $(\sigma;\rho)$: $(\sigma;\rho)_{dom} \triangleq \sigma_{dom}$, $(\sigma;\rho)_{ran} \triangleq \rho_{ran}$ and $(\sigma;\rho)_{map}(x) \triangleq (\sigma_{map}(x))\{\rho\}$. If $M\{\sigma;\rho\}$ is defined, then so is $M\{\sigma\}\{\rho\}$ and $M\{\sigma\}\{\rho\} = M\{\sigma;\rho\}$. The same equation holds (up to α-equivalence), if message M is replaced by a type, pattern, assertion set, process or environment.

A.2 Unification

If $fv(M,N) \subseteq \bar{x}$, then a *unifier* of M and N from $\bar{x}$ is a substitution σ from $\bar{x}$ such that $M\{\sigma\} = N\{\sigma\}$. Such a unifier σ is called *most general* whenever for all such unifiers ρ there exists a mediating substitution ρ' from $ran(\sigma)$ to $ran(\rho)$ such that $\rho = (\sigma;\rho')$. Without proof, we claim that most general unifiers exist. More precisely, if $fv(M,N) \subseteq \bar{x}$ and a unifier of M and N from $\bar{x}$ exists, then a most general unifier of M and N from $\bar{x}$ exists. In our typing rules, we make use of a fixed total function *mgu* for computing most general unifiers. This function takes in a variable set $\bar{x}$ and two messages M, N such that $fv(M,N) \subseteq \bar{x}$. It either returns a substitution σ from $\bar{x}$, or a special element $\perp$, and has the following properties:

(a) If a unifier of M and N from $\bar{x}$ exists, then $mgu(\bar{x},M,N) = \sigma$ for some σ.

(b) If $mgu(\bar{x},M,N) = \sigma$, then σ is a most general unifier of M and N from $\bar{x}$.

As shorthand, we write $mgu(E,M,N)$ for $mgu(dom(E),M,N)$. An implementation of *mgu* can be derived from a set of transformation rules for simplifying sets of equations. Almost all transformation rules are exactly as in syntactic unification (see, for instance, [10]). Here is the only rule that differs:

$$\{\{\!|M|\!\}_K = \{\!|N|\!\}_{L^{-1}}\} \cup S \quad \longrightarrow \quad \{M = N, K = \mathsf{Enc}(x), L = \mathsf{Dec}(x)\} \cup S \quad \text{where } x \text{ is fresh}$$

B Technical Properties and their Proofs

B.1 Properties of Subkinding, Kinding and Subtyping

Lemma B.1 *Subkinding is a partial order.*

Proof Immediate from the definition of subkinding. □

Lemma B.2 (Uniqueness of Kinding) *If $(T :: K)$ and $(T :: H)$, then $(K = H)$.*

Proof Immediate from the definition of kinding. □

Definition B.1 (Public and Tainted Types) A type T is called *public* iff there exists K such that $(T :: K \cup \{\mathsf{Public}\})$. A type T is called *tainted* iff there exists K such that $(T :: K \cup \{\mathsf{Tainted}\})$.

Lemma B.3 (Tainted Up-Closed, Public Down-Closed)

(a) *If T is tainted and $(T \leq U)$, then U is tainted.*

(b) *If U is public and $(T \leq U)$, then T is public.*

Proof Immediate from the definitions of subtyping, kinding and subkinding. □

For the next lemma, remember that a *nonce type* is a type of the form $(K,H)\,\mathsf{Chall}(\overline{A})$, $(K,H)\,\mathsf{Resp}(\overline{A})$ or Stale.

Lemma B.4 (Sub- and Supertypes of Nonce Types)

(a) *If T is a nonce type and $U \leq T$, then $T = U$ or U is public.*

(b) *If T is a response type and $T \leq U$, then $T = U$ or U is tainted.*

Proof By inspection of the possible reasons for $T \leq U$. Note that for part (b), we need the restriction to generative types in (Subty Top Gen). □

Lemma B.5 (Monotonicity Properties)

(a) *If $(K \leq H)$, then $(K\,\mathsf{Top} \leq H\,\mathsf{Top})$ and $(K\,\mathsf{Auth}(M) \leq H\,\mathsf{Auth}(M))$.*

(b) *If* $(T \leq U)$, $(T :: K)$ *and* $(U :: H)$, *then* $(K \leq H)$.

(c) *If* $(K,H)\,\mathsf{Chall}(\overline{A}) \leq (K,H)\,\mathsf{Chall}(\overline{B})$, *then* $\overline{A} = \overline{B}$.

(d) *If* $(K,H)\,\mathsf{Resp}(\overline{A}) \leq (K,H)\,\mathsf{Resp}(\overline{B})$, *then* $\overline{A} = \overline{B}$.

Proof Parts (a) and (b) are obvious, by inspection of the subkinding definition and the subtyping rules. To prove part (c), suppose that $(K,H)\,\mathsf{Chall}(\overline{A}) \leq (K,H)\,\mathsf{Chall}(\overline{B})$. There are two possible reasons for this judgment, namely, (Subty Refl) and (Subty Public Tainted). In the first case, $\overline{A} = \overline{B}$. In the second case, $(K,H)\,\mathsf{Chall}(\overline{A})$ is public. Then $\mathsf{Public} \in K$ and, therefore, $(K,H)\,\mathsf{Chall}(\overline{B})$ is public, too. Then $\overline{A} = \overline{B} = \emptyset$, by the syntactic restriction on public challenge types. The proof of part (d) is similar. □

Lemma B.6 *Subtyping is a preorder.*

Proof We need to show transitivity. Let $T \leq U$ by *(RuleTU)* and $U \leq V$ by *(RuleUV)*. The proof proceeds by inspecting all possible instantiations of these rulesand uses Lemmas B.3 and B.5. We omit the details. □

Lemma B.7 (Substitution Invariance)

(a) $(T :: K)$ *iff* $(T\{\sigma\} :: K)$.

(b) $(T \leq U)$ *iff* $(T\{\sigma\} \leq U\{\sigma\})$.

Proof Immediate. □

B.2 Elementary Properties

Lemma B.8 (Domain Soundness) *If* $E \vdash \diamond$ *and* $(E \vdash \mathcal{R})$, *then* $fv(\mathcal{R}) \subseteq dom(E)$.

Proof By induction on $(E \vdash \mathcal{R})$'s derivation. □

Lemma B.9 $(decrypt(M,N))\{\sigma\} = decrypt(M\{\sigma\},N\{\sigma\})$.

Proof If $N = \mathsf{Dec}\,(L)$, then both sides of the equation evaluate to $\{\!|M\{\sigma\}|\!\}_{\mathsf{Enc}(L\{\sigma\})}$. If N is not of this form nor a variable, then both sides evaluate to $\{\!|M\{\sigma\}|\!\}_{(N\{\sigma\})^{-1}}$. If $N = x$ and $\sigma(x)$ is not of the form $\mathsf{Dec}\,(L)$, then both sides evaluate to $\{\!|M\{\sigma\}|\!\}_{(N\{\sigma\})^{-1}}$. If $N = x$ and $\sigma(x) = \mathsf{Dec}\,(L)$, then both sides evaluate to $\{\!|M\{\sigma\}|\!\}_{\mathsf{Enc}(L)}$. □

Lemma B.10 (Weakening)

(a) *If* $\mathcal{D} \triangleright (E \vdash \mathcal{R})$ *and* $x \notin dom(E)$, *then there exists* $\mathcal{D}'$ *such that* $(x,E \vdash \mathcal{R})$ *and* $\mathsf{height}(\mathcal{D}') \leq \mathsf{height}(\mathcal{D})$.

(b) *If* $\mathcal{D} \triangleright (E \vdash \mathcal{R})$, *then there exists* $\mathcal{D}'$ *such that* $\mathcal{D}' \triangleright (E,A \vdash \mathcal{R})$ *and* $\mathsf{height}(\mathcal{D}') = \mathsf{height}(\mathcal{D})$.

(c) *If* $\mathcal{D} \triangleright (E \vdash \overline{A},B)$, *then there exists* $\mathcal{D}'$ *such that* $\mathcal{D}' \triangleright (E \vdash \overline{A})$ *and* $\mathsf{height}(\mathcal{D}') < \mathsf{height}(\mathcal{D})$.

Proof Part (a) is a corollary of substitutivity. Here is its proof: Suppose $(E \vdash \mathcal{R})$ and $x \notin dom(E)$. Let σ be such that $dom(\sigma) = dom(E)$, $ran(\sigma) = dom(E) \cup \{x\}$ and $\sigma(y) = y$ for all y in $dom(E)$. Then $(E\{\sigma\} \vdash \mathcal{R}\{\sigma\})$, by substitutivity. Moreover, $E\{\sigma\} = (x,E)$ and $\mathcal{R}\{\sigma\} = \mathcal{R}$. Parts (b) and (c) are proved by separate inductions on the structure of $\mathcal{D}$. □

We next prove a simple (Cut) lemma. This lemma is quite restrictive, because it only permits cutting over atomic type assertions. We will later prove a second, complementary (Cut) theorem that permits cutting over arbitrary assertions, but requires nominality of the environment.

Lemma B.11 (Atomic Cut) *If* $(\vec{x};\overline{A} \vdash \vec{n}:\vec{U},\vec{x}:\vec{V},\overline{B})$ *and* $(\vec{x};\vec{n}:\vec{U},\vec{x}:\vec{V} \vdash \overline{C})$*, then* $(\vec{x};\overline{A} \vdash \vec{n}:\vec{U},\vec{x}:\vec{V},\overline{B},\overline{C})$.

Proof By induction on $(\vec{x};\vec{n}:\vec{U},\vec{x}:\vec{V} \vdash \overline{C})$'s derivation height. Suppose $(\vec{x};\overline{A} \vdash \vec{y}:\vec{U},\vec{x}:\vec{V},\overline{B})$ and $\mathcal{D} \triangleright (\vec{x};\vec{n}:\vec{U},\vec{x}:\vec{V} \vdash \overline{C})$. Suppose $\mathcal{D}$ ends in (Lift). Then $\overline{C} = (C_0,\overline{C'})$ and C_0 is contained in $(\vec{n}:\vec{U},\vec{x}:\vec{V})$ for some C_0, $\overline{C'}$. Suppose that C_0 is contained in $(\vec{x}:\vec{V})$; the other case is similar. Thus, we assume that $C_0 = (x_0:V_0)$, $\vec{x} = (\vec{x}',x_0)$, $\vec{V} = (\vec{V}',V_0)$ and $(\vec{x};\vec{n}:\vec{U},\vec{x}':\vec{V}' \vdash \overline{C'})$ for some $\vec{x}'$, x_0, $\vec{V}'$, V_0. By weakening, $(\vec{x};\vec{n}:\vec{U},\vec{x}:\vec{V} \vdash \overline{C'})$. Because weakening does not increase the derivation height, we may apply the induction hypothesis, obtaining $(\vec{x};\overline{A} \vdash \vec{n}:\vec{U},\vec{x}:\vec{V},\overline{B},\overline{C'})$. Then $(\vec{x};\overline{A} \vdash \vec{n}:\vec{U},\vec{x}:\vec{V},\overline{B},\overline{C})$, by (Copy). The proof cases for the other right rules are all straightforward. If $\mathcal{D}$'s last rule is a left rule, then this can only be (Unsub). In this case one applies (Sub) to the left judgment and then uses the induction hypothesis. □

Our definition of matching uses a fixed function *mgu* that returns a particular most general unifier for every given environment and disagreement pair. The next lemma implies that derivability does not depend on the choice of this unification function. The lemma states that the following rules (Decrypt Trusted′) and (Decrypt Untrusted′) are admissible. These rules permit to pick the most general unifier arbitrarily.

Rules with Arbitrary Mgu:

$MGU(E,M,N) \ \stackrel{\Delta}{=}\ $ set of all most general unifiers of M and N from $dom(E)$

For $X = \exists\vec{x}\,.\,N[\overline{A}]$ where $\vec{x} \cap dom(E) = \emptyset$, define:

$$\begin{array}{lll} E, M \in' X \vdash \mathcal{R} & \stackrel{\Delta}{=} & (fv(\mathcal{R}) \subseteq dom(E)) \wedge \\ & & (\quad (MGU((\vec{x},E),M,N) = \emptyset) \vee \\ & & \quad\ (\exists\sigma \in MGU((\vec{x},E),M,N))\ ((\vec{x},E,M:\mathsf{Top},\overline{A})\{\sigma\} \vdash \mathcal{R}\{\sigma\}) \quad) \end{array}$$

(Decrypt Untrusted′)

$$\frac{\begin{array}{l}\mathsf{Tainted} \in J \quad x \notin dom(E) \cup fv(\overline{B}) \qquad E \vdash N : (K,H)\,\mathsf{DK}(X) \\ (x,E,(M,x) \in' X \vdash \overline{B}) \ \vee\ (\mathsf{Public},\mathsf{Tainted} \in K \cup H^{-1}) \\ (E, M : J\,\mathsf{Top} \vdash \overline{B}) \ \vee\ (\mathsf{Public} \notin K \cup H^{-1})\end{array}}{E, decrypt(M,N) : J\,\mathsf{Top} \vdash \overline{B}}$$

(Decrypt Trusted′)

$$\frac{\begin{array}{l}\mathsf{Tainted} \notin H \cup J \\ E \vdash N : (K,H)\,\mathsf{DK}(X) \quad E,(M,L) \in' X \vdash \overline{B}\end{array}}{E, decrypt(M,N) : J\,\mathsf{Auth}(L) \vdash \overline{B}}$$

Lemma B.12 *The rules* (Decrypt Trusted′) *and* (Decrypt Trusted′) *are admissible.*

Proof We prove (Decrypt Trusted′). The proof for (Decrypt Untrusted′) is similar. Let $\vec{x} \cap dom(E) = \emptyset$, $\mathsf{Tainted} \notin H \cup J$ and assume:

(1) $E \vdash N : (K,H)\,\mathsf{DK}(\exists\vec{x}\,.\,M_0[\overline{A}])$ assumption
(2) $E,\,(M,L) \in' \exists\vec{x}\,.\,M_0[\overline{A}] \vdash \overline{B}$ assumption

We want to construct a derivation of $(E, decrypt(M,N) : J\,\mathsf{Auth}(L) \vdash \overline{B})$ whose last rule is (Decrypt Trusted). To this end, we need to show:

(3) $E,\,(M,L) \in \exists\vec{x}\,.\,M_0[\overline{A}] \vdash \overline{B}$ goal

We apply the definition of $\in'$ to assumption (2): First note that $fv(\overline{B}) \subseteq dom(E)$, by this definition. If $MGU((\vec{x},E),(M,L),M_0) = \emptyset$, then $mgu((\vec{x},E),(M,L),M_0) = \bot$ and goal (3) follows, by definition of matching. So assume there exists a σ in $MGU((\vec{x},E),(M,L),M_0)$ such that:

(4) $(\vec{x},E,(M,L):\mathsf{Top},\overline{A})\{\sigma\} \vdash \overline{B}\{\sigma\}$

Because (M,L) and M_0 are unifiable, there exist a ρ such that:

(5) $mgu((\vec{x},E),(M,L),M_0) = \rho$

Because σ is a most general unifier, there exists a mediating substitution ρ' such that $(\sigma;\rho') = \rho$. By substitutivity, we may apply ρ' to judgment (4) obtaining:

(6) $(\vec{x}, E, (M,L) : \mathsf{Top}, \bar{A})\{\sigma; \rho'\} \vdash \bar{B}\{\sigma; \rho'\}$
(7) $(\vec{x}, E, (M,L) : \mathsf{Top}, \bar{A})\{\rho\} \vdash \bar{B}\{\rho\}$ because $(\sigma; \rho') = \rho$

Now, our proof goal (3) follows from (5) and (7), by definition of matching. □

B.3 Opponent Typability

Lemma B.13 (Opponent Rules)

(a) *If* $(E \vdash M : \mathsf{Un}, \bar{B})$ *and* $k \in \{\mathsf{Enc}, \mathsf{Dec}\}$*, then* $(E \vdash k(M) : \mathsf{Un}, \bar{B})$

(b) *If* $(E \vdash N : \mathsf{Un}, M : \mathsf{Un}, \bar{B})$*, then* $(E \vdash \{\!|M|\!\}_N : \mathsf{Un}, \bar{B})$.

(c) *If* $(E \vdash N : \mathsf{Un})$ *and* $(E, M : \mathsf{Un} \vdash \bar{B})$*, then* $(E, \mathit{decrypt}(M,N) : \mathsf{Un} \vdash \bar{B})$.

Proof *Part* (a)*:* Suppose $(E \vdash M : \mathsf{Un}, \bar{B})$ and $k \in \{\mathsf{Enc}, \mathsf{Dec}\}$. Let $K = H = \{\mathsf{Public}, \mathsf{Tainted}\}$, $T = (K,H)\,\mathsf{KP}(\_)$ and $U_k = (K,H)\,k\mathsf{K}(\_)$. By (Sub), $(E \vdash M : T, \bar{B})$. Then $(E \vdash k(M) : U_k, \bar{B})$, by ($k$ Part). Then $(E \vdash k(M) : \mathsf{Un}, \bar{B})$, by (Sub).

Parts (b) and (c) are proved similarly. □

For $\vec{M} = (M_1, \ldots, M_k)$, let $(\vec{M} : T)$ denote the assertion set $(M_1 : T, \ldots, M_k : T)$. For $\vec{M} = (M_1, \ldots, M_k)$ and $\vec{T} = (T_1, \ldots, T_k)$, let $(\vec{M} : \vec{T})$ denote the assertion set $(M_1 : T_1, \ldots, M_k : T_k)$.

Proof of Lemma 7.1(Dolev–Yao ⇒ Typability) *If* $(\vec{M} \Vdash \vec{N})$*, then* $(\vec{M} : \mathsf{Un} \vdash \vec{N} : \mathsf{Un})$.

Proof By induction on $(\vec{M} \Vdash \vec{N})$'s derivation, using Lemma B.13. □

Lemma B.14 (Message Typability) *If* $\vec{M}$ *are implementable messages,* $\mathit{fv}(\vec{M}) \subseteq \vec{x}$ *and* $\mathit{fn}(\vec{M}) \subseteq \vec{n}$*, then* $(\vec{n} : \mathsf{Un}, \vec{x} : \mathsf{Un} \vdash \vec{M} : \mathsf{Un})$.

Proof By induction on the structure of $\vec{M}$, using Lemma B.13. □

Lemma B.15 (Properties of $\Vdash$)

(a) *If* $(\bar{M} \Vdash \bar{N})$*, then* $(\bar{M}, L \Vdash \bar{N})$. (Weaken Left)

(b) *If* $(\bar{M} \Vdash \bar{N}, L)$*, then* $(\bar{M} \Vdash \bar{N})$. (Weaken Right)

(c) *If* $(\bar{M}_1 \Vdash \bar{N}_1, \bar{n}, \bar{x})$ *and* $(\bar{M}_2, \bar{n}, \bar{x} \Vdash \bar{N}_2)$*, then* $(\bar{M}_1, \bar{M}_2 \Vdash \bar{n}, \bar{x}, \bar{N}_1, \bar{N}_2)$. (Atomic Cut)

Proof The first two parts are proved, separately, by induction on the derivations. Both inductions are straightforward. (Atomic Cut) is proved by induction on $(\bar{M}_2, \bar{n}, \bar{x} \Vdash \bar{N}_2)$'s derivation height. The proof is very similar to the proof of Lemma B.11. □

Proof of Theorem 7.1 (Opponent Typability) *If O is an opponent process,* $\mathit{fn}(O) \subseteq \vec{n}$ *and* $\mathit{fv}(O) \subseteq \vec{x}$*, then* $(\vec{x}; \vec{n} : \mathsf{Un}, \vec{x} : \mathsf{Un} \vdash O)$.

Proof We show the following more general statement by induction on the structure of P:

If P is an opponent process, $\mathit{fn}(P) \subseteq \vec{n}$*,* $\mathit{fv}(P) \subseteq \vec{x}$ *and* $(\vec{M} \Vdash \vec{n}, \vec{x})$*, then* $(\vec{x}; \vec{M} : \mathsf{Un} \vdash P)$.

Suppose P is an opponent process, $fn(P) \subseteq \vec{n}$, $fv(P) \subseteq \vec{x}$ and $(\vec{M} \Vdash \vec{n},\vec{x})$.

Suppose $P = (\mathsf{out}\ N\ L)$ for some N, L. By Lemma 7.1, $(\vec{x}; \vec{M} : \mathsf{Un} \vdash \vec{n} : \mathsf{Un}, \vec{x} : \mathsf{Un})$. By Lemma B.14, $(\vec{x}; \vec{n} : \mathsf{Un}, \vec{x} : \mathsf{Un} \vdash N : \mathsf{Un}, L : \mathsf{Un})$. Then $(\vec{x}; \vec{M} : \mathsf{Un} \vdash N : \mathsf{Un}, L : \mathsf{Un})$, by atomic cut (Lemma B.11). Then $(\vec{x}; \vec{M} : \mathsf{Un} \vdash P)$, by (Proc Out).

Suppose $P = (\mathsf{inp}\ N\ \exists\vec{y}\,.\,L[L : \mathsf{Un}]; Q)$, where $\vec{y} \cap \vec{x} = \emptyset$. By Lemma 7.1, $(\vec{x}; \vec{M} : \mathsf{Un} \vdash \vec{n} : \mathsf{Un}, \vec{x} : \mathsf{Un})$. By Lemma B.14, $(\vec{x}; \vec{n} : \mathsf{Un}, \vec{x} : \mathsf{Un} \vdash N : \mathsf{Un})$. Then $(\vec{x}; \vec{M} : \mathsf{Un} \vdash N : \mathsf{Un})$, by atomic cut (Lemma B.11). Trivially, $(\vec{x},\vec{y}; \vec{M} : \mathsf{Un}, L : \mathsf{Un} \vdash L : \mathsf{Un})$. Because $\exists\vec{y}\,.\,L[L : \mathsf{Un}]$ is an implementable pattern, $(\vec{n},\vec{x},L \Vdash \vec{y})$. By Lemma B.15, we may cut $(\vec{M} \Vdash \vec{n},\vec{x})$ with $(\vec{n},\vec{x},L \Vdash \vec{y})$, obtaining $(\vec{M},L \Vdash \vec{n},\vec{x},\vec{y})$. Then, by induction hypothesis, $(\vec{x},\vec{y}; \vec{M} : \mathsf{Un}, L : \mathsf{Un} \vdash Q)$. At this point, we have shown that $(\vec{x}; \vec{M} : \mathsf{Un} \vdash N : \mathsf{Un})$, $(\vec{x},\vec{y}; \vec{M} : \mathsf{Un}, L : \mathsf{Un} \vdash L : \mathsf{Un})$ and $(\vec{x},\vec{y}; \vec{M} : \mathsf{Un}, L : \mathsf{Un} \vdash Q)$. Therefore, $(\vec{x}; \vec{M} : \mathsf{Un} \vdash P)$, by (Proc In).

The other proof cases are straightforward. □

B.4 Inversion Properties

Lemma B.16 (Inverting Begun- and Fresh-Assertions) *If E is nominal, then all of the following statements hold:*

(a) $(E \vdash !\mathsf{fresh}(N : T), \bar{B})$ *is false.*

(b) *If* $(E \vdash \mathsf{fresh}(N : T), \bar{B})$*, then* $E = (E', \mathsf{fresh}(N : T))$ *and* $(E' \vdash \bar{B})$ *for some* E'.

(c) *If* $(E \vdash !\mathsf{begun}(M), \bar{B})$*, then* $E = (E', !\mathsf{begun}(M))$ *and* $(E \vdash \bar{B})$ *for some* E'.

(d) *If* $(E \vdash \mathsf{begun}(M), \bar{B})$*, then either* $E = (E', !\mathsf{begun}(M))$ *and* $(E \vdash \bar{B})$ *or* $E = (E', \mathsf{begun}(M))$ *and* $(E' \vdash \bar{B})$ *for some* E'.

Proof For part (a), one proves the following statement by induction on $\mathcal{D}$: *If* $\mathcal{D} \rhd (E \vdash A, \bar{B})$*, then* $A \neq !\mathsf{fresh}(N : T)$ *for all* N, T. The other three parts are proved separately and in order by inductions on the derivations. □

The following lemma is a straightforward consequence of Lemma B.16.

Lemma B.17 (Inverting Sets of Non-Copyables) *If E is nominal, $(E \vdash \bar{A},\bar{B})$ and no member of $\bar{B}$ is copyable, then there exist $\bar{B}_0$, $\bar{B}_1$, E_0 such that $\bar{B} = (\bar{B}_0,\bar{B}_1)$, $E = (E_0,\bar{B}_1)$, $!C_0 \in E_0$ for all C_0 in $\bar{B}_0$, and $(E_0 \vdash \bar{A})$.*

Proof By induction on the size of $\bar{B}$, using Lemma B.16. □

Lemma B.18 (Inverting Type Assertions for Composite Messages) *If E is nominal, then all of the following statements hold:*

(a) *If* $(E \vdash (M,N) : K\,\mathsf{Top}, \bar{B})$*, then* $(E \vdash M : K\,\mathsf{Top}, N : K\,\mathsf{Top}, \bar{B})$.

(b) *If* $(E \vdash \mathsf{Enc}(M) : (K,H)\,\mathsf{EK}(X), \bar{B})$,
then there exist K',H',X' *such that* $(E \vdash M : (K',H')\,\mathsf{KP}(X'), \bar{B})$
and either $(K',H',X') = (K,H,X)$ *or* $(\mathsf{Public} \in K' \cap K^{-1})$.

(c) *If* $(E \vdash \mathsf{Dec}(M) : (K,H)\,\mathsf{DK}(X), \bar{B})$,
then there exist K',H',X' *such that* $(E \vdash M : (K',H')\,\mathsf{KP}(X'), \bar{B})$
and either $(K',H',X') = (K,H,X)$ *or* $(\mathsf{Public} \in H' \cap H^{-1})$.

(d) *If* $(E \vdash M : (K,H)\,\mathsf{KP}(X), \bar{B})$,
then either $(\mathsf{Tainted} \in K \cap H)$ *or* $E = (E', M : (K,H)\,\mathsf{KP}(X))$ *for some* E'.

(e) $(E \vdash \{\!|M|\!\}_{N^{-1}} : T, \overline{B})$ *is false.*

(f) *If* $(E \vdash \{\!|M_1|\!\}_N : J\,\mathsf{Auth}(M_2), \overline{B})$*, then there exist* K,H,X,M_2' *such that the following statements hold:*

- $E \vdash N : (K,H)\,\mathsf{EK}(X), (M_1,M_2') \in X, \overline{B}$
- $(\mathsf{Tainted} \notin J) \;\Rightarrow\; (M_2' = M_2)$
- $(\mathsf{Tainted} \in K \cup H^{-1})$
  $\Rightarrow\; (E \vdash N : (K,H)\,\mathsf{EK}(X), (M_1,M_2') \in X, M_1 : (J \cup \{\mathsf{Tainted}\})\,\mathsf{Top}, \overline{B})$
- $(\mathsf{Tainted} \in K) \;\Rightarrow\; (\mathsf{Tainted} \in J)$

(g) *If* $(E \vdash \{\!|M_1|\!\}_N : J\,\mathsf{Top}, \overline{B})$*, then there exist* K,H,X,M_2 *such that the following statements hold:*

- $E \vdash N : (K,H)\,\mathsf{EK}(X), (M_1,M_2) \in X, \overline{B}$
- $(\mathsf{Tainted} \in K \cup H^{-1})$
  $\Rightarrow\; (E \vdash N : (K,H)\,\mathsf{EK}(X), (M_1,M_2) \in X, M_1 : (J \cup \{\mathsf{Tainted}\})\,\mathsf{Top}, \overline{B})$
- $(\mathsf{Tainted} \in K) \;\Rightarrow\; (\mathsf{Tainted} \in J)$

Proof *Part* (a)*:* This part follows from the first of the following two auxiliary statements, which we prove simultaneously by induction on $\mathcal{D}$: *If E is nominal and* $\mathcal{D} \rhd (E \vdash (M,N) : T, \overline{B})$*, then the following statements hold:*

(a) *If* $T \leq K\,\mathsf{Top}$*, then* $(E \vdash M : K\,\mathsf{Top}, N : K\,\mathsf{Top}, (M,N) : T, \overline{B})$.

(b) *If* $T \notin \{\mathsf{Top}\} \cup \{K\,\mathsf{Top} \mid K \text{ is a kind}\}$*, then* $(E \vdash M : \mathsf{Un}, N : \mathsf{Un}, (M,N) : T, \overline{B})$.

We omit the proof details.

Part (b)*:* This part follows from the second of the following two auxiliary statements: *If E is nominal and* $\mathcal{D} \rhd (E \vdash \mathsf{Enc}(M) : T, \overline{B})$*, then the following statements hold:*

(a) *If $T \leq U$ and U is a nonce type,*
then $(E \vdash M : (K',H')\,\mathsf{KP}(X'), \mathsf{Enc}(M) : T, \overline{B})$ *for some K', H', X' such that* $\mathsf{Public} \in K'$.

(b) *If* $(T \leq (K,H)\,\mathsf{EK}(X))$,
then there exist K',H',X' such that $(E \vdash M : (K',H')\,\mathsf{KP}(X'), \mathsf{Enc}(M) : T, \overline{B})$
and either $(K',H',X') = (K,H,X)$ *or* $(\mathsf{Public} \in K' \cap K^{-1})$.

We prove these two statements, separately but in order, by induction on $\mathcal{D}$. We omit the details.

Part (c)*:* This proof is very similar to the proof of part (b).

Part (d)*:* By a straightforward induction on $(E \vdash M : (K,H)\,\mathsf{KP}(X), \overline{B})$'s derivation. In the proof cases for (Sub) and (Unsub), one uses that only tainted key types have proper subtypes.

Part (e)*:* One shows the following statement by induction on $\mathcal{D}$: *If E is nominal and* $\mathcal{D} \rhd (E \vdash M : T, \overline{B})$*, then M is not of the form* $M = \{\!|N|\!\}_{L^{-1}}$ *for any N,L.* The proof of this statement is entirely straightforward. Observe, however, that it requires the nominality of E. Without nominality, E could contain type assertions of the form $(\{\!|N|\!\}_{L^{-1}} : T)$.

Parts (f) *and* (g)*:* To prove these, we show the following statements simultaneously by induction on $\mathcal{D}$: *If E is nominal and* $\mathcal{D} \rhd (E \vdash \{\!|M_1|\!\}_N : T, \overline{B})$*, then there exist K,H,X,M_2' such that all of the following statements hold:*

(a) $E \vdash N : (K,H)\,\mathsf{EK}(X), (M_1,M_2') \in X, \{\!|M_1|\!\}_N : T, \overline{B}$

(b) *If* $(T \leq J\,\mathsf{Auth}(M_2))$ *and* $(\mathsf{Tainted} \notin J)$*, then* $(M_2' = M_2)$.

(c) *If* $(T \leq J\,\mathsf{Top})$ *and* $(\mathsf{Tainted} \in K \cup H^{-1})$,
then $(E \vdash N : (K,H)\,\mathsf{EK}(X),\, (M_1,M_2') \in X,\, M_1 : (J \cup \{\mathsf{Tainted}\})\,\mathsf{Top},\, \{\!|M_1|\!\}_N : T,\, \overline{B})$.

(d) *If* $(T \leq U)$*,* U *is a nonce type and* $(\mathsf{Tainted} \in K \cup H^{-1})$,
then $(E \vdash N : (K,H)\,\mathsf{EK}(X),\, (M_1,M_2') \in X,\, M_1 : \mathsf{Un},\, \{\!|M_1|\!\}_N : T,\, \overline{B})$.

(e) *If* $(T \leq J\,\mathsf{Top})$ *and* $(\mathsf{Tainted} \in K)$*, then* $\mathsf{Tainted} \in J$.

Suppose E is nominal and $\mathcal{D} \rhd (E \vdash \{\!|M_1|\!\}_N : T, \overline{B})$. The left rules do not cause a problem for the induction, because none of the statements requires or claims any properties of the environment. Because E is nominal, (Id) and (Lift) are unproblematic. The rule (Sub) is unproblematic, because the premises of statements (a) to (d) are closed under subtyping T and the conclusions of these statements are closed under supertyping T. The rule (Copy) is unproblematic, because the judgments that occur in (a), (c) and (d) keep the type assertion $(\{\!|M_1|\!\}_N : T)$. Thus, the only interesting proof cases are (Nonce Cast), (Weaken Chall), (Strengthen Resp), (Encrypt Trusted) and (Encrypt Untrusted). For convenience, we define the following predicate P: Let $P(T,K,H,X,M_2',\overline{B})$ be true iff statements (a) through (d) hold (for E and M_1 as fixed above).

Suppose $\mathcal{D}$ ends in (Nonce Cast), $T = (K',H')\,\mathsf{Resp}()$ and $(E \vdash \{\!|M_1|\!\}_N : (K',H')\,\mathsf{Chall}(),\, \overline{B})$ for some K', H'. Let $T' = (K',H')\,\mathsf{Chall}()$. By induction hypothesis, there exist K,H,X,M_2' such that $P(T',K,H,X,M_2',\overline{B})$. We will show $P(T,K,H,X,M_2',\overline{B})$: Part (a) of $P(T,K,H,X,M_2',\overline{B})$ follows from part (a) of $P(T',K,H,X,M_2',\overline{B})$ by (Nonce Cast), where (Nonce Cast) is used to convert T' to T. Similarly, part (d) of $P(T,K,H,X,M_2',\overline{B})$ follows from part (d) of $P(T',K,H,X,M_2',\overline{B})$ by (Nonce Cast). Because proper supertypes of response types are tainted, by Lemma B.4, parts (e) and (b) of $P(T,K,H,X,M_2',\overline{B})$ obviously hold. Part (c) of $P(T,K,H,X,M_2',\overline{B})$ follows from part (d) of $P(T,K,H,X,M_2',\overline{B})$ by (Sub), where (Sub) is used to convert Un to $(J \cup \{\mathsf{Tainted}\})\,\mathsf{Top}$.

Suppose $\mathcal{D}$ ends in (Weaken Chall), $T = (K',H')\,\mathsf{Chall}(\overline{A})$ and $(E \vdash \{\!|M_1|\!\}_N : (K',H')\,\mathsf{Chall}(\overline{A},C),\, C,\, \overline{B})$ for some K', H', $\overline{A}$, C. Let $T' = (K',H')\,\mathsf{Chall}(\overline{A},C)$. By induction hypothesis, there exist K,H,X,M_2' such that $P(T',K,H,X,M_2',(C,\overline{B}))$. We will show $P(T,K,H,X,M_2',\overline{B})$: Part (a) of $P(T,K,H,X,M_2',\overline{B})$ follows from part (a) of $P(T',K,H,X,M_2',(C,\overline{B}))$ by (Weaken Chall). Similarly, part (d) of $P(T,K,H,X,M_2',\overline{B})$ follows from part (d) of $P(T',K,H,X,M_2',(C,\overline{B}))$ by (Weaken Chall). Parts (e) and (b) of $P(T,K,H,X,M_2',\overline{B})$ follow from the same parts of $P(T',K,H,X,M_2',(C,\overline{B}))$, because $T < U$ implies $T' \leq U$ for all U. Part (c) of $P(T,K,H,X,M_2',\overline{B})$ follows from part (d) of $P(T,K,H,X,M_2',\,\overline{B})$ by (Sub), where (Sub) is used to convert Un to $(J \cup \{\mathsf{Tainted}\})\,\mathsf{Top}$.

The proof case (Strengthen Resp) is essentially identical to proof case (Weaken Chall).

Suppose $\mathcal{D}$ ends in (Encrypt Trusted), $T = \mathsf{Public}(M_2')$, $(E \vdash N : (K,H)\,\mathsf{EK}(X),\, (M_1,M_2') \in X,\, \overline{B})$ and $\mathsf{Tainted} \notin K \cup H^{-1}$ for some K, H, X, M_2'. We show $P(T,K,H,X,M_2')$: By (Copy) and (Encrypt Trusted), we obtain part (a). Parts (c), (e) and (d) holds vacuously, because $\mathsf{Tainted} \notin K$. To prove part (b), suppose that $T \leq J\,\mathsf{Auth}(M_2)$. There are three possible reasons for $T \leq J\,\mathsf{Auth}(M_2)$, namely, (Subty Refl), (Subty Auth) and (Subty Public Tainted). In the first two cases, $M_2' = M_2$, and in the third case, $\mathsf{Tainted} \in J$.

Suppose $\mathcal{D}$ ends in (Encrypt Untrusted), $T = J'\,\mathsf{Auth}(M_2')$, $(E \vdash N : (K,H)\,\mathsf{EK}(X),\, (M_1,M_2') \in X,\, M_1 : J''\,\mathsf{Top},\, \overline{B})$, $J' = (J'' - \{\mathsf{Tainted}\}) \cup (K - \{\mathsf{Public}\})$, and $\mathsf{Tainted} \in K \cup H^{-1}$. We show $P(T,K,H,X,M_2')$: By (Copy) and (Encrypt Untrusted), we obtain:

(1) $E \vdash N : (K,H)\,\mathsf{EK}(X),\, (M_1,M_2') \in X,\, M_1 : J''\,\mathsf{Top},\, \{\!|M_1|\!\}_N : T,\, \overline{B}$

By weakening, we get part (a). Now observe that $J'' \leq J' \cup \{\mathsf{Tainted}\}$. To see this, note, on the one hand, that $\mathsf{Tainted} \in J''$ implies $\mathsf{Tainted} \in J' \cup \{\mathsf{Tainted}\}$, trivially. On the other hand, it follows from $J' = (J'' - \{\mathsf{Tainted}\}) \cup (K - \{\mathsf{Public}\})$ that $\mathsf{Public} \in J' \cup \{\mathsf{Tainted}\}$ implies $\mathsf{Public} \in J''$.

To show parts (c) and (e), suppose $T \leq J\,\mathsf{Top}$ for some U. Then $J' \leq J$, by monotonicity of kinding. Then $J' \cup \{\mathsf{Tainted}\} \leq J \cup \{\mathsf{Tainted}\}$. Therefore, $J''\,\mathsf{Top} \leq (J' \cup \{\mathsf{Tainted}\})\,\mathsf{Top} \leq (J \cup \{\mathsf{Tainted}\})\,\mathsf{Top}$. Thus, part (c) follows from (1), by (Sub). To show part (e), suppose, in addition, that $\mathsf{Tainted} \in K$. Then $\mathsf{Tainted} \in (J'' - \{\mathsf{Tainted}\}) \cup (K - \{\mathsf{Public}\}) = J'$. Then $\mathsf{Tainted} \in J$, because $J' \leq J$.

To show part (d), suppose that $T \leq U$ for some nonce type U. Then $\mathsf{Public} \in J'$, because proper subtypes of nonce types are public. Therefore, $J'' \,\mathsf{Top} \leq (J' \cup \{\mathsf{Tainted}\})\,\mathsf{Top} = \{\mathsf{Public}, \mathsf{Tainted}\}\,\mathsf{Top} = \mathsf{Un}$. Thus, part (d) follows from (1), by (Sub).

Finally, to show part (b), suppose that $T \leq J\,\mathsf{Auth}(M_2)$. The possible reasons for $T \leq J\,\mathsf{Auth}(M_2)$ are (Subty Refl), (Subty Auth) and (Subty Public Tainted). In the first two cases, $M_2' = M_2$, and in the third case, $\mathsf{Tainted} \in J$. $\square$

We still have to deal with judgments for name type assertion, i.e., judgments of the form $(E \vdash n : T, \overline{B})$. One may expect that $(E \vdash n : T, \overline{B})$ implies $E = (E', n : U)$ and $U \leq T$ for some E', U. This is not quite true in our type system, because $(n : T)$ may have been obtained from a type assertion $(n : U)$ by the rules (Nonce Cast), (Weaken Chall) and (Strengthen Resp). Moreover, our weak functionality requirement on nominal environments permits the assignment of several different types to the same name (as long as all of these are supertypes of Stale).

Definition B.2 (n-stale Environments) A nominal environment E is called *n-stale* iff it only assigns supertypes of Stale to n, i.e., $E = (E', n : T)$ implies $\mathsf{Stale} \leq T$.

Definition B.3 (n-ambiguous Environments) A nominal environment E is called *n-ambiguous* iff either $E = (E', n : T, n : U)$ or $E = (E', \mathsf{fresh}(n : (K,H)\,\mathsf{Chall}(\overline{A})))$ for some E', T, U, K, H, $\overline{A}$.

Lemma B.19 (Inverting Type Assertions for Names) *If E is nominal, then all of the following statements hold:*

(a) *If $(E \vdash n : T, \overline{B})$, then $E = (E', n : U)$ for some E', U.*

(b) *If E is n-stale and $(E \vdash n : T, \overline{B})$, then* $\mathsf{Stale} \leq T$.

(c) *If $E = (E', n : T)$ and $(E \vdash n : U, \overline{B})$, then $T \leq U$, or T is a challenge type, or U is a response type, or E is n-ambiguous,* $\mathsf{Stale} \leq T$ *and* $\mathsf{Stale} \leq U$.

Proof Parts (a) and (b) are proved separately by straightforward inductions on the derivation. Part (c) requires to first prove a sequence of auxiliary claims. To state these, we make the following definition: A nominal environment E is called *n-unique* iff it only assigns one type to n, i.e., $E = (E', n : T, n : U)$ is false. Note that there is only one typing rule that may turn n-unique nominal environments into environments that are not n-unique, namely, the rule (Discard Chall). Essentially, the proof strategy is to first show Lemma B.19(c) for n-unique environments and for derivations that do not make critical use of the rule (Discard Chall). We prove the following statements, the last of which is part (c) of the lemma:

(a) *If $\mathcal{D} \rhd (E \vdash \overline{A})$ and $\mathcal{D}$ makes use of rule (Discard Chall) with the rule scheme's meta-variable N instantiated by n, then $E = (E', \mathsf{fresh}(n : (K,H)\,\mathsf{Chall}(\overline{B})))$ for some E', K, H, $\overline{B}$.*

(b) *If $E = (E', n : T)$ is nominal and n-unique, $\mathcal{D} \rhd (E \vdash n : U_0, \overline{B})$, $U_0 \leq U$ and $\mathcal{D}$ does not make use of rule (Discard Chall) with the rule scheme's meta-variable N instantiated by n, then $T \leq U$ or T is public or T is a challenge type.*

(c) *If $T \leq T_0$, $E = (E', n : T_0)$ is nominal and n-unique, $\mathcal{D} \rhd (E \vdash n : U, \overline{B})$ and $\mathcal{D}$ does not make use of rule (Discard Chall) with the rule scheme's meta-variable N instantiated by n, then $T \leq U$ or U is tainted or U is a response type.*

(d) *If $E = (E', n : T)$ is nominal and n-unique, $\mathcal{D} \rhd (E \vdash n : U, \overline{B})$ and $\mathcal{D}$ does not make use of rule (Discard Chall) with the rule scheme's meta-variable N instantiated by n, then $T \leq U$ or T is a challenge type or U is a response type.*

(e) *If $E = (E', n : T)$ is nominal and $(E \vdash n : U, \overline{B})$, then $T \leq U$, or T is a challenge type, or U is a response type, or E is n-ambiguous,* $\mathsf{Stale} \leq T$ *and* $\mathsf{Stale} \leq U$.

Statements (a), (b) and (c) are proved separately by inductions on $\mathcal{D}$. Statement (d) is proved by an induction on $\mathcal{D}$ and uses (b) and (c). □

B.5 Key Uniqueness

Proof of Lemma 7.6 (Key Uniqueness) *If E is nominal, $E \vdash \mathsf{Enc}(N) : (K,H)\,\mathsf{EK}(X)$ and $E \vdash \mathsf{Dec}(N) : (K',H')\,\mathsf{DK}(X')$, then $(K,H,X) = (K',H',X')$ or* $\mathsf{Tainted} \in (K \cap K'^{-1}) \cup (H^{-1} \cap H') \cup (K \cap H')$

Proof Suppose E is nominal. We first show the following auxiliary statements.

(a) *If $(E \vdash \mathsf{Enc}(N) : (K,H)\,\mathsf{EK}(X))$ and* $\mathsf{Tainted} \notin K$, *then $E = (E', N : (K,H)\,\mathsf{KP}(X))$ for some E'.*

(b) *If $(E \vdash \mathsf{Dec}(N) : (K,H)\,\mathsf{DK}(X))$ and* $\mathsf{Tainted} \notin H$, *then $E = (E', N : (K,H)\,\mathsf{KP}(X))$ for some E'.*

(c) *If $E = (E', n : (K,H)\,\mathsf{KP}(X))$ and $(E \vdash \mathsf{Enc}(n) : (K',H')\,\mathsf{EK}(X'))$, then* $\mathsf{Tainted} \in K' \cap (K^{-1} \cup H)$ *or* $(K,H,X) = (K',H',X')$.

(d) *If $E = (E', n : (K,H)\,\mathsf{KP}(X))$ and $(E \vdash \mathsf{Dec}(n) : (K',H')\,\mathsf{DK}(X'))$, then* $\mathsf{Tainted} \in H' \cap (H^{-1} \cup K)$ *or* $(K,H,X) = (K',H',X')$.

Auxiliary statement (a)*:* Let $(E \vdash \mathsf{Enc}(N) : (K,H)\,\mathsf{EK}(X))$ and $(\mathsf{Tainted} \notin K)$. Let $T = (K,H)\,\mathsf{KP}(X)$. Then $(E \vdash N : T)$, by Lemma B.18(b). Then $E = (E', N : T)$ for some E', by Lemma B.18(d).

Auxiliary statement (b) is proved similarly.

Auxiliary statement (c)*:* Let $T = (K,H)\,\mathsf{KP}(X)$, $T' = (K',H')\,\mathsf{EK}(X')$, $E = (E', n : T)$ and $(E \vdash \mathsf{Enc}(n) : T')$. By Lemma B.18(b), there exist T'',K'',H'',X'' such that $T'' = (K'',H'')\,\mathsf{KP}(X'')$, $(E \vdash n : T'')$ and either $(K'',H'',X'') = (K',H',X')$ or $(\mathsf{Public} \in K'' \cap K'^{-1})$. By Lemma B.19(c), either $T \leq T''$ or both $\mathsf{Stale} \leq T$ and $\mathsf{Stale} \leq T''$.

Suppose first that both $\mathsf{Stale} \leq T$ and $\mathsf{Stale} \leq T''$. Then $\mathsf{Tainted} \in K'' \cap H$, because the only possible reason for key types being supertypes of Stale is (Subty Public Tainted). Suppose, towards a contradiction, that $\mathsf{Tainted} \notin K'$. Then $K'' = K'$, because $(\mathsf{Public} \notin K'' \cap K'^{-1})$. Then $\mathsf{Tainted} \in K'' = K'$, a contradiction. Thus $\mathsf{Tainted} \in K' \cap H$.

Suppose now that $T \leq T''$. It suffices to show that $(\mathsf{Public} \notin K)$ implies $(K,H,X) = (K',H',X')$ and that $(\mathsf{Tainted} \notin K')$ implies $(K,H,X) = (K',H',X')$. Suppose first that $(\mathsf{Public} \notin K)$. Then the only possible reason for $(T \leq T'')$ is reflexivity. Thus, $(T = T'')$. From $(T = T'')$ it follows that $(K,H,X) = (K'',H'',X'')$. Thus, $(\mathsf{Public} \notin K'')$. Then $(K'',H'',X'') = (K',H',X')$, because $(\mathsf{Public} \notin K'' \cap K'^{-1})$. Thus, $(K,H,X) = (K'',H'',X'') = (K',H',X')$. Suppose now that $(\mathsf{Tainted} \notin K')$. Then $(K'',H'',X'') = (K',H',X')$, because $(\mathsf{Public} \notin K'' \cap K'^{-1})$. Then $\mathsf{Tainted} \notin K''$. Then the only possible reason for $(T \leq T'')$ is reflexivity. Thus, $(T = T'')$. Thus, $(K,H,X) = (K'',H'',X'') = (K',H',X')$.

Auxiliary statement (d) is proved similarly.

Main statement: Let $(E \vdash \mathsf{Enc}(N) : (K,H)\,\mathsf{EK}(X))$, $(E \vdash \mathsf{Dec}(N) : (K',H')\,\mathsf{DK}(X'))$, $(K,H,X) \neq (K',H',X')$ and $\mathsf{Tainted} \notin K \cap H'$. We will show that $\mathsf{Tainted} \in (K \cap K'^{-1}) \cup (H^{-1} \cap H')$. Suppose first that $\mathsf{Tainted} \notin K$. Then $E = (E', N : (K,H)\,\mathsf{KP}(X))$ for some E', by auxiliary statement (a). Then $\mathsf{Tainted} \in H' \cap H^{-1}$, by auxiliary statement (d). Suppose now that $\mathsf{Tainted} \notin H'$. Then $E = (E', N : (K',H')\,\mathsf{KP}(X'))$ for some E', by auxiliary statement (b). Then $\mathsf{Tainted} \in K \cap K'^{-1}$, by auxiliary statement (c). □

B.6 Nonce Safety

A name that was generated with a challenge type can have many different types due to the subsumption rules (Sub) and (Unsub) and the nonce rules (Weaken Chall), (Nonce Cast) and (Strengthen Resp). It is, for instance, possible to apply a sequence of several nonce casts to the same nonce challenge: First cast a nonce's challenge type to a response type using (Nonce Cast), then cast the response type back to a challenge type by the subsumption rule (Sub), then apply (Nonce Cast) to get a response type again, and so on. A rule sequence like this may also be combined with intermediate applications of (Weaken Chall) and (Strengthen Resp). Such undirected combinations of nonce-casting and subsumption are quite pointless. However, our safety proof has to deal with these possibilities. In the inductive proof of the nonce safety lemma (Lemma 7.7), it is sometimes necessary to skip inside a derivation to a previous use of (Nonce Cast). Technically, this is achieved by case (f) of the following Lemma B.20. Cases (a), (b) and (c) collect possible types of a nonce challenge before the first nonce cast and if (d) does not hold. Case (e) accounts for the (Copy) rule.

Lemma B.20 (Possible Challenge Types) *If* $(E, \mathsf{fresh}(n : (K,H)\,\mathsf{Chall}(\overline{A})))$ *is nominal and* $\mathcal{D} \rhd (E \vdash n : T, \overline{B})$, *then one of the following holds:*

(a) $T = (K,H)\,\mathsf{Chall}(\overline{C})$ *for some submultiset* $\overline{C}$ *of* $\overline{A}$.

(b) $T = K'\,\mathsf{Top}$ *and* $(K,H)\,\mathsf{Chall}(\overline{A}) :: K \leq K'$ *for some* K'.

(c) $T = \mathsf{Top}$.

(d) $\overline{A} = \emptyset$.

(e) $\overline{B} = (\overline{B'}, n : U)$ *and* $\mathsf{Stale} \leq U$ *for some* $\overline{B'}, U$.

(f) $\mathcal{D}' \rhd (E \vdash n : (K,H)\,\mathsf{Chall}(), \overline{B})$ *and* $\mathsf{height}(\mathcal{D}') < \mathsf{height}(\mathcal{D})$ *for some* $\mathcal{D}'$.

Proof By induction on $\mathcal{D}$. ☐

The following lemma is a consequence of the fact that the subtyping rule (Subty Public Tainted) does not apply to SOSH nonce types.

Lemma B.21 (Possible SOSH Nonce Types) *If* $(E, \mathsf{fresh}(n : (K,H)\,\mathsf{Chall}(\overline{A})))$ *is nominal,* $\mathsf{Public} \notin K \cup H$ *and* $(E \vdash n : T, \overline{B})$, *then one of the following holds:*

(a) $T = (K,H)\,\mathsf{Chall}(\overline{C})$ *for some* $\overline{C}$.

(b) $T = (K,H)\,\mathsf{Resp}(\overline{C})$ *for some* $\overline{C}$.

(c) $T = \{\mathsf{Tainted}\}\,\mathsf{Top}$ *or* $T = \mathsf{Top}$.

Proof By a straightforward induction on $(E \vdash n : T, \overline{B})$'s derivation. For proof case (Sub), one uses that the only proper supertypes of SOSH nonce types are $\{\mathsf{Tainted}\}\,\mathsf{Top}$ and Top. ☐

Proof of Lemma 7.7 (Nonce Safety) *If* $(E \vdash \mathsf{fresh}(N : (K,H)\,\mathsf{Chall}(\overline{A})), N : (K,H)\,\mathsf{Resp}(\overline{B}), \overline{C})$ *and* E *is nominal, then* $(E \vdash N : \mathsf{Stale}, \overline{A}, \overline{B}, \overline{C})$.

Proof Fix K and H. Define $Ch(\overline{A}) \triangleq (K,H)\,\mathsf{Chall}(\overline{A})$ and $Rp(\overline{A}) \triangleq (K,H)\,\mathsf{Resp}(\overline{A})$ for all $\overline{A}$. We first prove the following auxiliary statements:

(a) *If* $(E, \mathsf{fresh}(n : Ch(\overline{A}, \overline{B})))$ *is nominal,* $\mathcal{D} \rhd (E \vdash n : T, \overline{C})$ *and* $T \leq Ch(\overline{A})$,
then $(E, n : \mathsf{Stale} \vdash \overline{B}, \overline{C})$.

(b) *If* $(E, \mathsf{fresh}(n : Ch(\overline{A})))$ *is nominal,* $\mathcal{D} \rhd (E \vdash n : T, \overline{C})$ *and* $T \leq Rp(\overline{B})$, *then* $(E, n : \mathsf{Stale} \vdash \overline{A}, \overline{B}, \overline{C})$.

We first prove auxiliary statement (a) and then (b). The proof of (b) uses statement (a).

Proof of auxiliary statement (a)*:* By induction on $\mathsf{height}(\mathcal{D})$. Suppose $(E, \mathsf{fresh}(n : Ch(\overline{A}, \overline{B})))$ is nominal, $\mathcal{D} \rhd (E \vdash n : T, \overline{C})$ and $T \leq Ch(\overline{A})$. Our proof is structured as follows: Before distinguishing proof cases by $\mathcal{D}$'s last rule, we consider three special cases in order avoid repeating the same argumentation for multiple last rules of $\mathcal{D}$. Here are the three special cases:

Suppose $\mathcal{D}' \rhd (E \vdash n : Ch(), \overline{C})$ and $\mathsf{height}(\mathcal{D}') < \mathsf{height}(\mathcal{D})$ for some $\mathcal{D}'$. Then, by induction hypothesis, $(E, n : \mathsf{Stale} \vdash \overline{A}, \overline{B}, \overline{C})$. Then $(E, n : \mathsf{Stale} \vdash \overline{B}, \overline{C})$, by weakening.

Suppose $\overline{C} = (\overline{C'}, n : U)$ and $\mathsf{Stale} \leq U$ for some U, $\overline{C'}$. Then, by weakening, $\mathcal{D}' \rhd (E \vdash n : T, \overline{C'})$ and $\mathsf{height}(\mathcal{D}') < \mathsf{height}(\mathcal{D})$ for some $\mathcal{D}'$. Then, by induction hypothesis, $(E, n : \mathsf{Stale} \vdash \overline{B}, \overline{C'})$. Then $(E, n : \mathsf{Stale} \vdash \overline{B}, \overline{C})$, by (Id) and (Sub).

Suppose $\overline{B} = \emptyset$. Then $(E, n : \mathsf{Stale} \vdash \overline{B}, \overline{C})$ is obtained from $(E \vdash n : T, \overline{C})$ by weakening.

(*) *In the remainder of the proof of statement* (a)*, suppose that* $\mathcal{D}' \rhd (E \vdash n : Ch(), \overline{C})$ *implies* $\mathsf{height}(\mathcal{D}') \geq \mathsf{height}(\mathcal{D})$*, and that* $\overline{C} = (\overline{C'}, n : U)$ *implies* $\mathsf{Stale} \not\leq U$*, and that* $\overline{B} \neq \emptyset$*.*

We now distinguish cases by $\mathcal{D}$'s possible last rules:

Suppose $\mathcal{D}$ ends in (Id) or (Lift) and E contains $(n : T)$. Then $Ch(\overline{A}, \overline{B}) \leq T \leq Ch(\overline{A})$, by type consistency of $(E, \mathsf{fresh}(n : Ch(\overline{A}, \overline{B})))$. Then $\overline{B} = \emptyset$, by Lemma B.5(c). This contradicts assumption (*).

Suppose $\mathcal{D}$ ends in (Copy), $\overline{C} = (n : T, \overline{C'})$ and $(E \vdash \overline{C})$ for some $\overline{C'}$. Note that E is n-stale, by type consistency of $(E, \mathsf{fresh}(n : Ch(\overline{A}, \overline{B})))$ and (Subty Stale Nonce). Therefore $\mathsf{Stale} \leq T$, by Lemma B.19(b). This contradicts assumption (*).

Suppose $\mathcal{D}$ ends in (Nonce Cast) or (Strengthen Resp) and $T = (K', H')\,\mathsf{Resp}(\overline{D})$ for some K', H', $\overline{D}$. But this is impossible, by Lemma B.20 and assumption (*).

Suppose $\mathcal{D}$ ends in (Weaken Chall), $T = (K', H')\,\mathsf{Chall}(\overline{D'})$ and $(E \vdash n : (K', H')\,\mathsf{Chall}(\overline{D'}, D), D, \overline{C})$ for some K', H', $\overline{D'}$, D. By Lemma B.20 and assumption (*), it is the case that $(K', H') = (K, H)$ and $(\overline{D'}, D)$ is a submultiset of $(\overline{A}, \overline{B})$. From $Ch(\overline{D'}) = T \leq Ch(\overline{A})$, it follows that $\overline{D'} = \overline{A}$, by Lemma B.5(c). From the fact that $(\overline{A}, D) = (\overline{D'}, D)$ is a submultiset of $(\overline{A}, \overline{B})$, it follows that $\overline{B} = (D, \overline{B'})$ for some $\overline{B'}$. By substituting $(K', H') = (K, H)$ and $\overline{D'} = \overline{A}$ into the rule premise $(E \vdash n : (K', H')\,\mathsf{Chall}(\overline{D'}, D), D, \overline{C})$, we obtain $(E \vdash n : Ch(\overline{A}, D), D, \overline{C})$. Moreover, $\mathsf{fresh}(n : Ch(\overline{A}, \overline{B})) = \mathsf{fresh}(n : Ch(\overline{A}, D, \overline{B'}))$, by $\overline{B} = (D, \overline{B'})$. Therefore, by induction hypothesis, $(E, n : \mathsf{Stale} \vdash \overline{B'}, D, \overline{C})$. Substituting $(\overline{B'}, D) = \overline{B}$ back into this judgment, we obtain $(E, n : \mathsf{Stale} \vdash \overline{B}, \overline{C})$.

Proof of auxiliary statement (b)*:* The proof is by induction on $(E \vdash n : T, \overline{C})$'s derivation height, using statement (a). This proof is similar to the proof of (a) and we omit it.

Proof of lemma: Suppose $(E \vdash \mathsf{fresh}(N : Ch(\overline{A})), N : Rp(\overline{B}), \overline{C})$ and E is nominal. Then $E = (E', \mathsf{fresh}(N : Ch(\overline{A})))$ and $(E' \vdash N : Rp(\overline{B}), \overline{C})$, by Lemma B.16. Moreover, N is a name, by atomicity of E. Then $(E', N : \mathsf{Stale} \vdash \overline{A}, \overline{B}, \overline{C})$, by auxiliary statement (b). Then $(E', N : \mathsf{Stale} \vdash N : \mathsf{Stale}, \overline{A}, \overline{B}, \overline{C})$, by (Id). Then $(E \vdash N : \mathsf{Stale}, \overline{A}, \overline{B}, \overline{C})$, by (Discard Chall). □

B.7 Cut

The following technical lemma helps to deal with pattern matching in the (Cut) proof.

Lemma B.22 (Expanding Pattern-Matching) *If E is nominal,* $(E \vdash M \in X, \overline{B})$, $\mathcal{D} \rhd (M \in X, \overline{B} \vdash \overline{C})$ *and* $X = \exists \vec{x}.N[\overline{A}]$*, then* $M = N\{\vec{x} \leftarrow \vec{L}\}$, $(E \vdash M : \mathsf{Top}, \overline{A}\{\vec{x} \leftarrow \vec{L}\}, \overline{B})$, $\mathcal{D}' \rhd (M : \mathsf{Top}, \overline{A}\{\vec{x} \leftarrow \vec{L}\}, \overline{B} \vdash \overline{C})$ *and* $\mathsf{height}(\mathcal{D}') \leq \mathsf{height}(\mathcal{D})$ *for some* $\vec{L}$, $\mathcal{D}'$.

Proof This follows from the definitions of right and left pattern-matching. □

We are now prepared to prove (Cut).

Proof of Theorem 7.2 (Cut) *If E is nominal, $(E \vdash !\overline{B}_0, \overline{B}_1, \overline{B}_2)$ and $(!\overline{B}_0, \overline{B}_1 \vdash \overline{C})$, then $(E \vdash !\overline{B}_0, \overline{B}_2, \overline{C})$.*

Proof We first prove the following auxiliary statement:

(a) *If E is nominal, $(E \vdash !\overline{B}_0, \overline{B}_1)$, $\mathcal{D} \rhd (!\overline{B}_0, \overline{B}_1 \vdash \overline{C})$ and no member of $\overline{B}_1$ is copyable, then $(E \vdash !\overline{B}_0, \overline{C})$.*

Proof of auxiliary statement (a): By induction on height($\mathcal{D}$). Suppose E is nominal, $(E \vdash !\overline{B}_0, \overline{B}_1)$, $\mathcal{D} \rhd (!\overline{B}_0, \overline{B}_1 \vdash \overline{C})$ and no member of $\overline{B}_1$ is copyable.

Suppose $\mathcal{D}$ ends in (Id). Then $\overline{C} = (!D, \overline{C}')$, $!\overline{B}_0 = (!\overline{B}'_0, !D)$ and $(!\overline{B}_0, \overline{B}_1 \vdash \overline{C}')$ for some $!D$, $\overline{C}'$, $\overline{B}'_0$. By induction hypothesis, $(E \vdash !\overline{B}_0, \overline{C}')$. Then $(E \vdash !\overline{B}_0, \overline{C})$, by (Copy).

Suppose $\mathcal{D}$ ends in (Lift). Then $\overline{C} = (D, \overline{C}')$, $(!\overline{B}_0, \overline{B}_1) = (\overline{B}', D)$ and $(\overline{B}' \vdash \overline{C}')$ for some D, $\overline{C}'$, $\overline{B}'$. Suppose first that $!D \neq D$. Then either $!D \in E$ or $E = (E', D)$ and $(E' \vdash \overline{B}')$ for some E', by Lemma B.17. Suppose first that $!D \in E$. From $(E \vdash !\overline{B}_0, \overline{B}_1)$ we obtain $(E \vdash \overline{B}')$, by weakening. From $(E \vdash \overline{B}')$ and $(\overline{B}' \vdash \overline{C}')$ we obtain $(E \vdash !\overline{B}_0, \overline{C}')$, by induction hypothesis. Because $!D \in E$, we obtain $(E \vdash !\overline{B}_0, \overline{C})$, by (Id). Suppose now that $E = (E', D)$ and $(E' \vdash \overline{B}')$ for some E'. By induction induction hypothesis, we obtain $(E' \vdash !\overline{B}_0, \overline{C}')$. Then $(E \vdash !\overline{B}_0, \overline{C})$, by (Lift). Suppose finally that $!D = D$. Then $!\overline{B}_0 = (!\overline{B}'_0, !D)$ for some $\overline{B}'_0$. By weakening, we obtain $(\overline{B} \vdash \overline{C}')$ from $(\overline{B}' \vdash \overline{C}')$. From $(E \vdash \overline{B})$ and $(\overline{B} \vdash \overline{C}')$, we obtain $(E \vdash !\overline{B}_0, \overline{C}')$, by induction hypothesis. Then $(E \vdash !\overline{B}_0, \overline{C})$, by (Copy).

Suppose $\mathcal{D}$ ends in (Unsub). Then $!\overline{B}_0 = (!\overline{B}'_0, M : T)$, $T \leq U$, $fv(U) = \emptyset$ and $(!\overline{B}'_0, M : U, \overline{B}_1 \vdash \overline{C})$ for some $!\overline{B}'_0$, M, T, U. From $(E \vdash !\overline{B}_0, \overline{B}_1)$ we obtain $(E \vdash !\overline{B}'_0, M : U, \overline{B}_1)$, by (Sub). From $(E \vdash !\overline{B}'_0, M : U, \overline{B}_1)$ and $(!\overline{B}'_0, M : U, \overline{B}_1 \vdash \overline{C})$, we obtain $(E \vdash !\overline{B}'_0, M : U, \overline{C})$, by induction hypothesis. Then $(E \vdash !\overline{B}_0, \overline{C})$, by (Sub).

Suppose $\mathcal{D}$ ends in (Pair). Then $!\overline{B}_0 = (!\overline{B}'_0, (M,N) : K\,\mathsf{Top})$ and $(!\overline{B}'_0, M : K\,\mathsf{Top}, N : K\,\mathsf{Top}, \overline{B}_1 \vdash \overline{C})$. From $(E \vdash !\overline{B}_0, \overline{B}_1)$ we obtain $(E \vdash !\overline{B}'_0, M : K\,\mathsf{Top}, N : K\,\mathsf{Top}, \overline{B}_1)$, by Lemma B.18(a). From this and $(!\overline{B}'_0, M : K\,\mathsf{Top}, N : K\,\mathsf{Top}, \overline{B}_1 \vdash \overline{C})$ we obtain $(E \vdash !\overline{B}'_0, M : K\,\mathsf{Top}, N : K\,\mathsf{Top}, \overline{C})$, by induction hypothesis. Then $(E \vdash !\overline{B}_0, \overline{C})$, by (Pair).

Suppose $\mathcal{D}$ ends in (Decrypt Trusted). Then $!\overline{B}_0 = (!\overline{B}'_0, \mathit{decrypt}(M_1, N) : J\,\mathsf{Auth}(M_2))$ and $(!\overline{B}'_0, \overline{B}_1 \vdash N : (K,H)\,\mathsf{DK}(X))$ and $(!\overline{B}'_0, \overline{B}_1, (M_1, M_2) \in X \vdash \overline{C})$ and $\mathsf{Tainted} \notin H \cup J$ for some $!\overline{B}'_0$, M_1, M_2, N, J, K, H, X. From $(E \vdash !\overline{B}_0, \overline{B}_1)$ it follows, by Lemma B.18(e), that $\mathit{decrypt}(M_1, N)$ is not a decryption pattern, i.e., $N = \mathsf{Dec}(N')$ and $\mathit{decrypt}(M_1, N) = \{\!|M_1|\!\}_{\mathsf{Enc}(N')}$ for some N'. From $(E \vdash !\overline{B}_0, \overline{B}_1)$ it follows, by Lemma B.18(f), that there exist K', H', X', M'_2 such that the following statements hold:

(1) $E \vdash N : (K', H')\,\mathsf{EK}(X')$, $(M_1, M'_2) \in X'$, $!\overline{B}'_0$, $\overline{B}_1$
(2) $(\mathsf{Tainted} \notin J) \;\Rightarrow\; (M'_2 = M_2)$
(3) $(\mathsf{Tainted} \in K') \;\Rightarrow\; \mathsf{Tainted} \in J$.

Because $\mathsf{Tainted} \notin H \cup J$, we obtain from (2) and (3):

(4) $M'_2 = M_2$
(5) $\mathsf{Tainted} \notin K'$

By induction hypothesis, we may cut $(E \vdash !\overline{B}'_0, \overline{B}_1)$ with $(!\overline{B}'_0, \overline{B}_1 \vdash N : (K,H)\,\mathsf{DK}(X))$, obtaining $(E \vdash N : (K,H)\,\mathsf{DK}(X))$. Then, by key uniqueness (Lemma 7.6), either $(K,H,X) = (K', H', X')$ or $\mathsf{Tainted} \in (K' \cap K^{-1}) \cup (H \cap H'^{-1}) \cup (K' \cap H)$. Because $\mathsf{Tainted} \notin H \cup J$ and $\mathsf{Tainted} \notin K'$, it follows that $(K,H,X) = (K', H', X')$. Now we can use Lemma B.22 and the induction hypothesis, in order to cut (1) with $(!\overline{B}'_0, \overline{B}_1, (M_1, M_2) \in X, N : (K', H')\,\mathsf{EK}(X') \vdash \overline{C})$, obtaining $(E \vdash N : (K', H')\,\mathsf{EK}(X'), (M_1, M'_2) \in X', !\overline{B}'_0)$. Then $(E \vdash !\overline{B}_0, \overline{C})$, by (Encrypt Trusted).

Suppose $\mathcal{D}$ ends in (Decrypt Untrusted). Then there exist $!\overline{B}'_0$, M_1, N, J, K, H, X, x such that $!\overline{B}_0 = (!\overline{B}'_0, \mathit{decrypt}(M_1, N) : J\,\mathsf{Top})$ and $(!\overline{B}'_0, \overline{B}_1 \vdash N : (K,H)\,\mathsf{DK}(X))$ and $\mathsf{Tainted} \in J$ and both the following hold:

(1) $(x, !\overline{B}'_0, \overline{B}_1, (M_1, x) \in X \vdash \overline{C}) \;\vee\; (\mathsf{Public}, \mathsf{Tainted} \in K \cup H^{-1})$
(2) $(!\overline{B}'_0, \overline{B}_1, M : J\,\mathsf{Top} \vdash \overline{C}) \vee (\mathsf{Public} \notin K \cup H^{-1})$

From $(E \vdash !\overline{B}_0, \overline{B}_1)$ it follows, by Lemma B.18(e), that *decrypt*(M_1,N) is not a decryption pattern, i.e., $N = \mathsf{Dec}(N')$ and *decrypt*$(M_1,N) = \{\!|M_1|\!\}_{\mathsf{Enc}(N')}$ for some N'. From $(E \vdash !\overline{B}_0, \overline{B}_1)$ it follows, by Lemma B.18(g), that there exist K', H', X', M_2 such that the following statements hold:

(3) $E \vdash N : (K',H')\,\mathsf{EK}(X'),\, (M_1,M_2) \in X',\, !\overline{B}'_0,\, \overline{B}_1$

(4) $(\mathsf{Tainted} \in K' \cup H'^{-1})$
$\Rightarrow\ (E \vdash N : (K',H')\,\mathsf{EK}(X'),\, (M_1,M_2) \in X',\, M_1 : (J \cup \{\mathsf{Tainted}\})\,\mathsf{Top},\, !\overline{B}'_0,\, \overline{B}_1)$

By induction hypothesis, we may cut $(E \vdash !\overline{B}'_0, \overline{B}_1)$ with $(!\overline{B}'_0, \overline{B}_1 \vdash N : (K,H)\,\mathsf{DK}(X))$, obtaining $(E \vdash N : (K,H)\,\mathsf{DK}(X))$. Then, by key uniqueness (Lemma 7.6), we get:

(5) $(K,H,X) = (K',H',X') \ \vee\ \mathsf{Tainted} \in (K' \cap K^{-1}) \cup (H \cap H'^{-1}) \cup (K' \cap H)$

We will now consider two cases, namely, $\{\mathsf{Public},\mathsf{Tainted}\} \subseteq K \cup H^{-1}$ and $\mathsf{Public} \notin K \cup H^{-1}$:

Suppose first that $\{\mathsf{Public},\mathsf{Tainted}\} \subseteq K \cup H^{-1}$. We claim that $\mathsf{Tainted} \in K' \cup H'^{-1}$. If $(K,H,X) = (K',H',X')$, then this is a consequence of $\{\mathsf{Public},\mathsf{Tainted}\} \subseteq K \cup H^{-1}$. Otherwise, by (5), $\mathsf{Tainted} \in (K' \cap K^{-1}) \cup (H \cap H'^{-1}) \cup (K' \cap H) \subseteq K' \cup H'^{-1}$. So, in any case, $\mathsf{Tainted} \in K' \cup H'^{-1}$. Then, by (4), it is the case that

$(E \vdash N : (K',H')\,\mathsf{EK}(X'),\, (M_1,M_2) \in X',\, M_1 : (J \cup \{\mathsf{Tainted}\})\,\mathsf{Top},\, !\overline{B}'_0,\, \overline{B}_1)$

Moreover, $(!\overline{B}'_0, \overline{B}_1, M : J\,\mathsf{Top} \vdash \overline{C})$, by (2). Because $\mathsf{Tainted} \in J$, it is the case that $(J \cup \{\mathsf{Tainted}\})\,\mathsf{Top} = J\,\mathsf{Top}$, and we can use Lemma B.22 and the induction hypothesis to cut these two judgments, obtaining $(E \vdash N : (K',H')\,\mathsf{EK}(X'),\, (M_1,M_2) \in X',\, M_1 : J\,\mathsf{Top},\, !\overline{B}'_0, \overline{C})$. Applying (Encrypt Untrusted) and (Sub) to this judgment results in $(E \vdash !\overline{B}_0, \overline{C})$.

Suppose now that $\mathsf{Public} \notin K \cup H^{-1}$. Then $(x,\, !\overline{B}'_0,\, \overline{B}_1,\, (M_1,x) \in X \vdash \overline{C})$, by (1). By substitutivity (Lemma 7.3), we get $(!\overline{B}'_0,\, \overline{B}_1,\, (M_1,M_2) \in X \vdash \overline{C})$. From $\mathsf{Public} \notin K \cup H^{-1}$ it follows that $\mathsf{Tainted} \notin (K' \cap K^{-1}) \cup (H \cap H'^{-1}) \cup (K' \cap H)$. Therefore, $(K,H,X) = (K',H',X')$, by (5). If $\mathsf{Tainted} \in K' \cup H'^{-1}$, we now cut (4) with $(!\overline{B}'_0,\, \overline{B}_1,\, (M_1,M_2) \in X \vdash \overline{C})$ followed by an application of (Encrypt Untrusted) to obtain the desired result. On the other hand, if $\mathsf{Tainted} \notin K' \cup H'^{-1}$, we cut (3) with $(!\overline{B}'_0,\, \overline{B}_1,\, (M_1,M_2) \in X \vdash \overline{C})$ followed by an application of (Encrypt Trusted) to obtain the desired result.

Suppose $\mathcal{D}$ ends in (Nonce Use). We have $!\overline{B}_0 = (!\overline{B}'_0,\, N : (K,H)\,\mathsf{Resp}(\overline{D}_2))$, $\overline{B}_1 = (\overline{B}'_1,\, \mathsf{fresh}(N : (K,H)\,\mathsf{Chall}(\overline{D}_1)))$ and $(!\overline{B}'_0,\, \overline{B}'_1,\, N : \mathsf{Stale},\, \overline{D}_1,\, \overline{D}_2 \vdash \overline{C})$ for some $!\overline{B}'_0$, $\overline{B}'_1$, K, H, $\overline{D}_1$, $\overline{D}_2$. From $(E \vdash !\overline{B}_0, \overline{B}_1)$ we obtain $(E \vdash !\overline{B}'_0,\, \overline{B}'_1,\, N : \mathsf{Stale},\, \overline{D}_1,\, \overline{D}_2)$, by nonce safety (Lemma 7.7). Then we get $(E \vdash !\overline{B}'_0,\, N : \mathsf{Stale},\, \overline{C})$, by induction hypothesis and weakening. From this we get $(E \vdash !\overline{B}_0, \overline{C})$, by (Subty Stale Nonce) and (Sub).

Suppose $\mathcal{D}$ end in (Discard Chall). We have $\overline{B}_1 = (\overline{B}'_1,\, \mathsf{fresh}(N : (K,H)\,\mathsf{Chall}(\overline{D})))$ and $(!\overline{B}_0,\, \overline{B}'_1,\, N : \mathsf{Stale} \vdash \overline{C})$ for some $\overline{B}'_1, N, K, H, \overline{D}$. Then $E = (E',\, \mathsf{fresh}(N : (K,H)\,\mathsf{Chall}(\overline{D})))$ and $(E' \vdash !\overline{B}_0, \overline{B}'_1)$, by Lemma B.16(b). Then $(E',\, N : \mathsf{Stale} \vdash !\overline{B}_0, \overline{B}'_1,\, N : \mathsf{Stale})$, by (Lift). Then $(E \vdash !\overline{B}_0, \overline{B}'_1,\, N : \mathsf{Stale})$, by (Discard Chall). From $(E \vdash !\overline{B}_0, \overline{B}'_1,\, N : \mathsf{Stale})$ and $(!\overline{B}_0,\, \overline{B}'_1,\, N : \mathsf{Stale} \vdash \overline{C})$ we obtain $(E \vdash !\overline{B}_0,\, N : \mathsf{Stale},\, \overline{C})$, by induction hypothesis. Then $(E \vdash !\overline{B}_0, \overline{C})$, by weakening.

All other proof cases are straightforward.

Proof of theorem: Suppose E is nominal, $(E \vdash !\overline{B}_0, \overline{B}_1, \overline{B}_2)$ and $(!\overline{B}_0, \overline{B}_1 \vdash \overline{C})$. For $i = 1,2$, decompose $\overline{B}_i$ into $\overline{B}_i = (!\overline{B}_{i1}, \overline{B}_{i2})$ such that no member of $\overline{B}_{i2}$ is copyable. By Lemma B.17, there exist $\overline{B}_{220}$, $\overline{B}_{221}$, E_0 such that $\overline{B}_{22} = (\overline{B}_{220}, \overline{B}_{221})$, $E = (E_0, \overline{B}_{221})$, $!D \in E_0$ for all D in $\overline{B}_{220}$, and $(E_0 \vdash !\overline{B}_0, \overline{B}_1, !\overline{B}_{21})$. From $(!\overline{B}_0, \overline{B}_1 \vdash \overline{C})$ we obtain $(!\overline{B}_0, \overline{B}_1, !\overline{B}_{21} \vdash \overline{C})$, by weakening. From $(E_0 \vdash !\overline{B}_0, \overline{B}_1, !\overline{B}_{21})$ and $(!\overline{B}_0, \overline{B}_1, !\overline{B}_{21} \vdash \overline{C})$, we obtain $(E_0 \vdash !\overline{B}_0, !\overline{B}_{11}, !\overline{B}_{21}, \overline{C})$, by auxiliary statement (a). Then $(E_0 \vdash !\overline{B}_0, !\overline{B}_{21}, \overline{C})$, by weakening. Then $(E \vdash !\overline{B}_0, \overline{B}_2, \overline{C})$, by (Id) and (Lift). □

B.8 Secrecy

For the following simple auxiliary lemma, recall Definition B.3 of *n*-ambiguous environments.

Lemma B.23 *If $\vec{n}$ are distinct, $(\vec{n} : \vec{T} \ ::: \ P) \rightarrow^* (\overline{A}, m : U \ ::: \ Q)$ and $(\overline{A}, m : U)$ is m-ambiguous, then U is a challenge type.*

Proof By inspection of the state transition rules, $\overline{A}$ does not contain a type assertion of the form $m : V$. Then, by Definition B.3 of m-ambiguity, $\overline{A}$ must contain an assertion of the form $\mathsf{fresh}(m : C)$ for some challenge type C. This is only possible if $C = U$, by inspection of the state transition rules. ☐

Recall Definition 4.1 of write-safety.

Lemma B.24 (Write-Safety) *If $\vec{n}$ are distinct, $\vec{T}$ are closed types and $(\vec{n} : \vec{T} \vdash P)$, then $(\vec{n} : \vec{T} \ ::: \ P)$ is write-safe.*

Proof Suppose $\vec{n}$ are distinct, $\vec{T}$ are closed types and $(\vec{n} : \vec{T} \vdash P)$. Suppose that $(\vec{n} : \vec{T} \ ::: \ P) \rightarrow^* (\overline{A}, m : U \ ::: \ \mathsf{out}\, L\ m \mid Q)$ and U is not a challenge type. Because $(\vec{n} : \vec{T} \vdash P)$, it is the case that $\vdash \vec{n} : \vec{T} \ ::: \ P$, by (State). Then, by type preservation, $\vdash \overline{A}, m : U \ ::: \ \mathsf{out}\, L\ m \mid Q$. By inverting the last rules of this typing judgment, we obtain that $(\overline{A}, m : U \vdash \overline{A'})$ and $(\overline{A'} \vdash m : \mathsf{Un})$ for some $\overline{A'}$. Cutting these two judgments, we get that $(\overline{A}, m : U \vdash m : \mathsf{Un})$. Then $U \leq \mathsf{Un}$, by Lemmas B.19(c) and B.23. Then U is public, because subtypes of Un are public. ☐

Proof of Theorem 4.1 (Robust Write-Safety) *If $\vec{n}$ are distinct names and $(\vec{n} : \vec{T} \vdash P)$, then P is robustly write-safe.*

Proof Same as proof of Theorem 3.1 in Section 7.2. ☐

B.9 Integrity

Recall Definition 4.3 of read-safety.

Lemma B.25 (Read-Safety) *If $\vec{n}$ are distinct, $\vec{T}$ are closed types and $(\vec{n} : \vec{T} \vdash P)$, then $(\vec{n} : \vec{T} \ ::: \ P)$ is read-safe.*

Proof Suppose $\vec{n}$ are distinct, $\vec{T}$ are closed types and $(\vec{n} : \vec{T} \vdash P)$. Suppose that $(\vec{n} : \vec{T} \ ::: \ P) \rightarrow^* (\overline{A}, m : \mathsf{Un} \ ::: \ \mathsf{out}\, L\ M\{x{\leftarrow}m, \vec{y}{\leftarrow}\vec{N}\} \mid (\mathsf{inp}\, L\ \exists x, \vec{y}\, .\, M[x : U, \overline{A}];\ P') \mid Q)$ and U is not a response type. Because $(\vec{n} : \vec{T} \vdash P)$, it is the case that $\vdash \vec{n} : \vec{T} \ ::: \ P$, by (State). Then, by type preservation, $\vdash \overline{A}, m : \mathsf{Un} \ ::: \ \mathsf{out}\, L\ M\{x{\leftarrow}m, \vec{y}{\leftarrow}\vec{N}\} \mid (\mathsf{inp}\, L\ \exists x, \vec{y}\, .\, M[x : U, \overline{A}];\ P') \mid Q$. By inverting the last rules of this typing judgment, we obtain that $(\overline{A}, m : \mathsf{Un} \vdash \overline{A'})$, $(!\overline{A'_0}, \overline{A'_1} \vdash M\{x{\leftarrow}m, \vec{y}{\leftarrow}\vec{N}\} : \mathsf{Un})$, and $(x, \vec{y};\ !\overline{A'_0}, \overline{A'_2}, M : \mathsf{Un} \vdash x : U)$ for some $\overline{A'}$, $!\overline{A'_0}, \overline{A'_1}, \overline{A'_2}$ such that $\overline{A'} = (!\overline{A'_0}, \overline{A'_1}, \overline{A'_2})$. By cutting the first two of these judgments, we obtain $(\overline{A}, m : \mathsf{Un} \vdash !\overline{A'_0}, !\overline{A'_2}, M\{x{\leftarrow}m, \vec{y}{\leftarrow}\vec{N}\} : \mathsf{Un})$. Applying the substitution $\{x{\leftarrow}m, \vec{y}{\leftarrow}\vec{N}\}$ to the third judgment, we obtain $(!\overline{A'_0}, \overline{A'_2}, M\{x{\leftarrow}m, \vec{y}{\leftarrow}\vec{N}\} : \mathsf{Un} \vdash m : U)$. Another cut results in $(\overline{A}, m : \mathsf{Un} \vdash m : U)$. Then $\mathsf{Un} \leq U$, by Lemmas B.19(c) and B.23. Then U is tainted, because supertypes of Un are tainted. ☐

Proof of Theorem 4.2 (Robust Read-Safety) *If $\vec{n}$ are distinct and $(\vec{n} : \mathsf{Un} \vdash P)$, then P is robustly read-safe.*

Proof Same as proof of Theorem 3.1 in Section 7.2. ☐

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

[2] M. Abadi. Security protocols and their properties. *20th Int. Summer School on Foundations of Secure Computation*, pages 39–60, 1999.

[3] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2001.

[4] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *29th ACM Symposium on Principles of Programming Languages*, pages 33–44, 2002.

[5] M. Abadi and L. de Alfaro, editors. *Proceedings of the International Conference on Concurrency Theory - CONCUR*, volume 3653 of *Lecture Notes in Computer Science*. Springer, 2005.

[6] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages*, pages 104–115, 2001.

[7] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.

[8] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[9] R. Amadio and W. Charatonik. On name generation and set-based analysis in the Dolev-Yao model. In *Proceedings of the International Conference on Concurrency Theory - CONCUR*, volume 2421 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2002.

[10] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[11] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the Computer Security Foundations Workshop - CSFW*, pages 82–96. IEEE Computer Society Press, 2001.

[12] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proceedings of the Computer Security Foundations Workshop - CSFW*, pages 126–140. IEEE Computer Society Press, 2003.

[13] D. Bolignano. An approach to the formal verification of cryptographic protocols. In *Third ACM Conference on Computer and Communications Security*, pages 106–118, 1996.

[14] M. Bugliesi, R. Focardi, and M. Maffei. Authenticity by tagging and typing. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 1–12. ACM Press, 2004.

[15] M. Burrows, M. Abadi, and R.M. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

[16] I. Cervesato. Typed MSR: Syntax and examples. In *First International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security*, volume 2052 of *Lecture Notes in Computer Science*, pages 159–177. Springer, 2001.

[17] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *CSFW*, pages 55–69, 1999.

[18] E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *Proceedings of the Computer Security Foundations Workshop - CSFW*, pages 144–158. IEEE Computer Society Press, 2000.

[19] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

[20] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols (FMSP99)*, 1999.

[21] J. Goguen. What is unification? In M. Nivat and H. Ait-Kaci, editors, *Resolution of Equations in Algebraic Structures*, volume 1, pages 217–261. Academic Press, 1989.

[22] A.D. Gordon, C. Haack, and A. Jeffrey. Cryptyc: Cryptographic protocol type checker. At *http://cryptyc.cs.depaul.edu/*, 2004.

[23] A.D. Gordon and A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Proc. Int. Software Security Symp.*, volume 2609 of *Lecture Notes in Computer Science*, pages 263–282. Springer-Verlag, 2002.

[24] A.D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *J. Computer Security*, 11(4):451–521, 2003.

[25] A.D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *J. Computer Security*, 12(3/4):435–484, 2003.

[26] A.D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In Abadi and de Alfaro [5], pages 186–201.

[27] C. Haack and A. Jeffrey. Timed spi-calculus with types for secrecy and authenticity. In Abadi and de Alfaro [5], pages 202–216.

[28] J. Heather. *'Oh! . . . Is it* really *you?' Using rank functions to verify authentication protocols*. PhD thesis, Royal Holloway, University of London, 2000.

[29] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings of the Computer Security Foundations Workshop - CSFW*, pages 255–268. IEEE Computer Society Press, 2000.

[30] J. Heather and S. Schneider. Towards automatic verification of authentication protocols on an unbounded network. In *Proceedings of the Computer Security Foundations Workshop - CSFW*, pages 132–143. IEEE Computer Society Press, 2000.

[31] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[32] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.

[33] Davide Marchignoli and Fabio Martinelli. Automatic verification of cryptographic protocols through compositional analysis techniques. In *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 1999.

[34] W. Marrero, E.M. Clarke, and S. Jha. Model checking for security protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. Preliminary version appears as Technical Report TR–CMU–CS–97–139, Carnegie Mellon University, May 1997.

[35] Fabio Martinelli. Analysis of security protocols as open systems. *Theor. Comput. Sci.*, 290(1):1057–1106, 2003.

[36] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[37] A.W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of the Computer Security Foundations Workshop - CSFW*, pages 98–107. IEEE Computer Society Press, 1995.

[38] S.A. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, 1998.

[39] F.J. Thayer Fábrega, J.C. Herzog, and J.D. Guttman. Strand spaces: Why is a security protocol correct? In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 160–171, 1998.

[40] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.