Union and Intersection Types for Secure Protocol Implementations

Michael Backes[1,2], Cătălin Hriţcu[1], Matteo Maffei[1]

[1]Saarland University [2]Max Planck Institute for Software Systems (MPI-SWS)

Abstract.
We present a new type system for verifying the security of cryptographic protocol implementations. The type system combines prior work on refinement types, with union, intersection, and polymorphic types, and with the novel ability to reason statically about the disjointness of types. The increased expressivity enables the analysis of important protocol classes that were previously out of scope for the type-based analyses of protocol implementations. In particular, our types can statically characterize: (i) more usages of asymmetric cryptography, such as signatures of private data and encryptions of authenticated data; (ii) authenticity and integrity properties achieved by showing knowledge of secret data; (iii) applications based on zero-knowledge proofs. The type system comes with a mechanized proof of correctness and an efficient type-checker.

1 Introduction

Modern applications are mostly distributed and they rely on complex cryptographic protocols to transmit data over potentially insecure networks (e.g., e-banking, e-commerce, social networks, and mobile applications). Protocol designers struggle to keep pace with the variety of possible security vulnerabilities, which have affected early authentication protocols like Needham-Schroeder [28, 39], carefully designed de facto standards like SSL and PKCS [19, 49], and even widely deployed products like Microsoft Passport [32] and Kerberos [21]. Even if the underlying cryptographic protocols are properly designed, security vulnerabilities may still arise due to flaws in the implementation. Manual security analyses of cryptographic protocols, and even more so protocol implementations, are extremely difficult and error-prone. Therefore, it is important to devise automated analysis techniques that can provide security guarantees for protocol implementations and, more generally, for the source code of distributed applications.

An effective approach for analyzing protocol implementations is to rely on software verification techniques, such as model checking and type theory, and to adapt them to the security problem. Type systems, in particular, proved successful in the automated analysis of both cryptographic protocol models [1, 2, 33] and protocol implementations [14, 16]. Type systems provide security proofs for an unbounded number of runs. Furthermore, the analysis is modular and has a predictable termination behavior. Finally, type systems were designed from the beginning to efficiently deal with programming language features such as data structures, recursion, state, and higher-order functions: consequently, type systems are more efficient and scale better than many state-of-the-art protocol verifiers (e.g., ProVerif [18] used as a back end by fs2pv [17]) in the analysis of source code [16].

Despite these promising features, the type-based analysis of the source code of modern distributed applications is still an open issue. The first problem is that many of

these applications (e.g., trusted computing [20], electronic voting [24], and social networks [11]) rely on complex cryptographic schemes, such as zero-knowledge proofs. Although the automated verification of protocols based on some of these schemes is possible in process calculi for abstract protocol specifications, which provide convenient mechanisms to symbolically abstract these schemes (e.g., flexible equational theories), this is not the case for standard programming languages, where one needs to encode these abstractions using the primitives provided by the language. These primitives were, however, not designed for abstractly representing cryptographic primitives, which makes providing encodings that are suitable for automatic analysis and capture all potential usages of cryptographic schemes a challenging task. The second, somewhat similar, problem is that some interesting security properties are obtained by specific cryptographic patterns that are difficult to encode in type systems for programming languages. For instance, authenticity and integrity properties can be achieved by showing the knowledge of secret data, as in the Needham-Schroeder-Lowe public-key protocol [39] that relies on the exchange of secret nonces to authenticate the participants or as in most authentication protocols based on zero-knowledge proofs (e.g., Direct Anonymous Attestation [20] and Civitas [24]).

1.1 Contributions

This paper presents a new type system for the verification of the source code of protocol implementations. The underlying type theory combines refinement types [14] with union, intersection, and polymorphic types. Additionally, we introduce a novel relation for statically reasoning about the disjointness of types. This expressive type system extends the scope of existing type-based analyses of protocol implementations [14, 16] to important protocol classes that were not covered so far. In particular, our types statically characterize: (i) more usages of asymmetric cryptography, such as signatures of private data and encryptions of authenticated data; (ii) authenticity and integrity properties achieved by showing knowledge of secret data; (iii) applications based on zero-knowledge proofs.

Protocols are implemented in RCF$^{\forall}_{\wedge\vee}$ [14], a concurrent lambda-calculus, and cryptographic primitives are considered fully reliable building blocks and represented symbolically using a sealing mechanism [14, 41, 47]. In addition to hashes, symmetric cryptography, public-key encryption, and digital signatures, our approach supports zero-knowledge proofs. Since the realization of zero-knowledge proofs changes according to the statement to be proven, we provide a tool that, given a statement, automatically generates a symbolic implementation of the corresponding zero-knowledge primitive.

We have formalized RCF$^{\forall}_{\wedge\vee}$, the type system, and all the important parts of the soundness proof in the Coq proof assistant. We achieve this by defining a core calculus, which we call Formal-RCF$^{\forall}_{\wedge\vee}$, and which is obtained from RCF$^{\forall}_{\wedge\vee}$ by type erasure and by adopting a locally nameless representation for binders [6]. We believe this formalization is important, since the powerful combination of refinement, union, and intersection types makes the proof of soundness non-trivial, tedious, and potentially error-prone. Indeed, this work allowed us to discover three relatively small problems in the soundness proofs of prior type systems with refinement types [10, 14] and to propose and evaluate fixes for the faulty proofs.

Our type-based analysis is automated, modular, efficient, and provides security proofs for an unbounded number of sessions. We have implemented a type-checker that performed very well in our experiments: it type-checks all our symbolic libraries and samples totaling more than 1500LOC in around 12 seconds, on a normal laptop. The type-checker features a user-friendly graphical interface for examining typing derivations. The

tool-chain we have developed additionally contains an automatic code generator for zero-knowledge proofs, an interpreter, and a visual debugger. The formalization and the implementation are available online [9].

1.2 Related Work

Our type system extends the refinement type system by Bengtson et al. [14] with union, intersection, and polymorphic types. We also encode a novel type Private, which is used to characterize data that are not known to the attacker. A crucial property is that the set of values of type Private is disjoint from the set of values of type Un, which is the type of the messages known to the attacker. This property allows us to prune typing derivations following equality tests between values of type Private and values of type Un. This technique was first proposed by Abadi and Blanchet in their seminal work on secrecy types for asymmetric cryptography [1], but later disappeared in the more advanced type systems for authorization policies. Our extension is necessary to deal with protocols based on zero-knowledge proofs and to verify integrity and authenticity properties obtained by showing knowledge of secret data (e.g., the Needham-Schroeder-Lowe public-key protocol). In addition, our extension removes the restrictions that the type system proposed in [14] poses on the usage of standard cryptographic primitives. For instance, if a key is used to sign a secret message, then the corresponding verification key cannot be made public. These limitations were preventing the analysis of many interesting cryptographic applications, such as the Direct Anonymous Attestation protocol [20], which involves digital signatures on secret TPM identifiers.

In recent parallel work, Bhargavan et al. [16] have developed an additional cryptographic library for a simplified version of the type system proposed in [14]. This library does not rely on sealing but on data type constructors and inductive logical invariants that allow for reasoning about symmetric and asymmetric cryptography, hybrid encryption, and different forms of nested cryptography. The aforementioned logical invariants are, however, fairly complex and have to be proven manually. Moreover, these logical invariants are global, which means that adding new cryptographic primitives could require reproving the previous established invariants. Therefore, extending a symbolic cryptographic library in the style of [16] to new primitives requires expertise and a considerable human effort. In contrast, extending our sealing-based library does not involve any additional proof: one has just to find a well-typed encoding of the desired cryptographic primitive, which is relatively easy.[1]

The main simplification Bhargavan et al. [16] propose over [14] is the removal of the kinding relation, which classifies types as public or tainted, and allows values of public types to also be given any tainted type by subsumption. While this simplification removes the last security-specific part of the type system, therefore making it more standard, this change also requires attackers to be well-typed with respect to a carefully constructed attacker interface. In contrast, by retaining the kinding relation from [14] we also retain the property that *all* attackers are well-typed with respect to our type system (this property is usually called opponent typability). Despite these disadvantages, Bhargavan et al. [16] manage to solve some of the problems we address in this paper, without relying on union and intersection types, but instead using the logical connectives inside the

[1] A master's student encoded the sophisticated cryptographic schemes used in the Civitas [24] electronic voting protocol (i.e., distributed decryption, plaintext equivalence tests, homomorphic encryptions, mix nets, and a variety of zero-knowledge proofs) in about three weeks [31].

refinement types. It would be interesting future work to try to combine the advantages of both approaches in a unified framework.

Backes et al. [13] have recently established a semantic correspondence for asymmetric cryptography between a library based on sealing and one based on constructors, showing that both libraries enjoy computational soundness guarantees.

Backes et al. [10] proposed a type system for statically analyzing security protocols based on zero-knowledge proofs in the setting of the Spi calculus. Zero-knowledge proofs are modeled using constructors and destructors. In an extension of this type system [8], union and intersection types are used to infer precise type information about the secret witnesses of zero-knowledge proofs. This is captured in a separate relation called statement verification, which is fairly complex and tailored to zero-knowledge proofs. In contrast, in our paper we encode zero-knowledge proofs symbolically using standard programming language primitives, and we type-check them using general typing rules.

Goubault-Larrecq and Parrennes developed a static analysis technique [35] based on pointer analysis and clause resolution for cryptographic protocols implemented in C. The analysis is limited to secrecy properties, it deals only with standard cryptographic primitives, and it does not offer scalability since the number of generated clauses is very high even on small protocol examples.

Chaki and Datta have proposed a technique [23] based on software model checking for the automated verification of protocols implemented in C. The analysis provides security guarantees for a bounded number of sessions and is effective in discovering attacks. It was used to check secrecy and authentication properties of the SSL handshake protocol for configurations of up to three servers and three clients. The analysis only deals with standard cryptographic primitives, and offers only limited scalability.

Bhargavan et al. proposed a technique [17] for the verification of F# protocol implementations by automatically extracting ProVerif models [18]. The technique was successfully used to verify implementations of real-world cryptographic protocols such as TLS [15]. The analysis, however, is not compositional and is significantly less scalable than type-checking [16]. Furthermore, the considered fragment of F# is restrictive: it does not include higher-order functions, and it allows only for a very limited usage of recursion and state.

The more technical discussion about the related work on union and intersection types is postponed to §9.

1.3 Outline

The remainder of the paper is structured as follows. §2 gives an intuitive overview of our type system and exemplifies the most important concepts on a simple authentication protocol. §3 introduces the syntax of $\mathrm{RCF}^{\forall}_{\wedge\vee}$, the language supported by our type-checker. §4 presents the type system. §5 describes the results of our Coq formalization. §6 and §7 show how our type system can be used to obtain an expressive characterization of asymmetric cryptography and zero-knowledge proofs, respectively. §8 describes our toolchain. §9 discusses some related work on union and intersection types. §10 concludes and gives some interesting research directions.

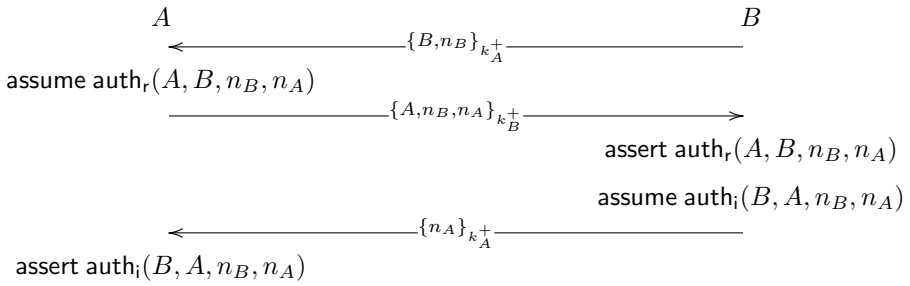
2 Our Type System at Work

Before giving the details of the calculus and the type system, we illustrate the main concepts of our static analysis technique on the Needham-Schroeder-Lowe public-key proto-

col [39] (NSL), which could not be analyzed with previous refinement type systems for protocol implementations [14, 16]. For convenience, throughout this section we use some syntactic sugar that is supported by our type-checker and can be obtained from the core calculus presented in §3 by standard encodings [14].

2.1 Protocol Description and Security Annotations

The Needham-Schroeder-Lowe protocol is depicted below:



The goal of this protocol is to allow A and B to authenticate with each other and to exchange two fresh nonces, which are meant to be private and be later used to construct a session key. B creates a fresh nonce n_B and encrypts it together with his own identifier with A's public key. A decrypts the ciphertext with her private key. At this point of the of the protocol, A does not know whether the ciphertext comes from B or from the opponent as the encryption key used to create the ciphertext is public. A continues the protocol by creating a fresh nonce n_A, and encrypts this nonce together with n_B and her own identifier with B's public key. B decrypts the ciphertext and, although the encryption key used to create the ciphertext is public, if the nonce he received matches the one he has sent to A then B does indeed know that the ciphertext comes from A, since the nonce n_B is *private* and only A has access to it. Finally, B encrypts the nonce n_A received from A with A's public key, and sends it back to A. After decrypting the ciphertext and checking the nonce, A knows that the ciphertext comes from B as the nonce n_A is *private* and only B has access to it.

Following [14], we decorate the code with assumptions and assertions. Intuitively, assumptions introduce new hypotheses, while assertions declare formulas that should logically follow from the previously introduced hypotheses. A program is safe if in all program runs the assertions are entailed by the assumptions. The assumptions and assertions of the NSL protocol capture the standard mutual authentication property.

2.2 Types for Cryptography

Before illustrating how we can type-check this protocol, let us introduce the typed interface of our library for public-key cryptography. Intuitively, since encryption keys are public, they can be used by honest principals to encrypt data as specified by the protocol, or by the attacker to encrypt arbitrary data. This intuitive reasoning is captured by the following typed interface:

$$\begin{array}{l} \mathit{encrypt} : \forall\alpha.\ \mathsf{PubKey}\langle\alpha\rangle \to \alpha \vee \mathsf{Un} \to \mathsf{Un} \\ \mathit{decrypt} : \forall\alpha.\ \mathsf{Un} \to \mathsf{PrivKey}\langle\alpha\rangle \to \alpha \vee \mathsf{Un} \end{array}$$

Like many of the functions in our cryptographic library, the *encrypt* and *decrypt* functions are polymorphic. Their code is type-checked only once and given an universal type. The type variable α stands in this case for the type of the payload that is encrypted, and can be instantiated with an arbitrary type when the functions are used.

Type Un describes those values that may be known to the opponent, i.e., data that may come from or be sent to the opponent. The type $\mathsf{PubKey}\langle\alpha\rangle$ describes public keys. Since the opponent has access to the public key and to the encryption function, the type system has to take into account that the library may be used by honest principals to encrypt data of type α or by the opponent to encrypt data of type Un. The *encrypt* function takes as input a public key of type $\mathsf{PubKey}\langle\alpha\rangle$ a message of type $\alpha \vee \mathsf{Un}$, and returns a ciphertext of type Un. The *decrypt* function takes as input a ciphertext of type Un, a private key of type $\mathsf{PrivKey}\langle\alpha\rangle$ and returns a payload of type $\alpha \vee \mathsf{Un}$. Without union types, the type of the payload is constrained to be Un or a supertype thereof [14], which severely limits the expressiveness of the type system and prevents the analysis of a number of protocols, including this very simple example.

2.3 Type-checking the NSL Protocol

We first introduce the type definitions[2] for the content of the three ciphertexts:

$$
\begin{array}{ll}
\mathsf{msg1} & = (\mathsf{Un} * \mathsf{Private}) \\
\mathsf{msg2}[x_B] & = (x_A : \mathsf{Un} * x_{nB} : \mathsf{Private} \vee \mathsf{Un} \ * \ \{x_{nA} : \mathsf{Private} \mid \mathsf{auth_r}(x_A, x_B, x_{nB}, x_{nA})\}) \\
\mathsf{msg3} & = \{x_{nA} : \mathsf{Private} \mid \exists x_A, x_B, x_{nB}. \\
 & \qquad\qquad \mathsf{auth_r}(x_A, x_B, x_{nB}, x_{nA}) \ \wedge \ \mathsf{auth_i}(x_B, x_A, x_{nB}, x_{nA})\}
\end{array}
$$

The first ciphertext contains a pair composed of a public identifier of type Un and a nonce of type $\mathsf{Private}$. Type $\mathsf{Private}$ describes values that are not known to the attacker: the set of values of type Un is disjoint from the set of values of type $\mathsf{Private}$. Type $\mathsf{msg2}[x_A]$ is a combination of two dependent pair types and one refinement type. This type describes a triple composed of an identifier x_A of type Un, a first nonce x_{nB} of type $\mathsf{Private} \vee \mathsf{Un}$, and a second nonce x_{nA} of type $\mathsf{Private}$ such that the predicate $\mathsf{auth_r}(x_A, x_B, x_{nB}, x_{nA})$ is entailed by the assumptions in the system (A assumes $\mathsf{auth_r}(A, B, n_B, n_A)$ before creating the second ciphertext). The free occurrence of x_B is bound in the type definition. Notice that x_{nB} is given type $\mathsf{Private} \vee \mathsf{Un}$ since A does not know whether the nonce received in the first ciphertext comes from B or from the opponent. Type $\mathsf{msg3}$ is a refinement type describing a nonce x_{nA} of type $\mathsf{Private}$ such that the formula $\exists x_A, x_B, x_{nB}.\ \mathsf{auth_r}(x_A, x_B, x_{nB}, x_{nA}) \wedge \mathsf{auth_i}(x_B, x_A, x_{nB}, x_{nA})$ is entailed by the assumptions in the system. Indeed, before creating the third ciphertext, B has asserted $\mathsf{auth_r}(A, B, n_B, n_A)$ and assumed $\mathsf{auth_i}(B, A, n_B, n_A)$. Since the payload of the third message only contains x_{nA} we existentially quantify the other variables. The overall type of the payload is obtained by combining the three previous types:

$$\mathsf{payload}[x] = \mathsf{Msg1\ of\ msg1} \mid \mathsf{Msg2\ of\ msg2}[x] \mid \mathsf{Msg3\ of\ msg3}$$

The type of A's public key is defined as $\mathsf{PubKey}\langle\mathsf{payload}[A]\rangle$ and the type of B's public key is defined as $\mathsf{PubKey}\langle\mathsf{payload}[B]\rangle$.

The code of the initiator (B in our diagram) and the code of the responder (A) abstract over the principal's identity and they are type-checked independently of each other.

[2] Type definitions are syntactic sugar, and are inlined by the type-checker.

Table 1 NSL Initiator Code and Responder Code

```
init = λx_B : Un. λx_A : Un.
       λk_B : PrivKey⟨payload[x_B]⟩.
       λpk_A : PubKey⟨payload[x_A]⟩.
       λch : Ch(Un).
  let n_B = mkPriv() in
  let p_1 = (Msg1 (x_B, n_B)) in
  let m_1 = encrypt⟨payload[x_A]⟩ pk_A p_1 in
  send⟨Un⟩ ch m_1;
  let z = recv⟨Un⟩ ch in
  let x = decrypt⟨payload[x_B]⟩ k_B z in
  case x_1 = x : payload[x_B] ∨ Un in
  match x_1 with Msg2 x_2 ⇒
  let (y_A, y_nB, y_nA) = x_2 in
  if y_A = x_A then
  if y_nB = n_B then
  assert auth_r(x_A, x_B, y_nB, y_nA);
  assume auth_i(x_B, x_A, y_nB, y_nA);
  let p_3 = (Msg3 y_nA) in
  let m_3 = encrypt⟨payload[x_A]⟩ pk_A p_3 in
  send⟨Un⟩ ch m_3
```

```
resp = λx_A : Un. λx_B : Un.
       λpk_B : PubKey⟨payload[x_B]⟩.
       λk_A : PrivKey⟨payload[x_A]⟩.
       λch : Ch(Un).
  let m_1 = recv⟨Un⟩ ch in
  let x_1 in decrypt⟨payload[x_A]⟩ k_A m_1
  case y_1 = x_1 : payload[x_A] ∨ Un in
  match y_1 with Msg1 z_1 ⇒
  let (y_B, x_nB) = z1 in
  if y_B = x_B then
  let n_A = mkPriv() in
  assume auth_r(x_A, x_B, x_nB, n_A);
  let p_2 = Msg2(x_A, x_nB, n_A) in
  let m_2 = encrypt⟨payload[x_B]⟩ pk_B p_2 in
  send⟨Un⟩ ch m_2;
  let m_3 = recv⟨Un⟩ch in
  let x_3 = decrypt⟨payload[x_A]⟩ k_A m_3 in
  case y_3 = x_3 : payload[x_A] ∨ Un in
  match y_3 with Msg3 y_nA ⇒
  if y_nA = n_A then
  assert auth_i(x_B, x_A, x_nB, n_A)
```

Since library functions such as *encrypt*, *decrypt*, *send* and so on are polymorphic, they are instantiated with a concrete types in the code (e.g., the encryptions in the initiator's code are instantiated with type $\mathsf{payload}[x_A]$ since they take as argument x_A's public key). The initiator creates a fresh private nonce by means of the function *mkPriv*. The nonce is encrypted together with B's identifier and sent on the network. The message x obtained by decrypting the second ciphertext is given type $\mathsf{payload}[x_B] \vee \mathsf{Un}$, which reflects the fact that B does not know whether the first ciphertext comes from A or from the attacker. Since we cannot statically predict which of the two types is the right one, we have to type-check the continuation code twice, once under the assumption that x has type $\mathsf{payload}[x_B]$ and once assuming that x has type Un. This is realized by the expression case $x_1 = x$: $\mathsf{payload}[x_B] \vee \mathsf{Un}$ in ....

If x has type $\mathsf{payload}[x_B]$, then its components are given strong types: y_A is given type Un, y_{nB} is given type $\mathsf{Private} \vee \mathsf{Un}$, and y_{nA} is given the refinement type $\{y_{nA} : \mathsf{Private} \mid \mathsf{auth_r}(x_A, x_B, y_{nB}, y_{nA})\}$. This refinement type ensures that $\mathsf{auth_r}(x_A, x_B, y_{nB}, y_{nA})$ will be entailed at run-time by the assumptions in the system and thus justifies the assertion assert $\mathsf{auth_r}(x_A, x_B, y_{nB}, y_{nA})$. Finally, the assumption assume $\mathsf{auth_i}(x_B, x_A, y_{nB}, y_{nA})$ allows us to give y_{nA} type msg3 and thus to type-check the final encryption.

If x has type Un then y_A, y_{nB}, and y_{nA} are also given type Un. The following equality check between the value y_{nB} of type Un and the nonce n_B of type Private makes type-checking the remaining code superfluous: since the set of values of type Un is disjoint from the set of values of type Private, it cannot be that the equality test succeeds. So type-checking the initiator's code succeeds.

Type-checking the responder's code is similar. The code contains two case expressions to deal with the union types introduced by the two decryptions. In particular, the code after the second decryption has to be type-checked under the assumption that the variable y_{nA} has type msg3 and under the assumption that y_{nA} has type Un.

In the former case, the assertion assert $\mathsf{auth_i}(x_B, x_A, x_{nB}, n_A)$ is justified by the previously assumed formula $\mathsf{auth_r}(x_A, x_B, x_{nB}, n_A)$, the formula in the above refinement type, and the following global assumption, stating that there cannot be two different assumptions $\mathsf{auth_r}(x_A, x_B, x'_{nB}, x'_{nA})$ and $\mathsf{auth_r}(x'_A, x'_B, x'_{nB}, x'_{nA})$ with the same nonce x_{nB}.

$$\mathsf{assume}\ \forall x_A, x_B, x'_A, x'_B, x_{nA}, x'_{nA}, x_{nB}.\ \mathsf{auth_r}(x_A, x_B, x_{nB}, x_{nA}) \wedge \mathsf{auth_r}(x'_A, x'_B, x_{nB}, x'_{nA}) \Rightarrow x_A = x'_A \wedge x_B = x'_B \wedge x_{nA} = x'_{nA}$$

This assumption is justified by the fact that the predicate $\mathsf{auth_r}$ is assumed only in the responder's code, immediately after the creation of a fresh nonce x_{nB}.

If y_{nA} is given type Un then type-checking the following code succeeds because the equality check between y_{nA} and the value n_A of type Private cannot succeed.

The functions init and resp take private keys as input, so they are not available to the attacker. We provide two public functions that capture the capabilities of the attacker.

Attacker's Interface for NSL

```
createPrincipal = λx : Un.
  let k = mkPrivKey⟨payload[x]⟩ () in addToDB x k; getPubKey⟨payload[x]⟩ k

startNSL = λ(role : Un)(x_A : Un)(x_B : Un)(c : Un).
  let k_A = getFromDB x_A in let pk_A = getPubKey⟨payload[x_A]⟩ k_A in
  let k_B = getFromDB x_B in let pk_B = getPubKey⟨payload[x_B]⟩ k_B in
  match role with inl _ ⇒ (init x_A x_B k_A pk_B c)
  | inr _ ⇒ (resp x_B x_A pk_A k_B c)
```

We allow the attacker to create arbitrarily many new principals using the createPrincipal function. This generates a new encryption key-pair, stores it in a private database, and then returns the corresponding public key to the attacker. The second function, startNSL, allows the attacker to start an arbitrary number of sessions of the protocol, between principals of his choice. When calling startNSL, the attacker chooses whether he wants to start an initiator or a responder, the principals to be involved in the session, and the channel on which the communication occurs. One principal can be involved in many sessions simultaneously, in which it may play different roles.

The two functions above express the capabilities of the attacker for verification purposes, and would not be exposed in a production setting. However, they can also be useful for testing and debugging the code of the protocol: for instance we can execute a protocol run using the following code.

Test Setup for NSL

```
createPrincipal "Alice"; createPrincipal "Bob";
let c = mkChan⟨Un⟩ () in
(startNSL (inl ()) "Alice" "Bob" c) ↱ (startNSL (inr ()) "Alice" "Bob" c)
```

Since the code of the NSL protocol is well-typed, the soundness result of the type system ensures that in all program runs the assertions are entailed by the assumptions, i.e., the code is safe when executed by an arbitrary attacker. In addition, the two nonces are given type Private and thus they are not revealed to the opponent.

3 The RCF$^{\forall}_{\wedge\vee}$ Calculus

The Refined Concurrent FPC (RCF) [14] is a simple programming language extending the Fixed Point Calculus with refinement types and concurrency [4]. This core calculus is expressive enough to encode a considerable fragment of an ML-like programming language [14]. In this paper, we further increase the expressivity of the calculus by adding intersection types [43], union types [42], and parametric polymorphism. We call the extended calculus RCF$^{\forall}_{\wedge\vee}$ and describe it in this and the following section.

We start by presenting the surface syntax of RCF$^{\forall}_{\wedge\vee}$, which is a subset of the syntax supported by our type-checker. In the surface syntax of RCF$^{\forall}_{\wedge\vee}$ variables are named, which makes programs human-readable. The surface syntax also contains explicit typing annotations that guide type-checking. It is given semantics by translation (i.e., type erasure) into a core implicitly-typed calculus, Formal-RCF$^{\forall}_{\wedge\vee}$, which we have formalized in Coq (see §5). The syntax comprises the four mutually-inductively-defined sets of values, types, expressions, and formulas. We mark with star (*) the constructs that are completely new with respect to RCF [14].

Surface syntax of RCF$^{\forall}_{\wedge\vee}$ values

x, y, z	variable
$h ::= \mathsf{inl} \mid \mathsf{inr}$	constructor for sum types
$M, N ::=$	value
x	variable
$()$	unit
$\lambda x : T.\, A$	function (scope of x is A)
(M, N)	pair
$h\ M$	value of sum type
$\mathsf{fold}_{\mu\alpha.\, T}\ M$	recursive value
$\Lambda\alpha.\, A$	type abstraction* (scope of α is A)
$\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}.\ M$	value of intersection type* (scope of $\widetilde{\alpha} = \alpha_1, \ldots, \alpha_n$ is M)

The set of *values* is composed of variables, the unit value, functions, pairs, and introduction forms for disjoint union, recursive, polymorphic, and intersection types.

Surface syntax of RCF$^{\forall}_{\wedge\vee}$ types

α, β	type variable
$T, U, V ::=$	type
unit	unit type
$x : T \to U$	dependent function type (x bound in U)
$x : T * U$	dependent pair type (x bound in U)
$T + U$	disjoint sum type
$\mu\alpha.\, T$	iso-recursive type (α bound in T)
α	type variable
$\{x : T \mid C\}$	refinement type (x bound in C)
$T \wedge U$	intersection type*
$T \vee U$	union type*
$\top$	top type*
$\forall\alpha.\, T$	polymorphic type* (α bound in T)

The unit value $()$ is given type unit. Functions $\lambda x : T.\, A$ taking as input values of type T and returning values of type U are given the dependent type $x : T \to U$, where the result type U can depend on the input value x. Pairs are given dependent types of the

form $x : T * U$, where the type U of the second component of the pair can depend on the value x of the first component. If U does not depend on x, then we use the abbreviations $T \to U$ and $T * U$. The sum type $T + U$ describes values $\mathsf{inl}(M)$ where M is of type T and values $\mathsf{inr}(N)$ where N is of type U. The iso-recursive type $\mu\alpha.\, T$ is the type of all values $\mathsf{fold}_{\mu\alpha.\, T}\ M$ where M is of type $T\{\mu\alpha.\, T/\alpha\}$. We use refinement types [14] to associate logical formulas to messages. The refinement type $\{x : T \mid C\}$ describes values M of type T for which the formula $C\{M/x\}$ is entailed by the current typing environment. A value is given the intersection type $T \wedge U$ if it has both type T and type U. A value is given a union type $T \vee U$ if it has type T or if it has type U, but we do not necessarily know what its precise type is. The top type $\top$ is supertype of all the other types, and contains all well-typed values. The universal type $\forall\alpha.\, T$ describes polymorphic values $\Lambda\alpha.\, A$ such that $A\{U/\alpha\}$ is of type $T\{U/\alpha\}$ for all types U.

Surface syntax of RCF$^{\forall}_{\wedge\vee}$ expressions

a, b	name
$A, B ::=$	expression
M	value
$M\ N$	function application
$M\langle T\rangle$	type instantiation*
$\mathsf{let}\ x = A\ \mathsf{in}\ B$	let (scope of x is B)
$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ A$	pair split (scope of x, y is A)
$\mathsf{match}\ M\ \mathsf{with}\ \mathsf{inl}\ x \Rightarrow A \mid \mathsf{inr}\ y \Rightarrow B$	pattern matching (scope of x is A, of y is B)
$\mathsf{unfold}_{\mu\alpha.\, T}\ M$	use recursive value
$\mathsf{case}\ x = M : T \vee U\ \mathsf{in}\ A$	elimination of union types* (scope of x is A)
$\mathsf{if}\ M = N\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$	equality check with type cast* (scope of x is A)
$(\nu a \updownarrow T)A$	restriction (scope of a is A)
$A \Rsh B$	fork off parallel expression
$a!M$	send M on channel a
$a?$	receive on channel a
$\mathsf{assume}\ C$	add formula C to global log
$\mathsf{assert}\ C$	formula C must hold

The syntax of expressions is mostly standard [14, 42]. A type instantiation $M\langle T\rangle$ specializes a polymorphic value M with the concrete type T. The elimination form for union types $\mathsf{case}\ x = M : T \vee U\ \mathsf{in}\ A$ substitutes the value M in A. The conditional $\mathsf{if}\ M = N\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$ checks if M is syntactically equal to N, if this is the case it substitutes x with the common value. Syntactic equality is defined up to alpha-renaming of binders and the erasure of typing annotations and constructs such as for (see §5). During type-checking the variable x is given the intersection of the types of M and N. When the variable x is not necessary we omit the as clause, as we did in §2. The restriction $(\nu a \updownarrow T)A$ generates a globally fresh channel a that can only be used in A to convey values of type T. The expression $A \Rsh B$ evaluates A and B in parallel, and returns the result of B (the result of A is discarded). The expression $a!M$ outputs M on channel a and returns the unit value $()$. Expression $a?$ blocks until some message M is available on channel a, removes M from the channel, and then returns M. Expression $\mathsf{assume}\ C$ adds the logical formula C to a global log. The assertion $\mathsf{assert}\ C$ returns $()$ when triggered. If at this point C is entailed by the multiset S of formulas in the global log, written as $S \models C$, we say the assertion *succeeds*; otherwise, we say the assertion *fails*.

Intuitively, an expression A is *safe* if, once it is translated into Formal-RCF$^{\forall}_{\wedge\vee}$, all assertions succeed in all evaluations. When reasoning about implementations of crypto-

graphic protocols, we are interested in the safety of programs executed in parallel with an arbitrary attacker. This property is called *robust safety* and is stated formally in §5 and statically enforced by our type system from §4.

We consider a variant of first-order logic with equality as the authorization logic. We assume that $\text{RCF}^{\forall}_{\wedge\vee}$ values are the terms of this logic, and equality $M = N$ is interpreted as syntactic equality between values.

4 Type System

This section presents our type system for enforcing authorization policies on $\text{RCF}^{\forall}_{\wedge\vee}$ code. This extends the type system proposed by Bengtson et al. [14] with union, intersection, and polymorphic types. Additionally, we encode a new type Private, which is used to characterize data that are not known to the attacker, and introduce a novel relation for statically reasoning about the disjointness of types. In the following we explain the typing judgements and present the most important typing rules.

4.1 Typing Environment and Entailment

A typing environment E is a list of bindings for variables ($x : T$), type variables (α or $\alpha :: k$), names ($a \updownarrow T$, where the name a stands for a channel conveying values of type T), and formulas (bindings of the form $\{C\}$). An environment is well-formed ($E \vdash \diamond$) if all variables, names, and type variables are defined before use, and no duplicate definitions exist. A type T is well-formed in environment E (written $E \vdash T$) if all its free variables, names, and type variables are defined in E.

A crucial judgment in the type system is $E \vdash C$, which states that the formula C is derivable from the formulas in E. Intuitively, our type system ensures that whenever $E \vdash C$ we have that C is logically entailed by the global formula log at execution time. This judgment is used for instance when type-checking $\mathsf{assert}\, C$ using (Exp Assert): type-checking succeeds only if C is entailed in the current typing environment.

4.2 Subtyping and Kinding

Intuitively, all data sent to and received from an untrusted channel have type Un, since such channels are considered under the complete control of the adversary. However, a system in which only data of type Un can be communicated over the untrusted network would be too restrictive, e.g., a value of type $\{x : \mathsf{Un} \mid Ok(x)\}$ could not be sent over the network. We therefore consider a *subtyping relation* on types, which allows a term of a subtype to be used in all contexts that require a term of a supertype. This preorder is most often used to compare types with type Un. In particular, we allow values having type T that is a subtype of Un, denoted $T <: \mathsf{Un}$, to be sent over the untrusted network, and we say that the type T has *kind public* in this case. Similarly, we allow values of type Un that are received from the untrusted network to be used as values of type U, provided that $\mathsf{Un} <: U$, and in this case we say that type U has *kind tainted*. We outline some important rules for kinding and subtyping (let k range over pub and tnt).

Kinding and subtyping for refinement types

(Kind Refine Pub)
$$\frac{E \vdash \{x : T \mid C\} \quad E \vdash T :: \mathsf{pub}}{E \vdash \{x : T \mid C\} :: \mathsf{pub}}$$

(Kind Refine Tnt)
$$\frac{E \vdash T :: \mathsf{tnt} \quad E, x : T \vdash C}{E \vdash \{x : T \mid C\} :: \mathsf{tnt}}$$

(Sub Refine Left)
$$\frac{E \vdash \{x : T \mid C\} \quad E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'}$$

(Sub Refine Right)
$$\frac{E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}$$

The refinement type $\{x : T \mid C\}$ is a subtype of T. This allows us to discard logical formulas when they are not needed. For instance, a value of type $\{x : \mathsf{Un} \mid Ok(x)\}$ can be sent on a channel of type Un. Conversely, the type T is a subtype of $\{x : T \mid C\}$ only if $\forall x.C$ is entailed in the current typing environment, so by subtyping we can only add universally valid formulas.

Kinding for pair and function types

(Kind Pair)
$$\frac{E \vdash T :: k \quad E, x : T \vdash U :: k}{E \vdash (x : T * U) :: k}$$

(Kind Fun)
$$\frac{E \vdash T :: \overline{k} \quad E, x : T \vdash U :: k}{E \vdash (x : T \to U) :: k}$$

A pair type $T * U$ is public (or tainted) only if both T and U are public (respectively tainted). On the other hand, a function type $T \to U$ is public only if the return type U is public (otherwise $\lambda x{:}\mathsf{unit}.\, M_{\text{secret}}$ would be public) and the argument type T is tainted (otherwise $\lambda k : \mathsf{PrivKey}\langle\mathsf{Private}\rangle.\, \mathsf{let}\ x = \mathit{encrypt}\langle\mathsf{Private}\rangle\ k\ M_{\text{secret}}\ \mathsf{in}\ a_{\text{pub}}!x$ would be public).

Kinding and subtyping for union and intersection types (*)

(Kind And Pub 1)
$$\frac{E \vdash T_1 :: \mathsf{pub} \quad E \vdash T_2}{E \vdash T_1 \wedge T_2 :: \mathsf{pub}}$$

(Kind And Pub 2)
$$\frac{E \vdash T_1 \quad E \vdash T_2 :: \mathsf{pub}}{E \vdash T_1 \wedge T_2 :: \mathsf{pub}}$$

(Kind And Tnt)
$$\frac{E \vdash T_1 :: \mathsf{tnt} \quad \Gamma \vdash T_2 :: \mathsf{tnt}}{\Gamma \vdash T_1 \wedge T_2 :: \mathsf{tnt}}$$

(Kind Or Pub)
$$\frac{E \vdash T_1 :: \mathsf{pub} \quad E \vdash T_2 :: \mathsf{pub}}{E \vdash T_1 \vee T_2 :: \mathsf{pub}}$$

(Kind Or Tnt 1)
$$\frac{E \vdash T_1 :: \mathsf{tnt} \quad E \vdash T_2}{E \vdash T_1 \vee T_2 :: \mathsf{tnt}}$$

(Kind Or Tnt 2)
$$\frac{E \vdash T_1 \quad E \vdash T_2 :: \mathsf{tnt}}{E \vdash T_1 \vee T_2 :: \mathsf{tnt}}$$

(Sub And LB 1)
$$\frac{E \vdash T_1 <: U \quad E \vdash T_2}{E \vdash T_1 \wedge T_2 <: U}$$

(Sub And LB 2)
$$\frac{E \vdash T_1 \quad E \vdash T_2 <: U}{E \vdash T_1 \wedge T_2 <: U}$$

(Sub And Greatest)
$$\frac{E \vdash T' <: T_1 \quad E \vdash T' <: T_2}{E \vdash T' <: T_1 \wedge T_2}$$

(Sub Or Least)
$$\frac{E \vdash T_1 <: U \quad E \vdash T_2 <: U}{E \vdash T_1 \vee T_2 <: U}$$

(Sub Or UB 1)
$$\frac{E \vdash T <: U_1 \quad E \vdash U_2}{E \vdash T <: U_1 \vee U_2}$$

(Sub Or UB 2)
$$\frac{E \vdash U_1 \quad E \vdash T <: U_2}{E \vdash T <: U_1 \vee U_2}$$

The intersection type $T_1 \wedge T_2$ can intuitively be seen as a[3] greatest lower bound of the types T_1 and T_2. Rules (Sub And LB 1) and (Sub And LB 2) ensure that $T_1 \wedge T_2$ is a lower bound: by using reflexivity in the premise we obtain that $T_1 \wedge T_2 <: T_1$ and $T_1 \wedge T_2 <: T_2$. Rule (Sub And Greatest) ensures that $T_1 \wedge T_2$ is greater than any other lower bound: if T' is another lower bound of T_1 and T_2 then T' is a subtype of $T_1 \wedge T_2$. As far as kinding is concerned, the type $T_1 \wedge T_2$ is public if T_1 is public or T_2 is public, and it is tainted if both T_1 and T_2 are tainted.

The union type $T_1 \vee T_2$ intuitively corresponds to a least upper bound of T_1 and T_2. The rules for union types are exactly the dual of the ones for intersection types.

[3] The subtyping relation of RCF is not anti-symmetric, so least and greatest elements are not necessarily unique.

Our type system has no distributivity rules between union and intersection types and the primitive type constructors. Some distributivity rules are derivable from the primitive rules above: for instance we can prove that $T \to (U_1 \wedge U_2)$ is a subtype of $(T \to U_1) \wedge (T \to U_2)$, but not the other way around. In fact adding a subtyping rule in the other direction would be unsound [26], since in our system functions can have side-effects and such distributivity rules would allow circumventing the value restriction on the introduction of intersection types (see §4.4 and §9).

Kinding and subtyping rules for universal types

(Kind Univ*)
$$\frac{E, \alpha \vdash T :: k}{E \vdash \forall \alpha. T :: k}$$

(Sub Univ*)
$$\frac{E, \alpha \vdash T <: U}{E \vdash \forall \alpha. T <: \forall \alpha. U}$$

Finally, the rule for subtyping polymorphic types (Sub Univ*) is simple: the type $\forall \alpha. T$ is subtype of $\forall \alpha. U$ if T is a subtype of U. Similarly, $\forall \alpha. T$ has kind k if T has kind k in an environment extended with a binding for α. Note that α can be substituted by any type, so we cannot assume anything about α when checking that $T :: k$ and $T <: U$ respectively.

Kinding and subtyping rules for recursive types

(Kind Rec)
$$\frac{E, \alpha :: k \vdash T :: k}{E \vdash (\mu \alpha. T) :: k}$$

(Sub Pos Rec*)
$$\frac{E, \alpha \vdash T <: U \quad \alpha \text{ only occurs positively in } T \text{ and } U}{E \vdash \mu \alpha. T <: \mu \alpha. U}$$

(Sub Refl*)
$$\frac{E \vdash T}{E \vdash T <: T}$$

The rule (Sub Pos Rec*) for subtyping recursive types is new, and differs significantly from Cardelli's Amber rule [5, 22], which is used by the original RCF:

Cardelli's Amber rule (used by the original RCF)

(Sub Rec)
$$\frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \neq \alpha' \quad \alpha \notin \mathit{ftv}(T') \quad \alpha' \notin \mathit{ftv}(T)}{E \vdash \mu \alpha. T <: \mu \alpha'. T'}$$

The soundness of the Amber rule (Sub Rec) is hard to prove syntactically [14] – in particular proving the transitivity of subtyping in the presence of the Amber rule requires a complicated inductive argument, which only works for "executable" environments (see [14]), as well as spurious restrictions on the usage of type variables in the rules (Sub Refl*), (Kind And Pub 1), (Kind And Pub 2), (Kind Or Tnt 1), (Kind Or Tnt 2), (Sub And LB 1), (Sub And LB 2), (Sub Or UB 1), (Sub Or UB 2). We use the simpler (Sub Pos Rec*) rule, which is much easier to prove sound and requires no restrictions on the other rules. It resembles (Sub Univ*), our rule for subtyping universal types, with the additional restriction that the recursive variable is not allowed to appear in a contravariant position (such as $\alpha \to T$). While this positivity restriction is crucial for the soundness of the (Sub Pos Rec*) rule, this does not pose any problem in practice, where most of the time only positive recursive types [40,48] are used. Moreover, this positivity restriction only affects subyping, so programs involving negative occurrences of recursion variables that do not involve subtyping can still be properly type-checked (e.g., we can still type-check the encodings of fixpoint combinators on expressions [14])

4.3 Encoding Types Un and Private in RCF

In RCF [14] the type Un is in fact not primitive. By the (Sub Pub Tnt) rule that relates kinding and subtyping, any type that is both public and tainted is equivalent to Un. Since type unit is both public and tainted, Un is actually encoded as unit.

The (Sub Pub Tnt) rule and kinding for type unit

$$\frac{E \vdash T :: \mathsf{pub} \quad E \vdash U :: \mathsf{tnt}}{E \vdash T <: U} \text{ (Sub Pub Tnt)} \qquad \frac{E \vdash \diamond}{E \vdash \mathsf{unit} :: k} \text{ (Kind Unit)}$$

The (Sub Pub Tnt) rule equates many of the types in the system. For instance in RCF all the following types are equivalent: Un, $\mathsf{Un} \to \mathsf{Un}$, $\mathsf{Un} * \mathsf{Un}$, $\mathsf{Un} + \mathsf{Un}$, $\mu\alpha.\,\mathsf{Un}$, and $\forall\alpha.\,\mathsf{Un}$. As a consequence it is hard to come up with RCF types that do not share any values with type Un, a property we want for our Private type. Perhaps unintuitively, it is not enough that a type is not public and not tainted to make it disjoint from Un. A final observation is that, in $\mathrm{RCF}^{\forall}_{\wedge\vee}$, in an inconsistent environment ($E \vdash \mathsf{false}$) *all* types are equivalent and all values inhabit all types. This means that Private being disjoint from Un is relative to the formulas in the environment.

Encoding type Private

$$\{C\} \triangleq \{x : \mathsf{unit} \mid C\} \qquad x \notin \mathit{free}(C)$$
$$\mathsf{Private}_C \triangleq \{f : \{C\} \to \mathsf{Un} \mid \exists x.\ f = \lambda y : \{C\}.\ \mathsf{assert}\ C; x\} \qquad \mathsf{Private} \triangleq \mathsf{Private}_{\mathsf{false}}$$

We therefore encode a more general type $\mathsf{Private}_C$, read "private unless C". The values in this type are not known to the attacker, unless the formula C is entailed by the environment. Intuitively, if the attacker would know a value of this type, then he could call it (values of type $\mathsf{Private}_C$ have to be functions), which would exercise the $\mathsf{assert}\ C$ and invalidate the safety of the system, unless C can be derived from the formula log. Type $\mathsf{Private}_C$ resembles a singleton type, in that it contains only values of a very specific form. We use an existential quantifier over values to ensure that there are infinitely many values of this type. The type Private is obtained as $\mathsf{Private}_{\mathsf{false}}$.

4.4 Typing Values and Expressions

The main judgments of the type system we consider are $E \vdash M : T$, which states that value M has type T, and $E \vdash A : T$, stating that expression A returns a value of type T. These two judgements are mutually-inductively defined, and the most important typing rules are reported below. Most of them are standard, so we focus the explanation only on the rules that are new with respect to [14].

Selected rules for typing values $E \vdash M : T$

$$\frac{E, x : T \vdash A : U}{E \vdash \lambda x : T.\, A : (x : T \to U)} \text{ (Val Lam)} \qquad \frac{E, \alpha \vdash A : T}{E \vdash \Lambda\alpha.\, A : \forall\alpha.\, T} \text{ (Val TLam*)} \qquad \frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x : T \mid C\}} \text{ (Val Refine)}$$

$$\frac{E \vdash M : T \quad E \vdash M : U}{E \vdash M : T \wedge U} \text{ (Val And*)} \qquad \frac{E \vdash M\{\widetilde{T}/\widetilde{\alpha}\} : V}{E \vdash \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}.\ M : V} \text{ (Val For 1*)} \qquad \frac{E \vdash M\{\widetilde{U}/\widetilde{\alpha}\} : V}{E \vdash \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}.\ M : V} \text{ (Val For 2*)}$$

Rule (Val And*) allows us to give value M an intersection type $T \wedge U$, if we can give M both type T and type U. As discovered by Davies and Pfenning [26] the value restriction is crucial for the soundness of this introduction rule in the presence of side-effects (also see §9). Also, unrelated to the value restriction, this rule is not very useful on its own: since we are in a calculus with typing annotations, it is hard to give one annotated value two different types. For instance, if we want to give the identity function type $(\mathsf{Private}\to\mathsf{Private}) \wedge (\mathsf{Un}\to\mathsf{Un})$ we need to annotate the argument with type $\mathsf{Private}$ (i.e., $\lambda x{:}\mathsf{Private}.\, x$) in order to give it type $\mathsf{Private}\to\mathsf{Private}$, but then we cannot give this value type $\mathsf{Un}\to\mathsf{Un}$. Following Pierce [42, 43] and Reynolds [44] we use the for construct to explicitly alternate type annotations. For instance, the identity function of type $(\mathsf{Private}\to\mathsf{Private}) \wedge (\mathsf{Un}\to\mathsf{Un})$ can be written as $(\mathsf{for}\ \alpha\ \mathsf{in}\ \mathsf{Private}; \mathsf{Un}.\ \lambda x{:}\alpha.\, x)$. By rule (Val For 1*) we can give this value type $\mathsf{Private}\to\mathsf{Private}$ if we can give value $\lambda x{:}\mathsf{Private}.\, x$ the same type, which is trivial. Similarly, by (Val For 2*) we can give the for value type $\mathsf{Un}\to\mathsf{Un}$, so by (Val And*) we can also give it the desired intersection type.

Selected rules for typing expressions $E \vdash A : T$

(Exp Appl)
$$\frac{E \vdash M : (x : T \to U) \quad E \vdash N : T}{E \vdash M\ N : U\{N/x\}}$$

(Exp Inst*)
$$\frac{E \vdash M : \forall\alpha.\, U}{E \vdash M\langle T\rangle : U\{T/\alpha\}}$$

(Exp Assert)
$$\frac{E \vdash C}{E \vdash \mathsf{assert}\ C : \mathsf{unit}}$$

(Exp If*)
$$\frac{\begin{array}{c} E \vdash M : T_1 \quad E \vdash N : T_2 \quad \vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow C \\ E, x : T_1 \wedge T_2, \{x = M \wedge M = N \wedge C\} \vdash A : U \qquad E, \{M \neq N\} \vdash B : U \end{array}}{E \vdash \mathsf{if}\ M = N\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B : U}$$

(Exp Case*)
$$\frac{E \vdash M : T_1 \vee T_2 \quad E, x : T_1 \vdash A : U \quad E, x : T_2 \vdash A : U}{E \vdash \mathsf{case}\ x = M : T_1 \vee T_2\ \mathsf{in}\ A : U}$$

(Exp Subsum)
$$\frac{E \vdash A : T \quad E \vdash T <: T'}{E \vdash A : T'}$$

Union Types are introduced by subtyping (T_1 is a subtype of $T_1 \vee T_2$ for any T_2), and eliminated by a $\mathsf{case}\ x = M : T_1 \vee T_2\ \mathsf{in}\ A$ expression [42] using the (Exp Case*) rule.[4] Given a value M of type $T_1 \vee T_2$, we do not know whether M is of type T_1 or of type T_2, so we have to type-check A under each of these assumptions. This is useful when type-checking code interacting with the attacker. For instance, suppose that a party receives a value encrypted with a public-key that is used by honest parties to encrypt messages of type T (as in the protocol from §2). After decryption, the obtained plaintext is given type $T \vee \mathsf{Un}$ since it might come from a honest party as well as from the attacker. We have thus to type-check the remaining code twice, once under the assumption that x is of type T, and once assuming that x is of type Un.

The rule (Exp If*) exploits intersection types for strengthening the type of the values tested for equality in the conditional $\mathsf{if}\ M = N\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$. If M is of type T_1 and N is of type T_2, then we type-check A under the assumption that $x = M \wedge M = N$, and x is of type $T_1 \wedge T_2$. This corresponds to a type-cast that is always safe, since the conditional succeeds only if M is syntactically equal to N, in which case the common value has indeed both the type of M and the type of N. This is useful for type-checking the symbolic implementations of digital signatures (see §6.2) and zero-knowledge (see §7). Additionally, if the equality test of the conditional succeeds then the types T_1 and T_2

[4] As pointed out by Dunfield and Pfenning [30] eliminating union types for expressions that are not in evaluation contexts is unsound in the presence of non-determinism (this is further discussed in §9).

are not disjoint. However, certain types such as Un and Private have common values only if the environment is inconsistent (i.e., $E \vdash \mathsf{false}$). Therefore, when comparing values of disjoint types it is safe to add false to the environment when type-checking A, which makes checking A always succeed. Intuitively, if T_1 and T_2 are disjoint the conditional cannot succeed, so the expression A will not be executed. This idea has been applied in [1] for verifying secrecy properties of nonce handshakes, but later disappeared in the more advanced type systems for authorization policies.

Non-disjointness of types (*) $\vdash \mathsf{NonDisj}\ T\ U \rightsquigarrow C$

(ND Private Un)
$$\frac{fv(C) = \emptyset}{\vdash \mathsf{NonDisj}\ \mathsf{Private}_C\ \mathsf{Un} \rightsquigarrow C}$$

(ND True)
$$\frac{}{\vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow \mathsf{true}}$$

(ND Sym)
$$\frac{\vdash \mathsf{NonDisj}\ T_2\ T_1 \rightsquigarrow C}{\vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow C}$$

(ND Refine)
$$\frac{\vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow C}{\vdash \mathsf{NonDisj}\ \{x : T_1 \mid C_1\}\ T_2 \rightsquigarrow C}$$

(ND Rec)
$$\frac{\vdash \mathsf{NonDisj}\ (T\{\alpha/\mu\alpha.\,T\})\ (U\{\beta/\mu\beta.\,U\}) \rightsquigarrow C}{\vdash \mathsf{NonDisj}\ (\mu\alpha.\,T)\ (\mu\beta.\,U) \rightsquigarrow C}$$

(ND Pair)
$$\frac{\vdash \mathsf{NonDisj}\ T_1\ U_1 \rightsquigarrow C_1 \quad \vdash \mathsf{NonDisj}\ T_2\ U_2 \rightsquigarrow C_2}{\vdash \mathsf{NonDisj}\ (T_1 * T_2)\ (U_1 * U_2) \rightsquigarrow C_1 \wedge C_2}$$

(ND Sum)
$$\frac{\vdash \mathsf{NonDisj}\ T_1\ U_1 \rightsquigarrow C_1 \quad \vdash \mathsf{NonDisj}\ T_2\ U_2 \rightsquigarrow C_2}{\vdash \mathsf{NonDisj}\ (T_1 + T_2)\ (U_1 + U_2) \rightsquigarrow (C_1 \vee C_2)}$$

(ND And)
$$\frac{\vdash \mathsf{NonDisj}\ T_1\ U \rightsquigarrow C_1 \quad \vdash \mathsf{NonDisj}\ T_2\ U \rightsquigarrow C_2}{\vdash \mathsf{NonDisj}\ (T_1 \wedge T_2)\ U \rightsquigarrow C_1 \wedge C_2}$$

(ND Or)
$$\frac{\vdash \mathsf{NonDisj}\ T_1\ U \rightsquigarrow C_1 \quad \vdash \mathsf{NonDisj}\ T_2\ U \rightsquigarrow C_2}{\vdash \mathsf{NonDisj}\ (T_1 \vee T_2)\ U \rightsquigarrow C_1 \vee C_2}$$

We take this idea a lot further: we inductively define a ternary relation, which relates two types with a logical formula. If $\vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow C$ holds then any environment E in which T_1 and T_2 have a common value, has to entail the condition C (i.e., $E \vdash C$). The base case of this relation is $\vdash \mathsf{NonDisj}\ \mathsf{Private}_C\ \mathsf{Un} \rightsquigarrow C$, in particular $\vdash \mathsf{NonDisj}\ \mathsf{Private}\ \mathsf{Un} \rightsquigarrow \mathsf{false}$. We call two types *provably disjoint* if $\vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow C$ for some formula C that logically entails false, so Private and Un are provably disjoint. Intuitively, two provably disjoint types have common values only in an inconsistent environment.

The other inductive rules lift the NonDisj relation to refinement, pair, sum, recursive, union, and intersection types. We explain two of them in terms of provable disjointness. In order to show that two (non-dependent) pair types $(T_1 * T_2)$ and $(U_1 * U_2)$ are provably disjoint, we apply rule (ND Pair) and we need to show that T_1 and U_1 are provably disjoint, or that T_2 and U_2 are provably disjoint (a conjunction is false if at least one of the conjuncts is false). On the other hand, in order to show that two sum types $(T_1 + T_2)$ and $(U_1 + U_2)$ are disjoint using (ND Sum) we need to show both that T_1 and U_1 are disjoint and that T_2 and U_2 are disjoint.

To illustrate the expressivity of this definition we consider a type for binary trees: $\mathsf{tree}\langle\alpha\rangle \triangleq \mu\beta.\,\alpha + (\alpha * \beta * \beta)$. Each node in the tree is either a leaf or has two children, and both kind of nodes store some information of type α. We can show that $\mathsf{tree}\langle\mathsf{Private}\rangle$ and $\mathsf{tree}\langle\mathsf{Un}\rangle$ are provably disjoint. By (ND Rec) we need to show that the unfolded types $\mathsf{Private} + (\mathsf{Private} * \mathsf{tree}\langle\mathsf{Private}\rangle * \mathsf{tree}\langle\mathsf{Private}\rangle)$ and $\mathsf{Un} + (\mathsf{Un} * \mathsf{tree}\langle\mathsf{Un}\rangle * \mathsf{tree}\langle\mathsf{Un}\rangle)$ are disjoint. By (ND Sum) we need to show both that Private and Un are disjoint, which is immediate by (ND Private Un), and that the pair types $(\mathsf{Private} * \mathsf{tree}\langle\mathsf{Private}\rangle * \mathsf{tree}\langle\mathsf{Private}\rangle)$ and $(\mathsf{Un} * \mathsf{tree}\langle\mathsf{Un}\rangle * \mathsf{tree}\langle\mathsf{Un}\rangle)$ are disjoint. For the latter, by (ND Pair)

it suffices to show that the types of the first components of the pair are disjoint, which follows again by (ND Private Un).

Finally, we remark that the property we called provable disjointness in this section is a tractable (mostly syntax-directed) approximation for the real disjointness of types. This approximation is formally proven sound in Theorem 2 from §5.

5 Formalization Results

We have formalized the metatheory of $\text{RCF}^{\forall}_{\wedge\vee}$ in the Coq proof assistant. We achieve this by defining Formal-$\text{RCF}^{\forall}_{\wedge\vee}$, a core calculus where terms are encoded using a *locally nameless representation* [6, 34]: free variables, free type variables and free RCF names are represented in a named way, while bound variables, bound type variables and bound names are represented using de Bruijn indices. Each alpha-equivalence class has thus a unique representation, avoiding the difficulties associated with alpha-renaming. Besides the formalization of binders, the only other difference between Formal-$\text{RCF}^{\forall}_{\wedge\vee}$ and $\text{RCF}^{\forall}_{\wedge\vee}$ is that in Formal-$\text{RCF}^{\forall}_{\wedge\vee}$ *all type annotations from values, expressions and formulas are erased*.

Type erasure for selected values and expressions

$\lVert \lambda x : T.\, A \rVert = \mathsf{v\_lam}\ (\mathsf{close}_x\ \lVert A \rVert)$	$\lVert \Lambda\alpha.\, A \rVert = \mathsf{v\_tlam}\ \lVert A \rVert$
$\lVert \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}.\, M \rVert = \lVert M \rVert$	$\lVert M\langle T \rangle \rVert = \mathsf{e\_inst}\ \lVert M \rVert$
$\lVert \mathsf{case}\ x = M : T \vee U\ \mathsf{in}\ A \rVert = \mathsf{e\_let}\ \lVert M \rVert\ (\mathsf{close}_x\ \lVert A \rVert)$	

While this erasure process is straightforward, it is crucial for the soundness of the type system that the operational semantics and authorization logic work on erased values. The following type derivation illustrates this aspect.

$$\dfrac{\dfrac{\emptyset \vdash M\{T_1/\alpha\} : T \quad \emptyset \models M\{T_1/\alpha\} = M\{T_1/\alpha\}}{\emptyset \vdash M\{T_1/\alpha\} : \{x : T \mid x = M\{T_1/\alpha\}\}}}{\emptyset \vdash \mathsf{for}\ \alpha\ \mathsf{in}\ T_1; T_2.\, M : \{x : T \mid x = M\{T_1/\alpha\}\}}$$

It uses the (Val Refine) rule to give $M\{T_1/\alpha\}$ a singleton type, and then the (Val For 1*) rule to give the *same* singleton type to the value $(\mathsf{for}\ \alpha\ \mathsf{in}\ T_1; T_2.\, M)$. The only way this can possibly work is because the logic equates $M\{T_1/\alpha\}$ and $(\mathsf{for}\ \alpha\ \mathsf{in}\ T_1; T_2.\, M)$, by working on values where all type annotations and the for construct for type annotation alternation are completely erased. So in our setting the main motivation for doing type erasure is not efficiency, but the soundness of the type system.

Another benefit of doing type erasure is that it makes Formal-$\text{RCF}^{\forall}_{\wedge\vee}$ very close to the original RCF [14], which is also extrinsically typed. In particular the operational semantics of Formal-$\text{RCF}^{\forall}_{\wedge\vee}$[5] corresponds directly to the one of the original RCF, which is defined in terms of a heating relation that allows for syntactic rearrangements of concurrent expressions ($e \Rrightarrow e'$) and a standard reduction relation ($e \to e'$). To prevent confusion, in the following we use e to stand for the expressions, v for the values, and F for the formulas of Formal-$\text{RCF}^{\forall}_{\wedge\vee}$.

[5] Note that while Formal-$\text{RCF}^{\forall}_{\wedge\vee}$ has an operational semantics of its own, $\text{RCF}^{\forall}_{\wedge\vee}$ is only given semantics by translation into Formal-$\text{RCF}^{\forall}_{\wedge\vee}$ (i.e., type erasure).

As advertised by Aydemir et al. [6], in our core language the inductive rules are defined using *cofinite quantification*. This yields strong induction and inversion principles for the relations of the system, and obviates the need for reasoning about alpha-equivalence.

Two of the rules using cofinite quantification

$$\frac{\forall a \notin L.\ \mathsf{open}_a\ e_1 \Rrightarrow \mathsf{open}_a\ e_2}{\mathsf{e\_new}\ e_1 \Rrightarrow \mathsf{e\_new}\ e_2} \qquad \frac{\forall \alpha \notin L.\ E, \alpha \Vdash e : \mathsf{open}_\alpha\ T}{E \Vdash \mathsf{v\_tlam}\ e : \mathsf{t\_univ}\ T}$$

When applying such a rule forwards, one has to choose a finite set L of avoided names (for instance the domain of E), and then has to prove the premise of the rule for an arbitrary name that is not in the set L. This provides a stronger induction principle, since for these rules the induction hypothesis will hold for all names except those in some finite set L, rather than just for a single name.

We have proved that the typing judgements of $\text{RCF}^{\forall}_{\wedge\vee}$ are preserved by type erasure. This proof relies on standard [6] renaming lemmas for the Formal-$\text{RCF}^{\forall}_{\wedge\vee}$ judgements (we use $\Vdash$ to denote Formal-$\text{RCF}^{\forall}_{\wedge\vee}$ judgements).

Lemma 1 (Renaming for $E \Vdash e : T$).
If $x, y \notin dom(E) \cup fv(e, T)$ and $E, x : U \Vdash \mathsf{open}_x\ e : T$ then $E, y : U \Vdash \mathsf{open}_y\ e : T$

Theorem 1 (Adequacy of $\text{RCF}^{\forall}_{\wedge\vee}$ Type System).
For all typing judgements $\mathcal{J}$, if $E \vdash \mathcal{J}$ then $[\![E]\!] \Vdash [\![\mathcal{J}]\!]$.

The main result we have proved for the type system is that well-typed expressions are robustly safe. As in [14], the property follows from the subject-reduction property of the type system. We also list a couple of important lemmas and theorems used in the proof.

Lemma 2 (Inconsistent Environment). *If $E \Vdash \mathsf{false}$, $E \Vdash T$ and $free(e) \subseteq dom(E)$ then $E \Vdash e : T$.*

Lemma 3 (Transitivity of Subtyping). *If $E \Vdash T_1 <: T_2$ and $E \Vdash T_2 <: T_3$ then $E \Vdash T_1 <: T_3$.*

Theorem 2 (Non-disjoint). *If $\Vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow F$ and v is a closed value so that $E \Vdash v : T_1$ and $E \Vdash v : T_2$, then $E \Vdash F$.*

Theorem 3 (Reduction Preserves Types). *If $fv(e) = \emptyset$, $E \Vdash e : T$, and $e \to e'$ then $E \Vdash e' : T$.*

Definition 1 (Safety). *A closed expression e is safe if and only if, in all evaluations of e, all assertions succeed.*

Theorem 4 (Safety). *If $\emptyset \Vdash e : T$ then e is safe.*

Definition 2 (Opponent). *An opponent is an expression e that does not contain asserts, free variables or names.*

Definition 3 (Robust Safety). *An expression e is robustly safe if the application $O\ e$ is safe for any opponent O.*

Theorem 5 (Robust Safety for Formal-$\text{RCF}^{\forall}_{\wedge\vee}$).
If $\emptyset \Vdash e : [\![\mathsf{Un}]\!]$ then e is robustly safe.

Corollary 1 (Robust Safety for RCF$^{\forall}_{\wedge\vee}$).
If $\emptyset \vdash A : \mathsf{Un}$ *then* $[\![A]\!]$ *is robustly safe.*

In a similar way to the definition of robust secrecy of Bengtson et al. [14] (which is, however, a property of contexts, not of values), we define a notion of *robustly private values*.

Definition 4 (Robustly Private Values). *We call a value* v *robustly private in* e *unless* C *if* $\mathit{free}(v, e) = \emptyset$ *and the pair expression* $(e, \lambda x.\, \mathsf{if}\ x = v\ \mathsf{then}\ \mathsf{assert}\ C)$ *is robustly safe.*

Intuitively, a robustly private value is not known to the attacker, since if the attacker would somehow produce or obtain such a value, he could pass it as an argument to the lambda abstraction causing the conditional to succeed and the assert to be triggered. It is very easy to show using Theorem 5 (Robust Safety) that every value of type $\mathsf{Private}_C$ is robustly private in e unless C, for any well-typed expression e.

Theorem 6 (Value of Type Private $\Rightarrow$ Robustly Private). *If* $\emptyset \Vdash v : [\![\mathsf{Private}_C]\!]$ *and* $\emptyset \Vdash e : [\![\mathsf{Un}]\!]$ *then* v *is robustly private in* e *unless* C.

The proof of these theorems was formalized in the Coq proof assistant, together with most of the necessary lemmas. The only notable exception is Theorem 1 (Adequacy of Surface Syntax), which is proved by hand. Adequacy proofs are usually done by hand, since formal and informal definitions (e.g. the "variable convention" in our surface syntax) are in general impossible to relate formally.

At the moment, our Coq formalization [9] totals more than 14kLOC,[6] out of which more than 1.5kLOC are just definitions. We used Ott [46] to generate a large part of these definitions from a 1kLOC long Ott specification, but for the more complex rules we often needed to patch the output of Ott. We used lngen [7] to generate an additional 25kLOC of infrastructure lemmas, which proved invaluable when working with the locally nameless representation.

During the formalization we found and fixed three problems in the paper proofs for the original RCF [14]. First, the "Public Down/Tainted Up" lemma was applying "Bound Weakening" in the wrong direction in the arrow type case, disregarding contravariance. Fixing this problem was easy, and only required proving a new lemma for replacing tainted bounds. Second, in the original RCF opponents can contain free names, so the proof of Theorem 5 (Robust Safety) used Theorem 4 (Safety) for a non-empty environment; however, safety was proved only for empty environments. We fixed this by not allowing the opponents to contain free names, since they can already generate names using the $(\nu a \updownarrow T)A$ expression. Finally, the proof of the "Strengthening" lemma in the original RCF [14], and also in other refinement type systems for security [10], is wrong, and the status of the lemma in its original form is still unclear. The solution the RCF authors proposed is to weaken the claim, so that the other results still go through, while avoiding the problem we found. More details are given in the long version [9].

6 Implementation of Symbolic Cryptography

In contrast to process calculi for cryptographic protocols [3,4], RCF$^{\forall}_{\wedge\vee}$ does not have any built-in construct to model cryptography. Cryptographic primitives are instead encoded

[6] All code size figures include whitespace and comments.

using a dynamic sealing mechanism [41], which is based on standard $\text{RCF}^{\forall}_{\wedge\vee}$ constructs. The resulting symbolic cryptographic libraries are type-checked using the regular typing rules. The main advantage is that, adding a new primitive to the library does not involve changes in the calculus or in the soundness proofs: one has just to find a well-typed encoding of the desired cryptographic primitive. In addition, Backes et al. have recently [13] shown that sealing-based libraries for asymmetric cryptography are computationally sound and semantically equivalent to the more traditional Dolev-Yao libraries based on datatype constructors. §6.1 overviews the dynamic sealing mechanism used in [14] to encode symbolic cryptography, while §6.2 and §6.3 show how our expressive type system can be used to improve this encoding and extend the class of supported protocols.

6.1 Dynamic Sealing

The notion of *dynamic sealing* was initially introduced by Morris [41] as a protection mechanism for programs. Later, Sumii and Pierce [47] studied the semantics of dynamic sealing in a λ-calculus, observing a close correspondence with symmetric encryption.

In RCF [14] seals are encoded using pairs, functions, references and lists. A seal is a pair of a *sealing function* and an *unsealing function*, having type:

$$\mathsf{Seal}\,\langle T\rangle = (T \to \mathsf{Un}) * (\mathsf{Un} \to T).$$

The sealing function takes as input a value M of type T and returns a fresh value N of type Un, after adding the pair (M, N) to a secret list that is stored in a reference. The unsealing function takes as input a value N of type Un, scans the list in search of a pair (M, N), and returns M. Only the sealing function and the unsealing function can access this secret list. In RCF, each key-pair is (symbolically) implemented by means of a seal. In the case of public-key cryptography, for instance, the sealing function is used for encrypting, the unsealing function is used for decrypting, and the sealed value N represents the ciphertext.

Let us take a look at the type $\mathsf{Seal}\,\langle T\rangle$. If T is neither public nor tainted, as it is usually the case for *symmetric-key cryptography*, neither the sealing function nor the unsealing function are public, meaning that the symmetric key is kept secret. If T is tainted but not public, as usually the case for *public-key encryption*, the sealing function is public but the unsealing function is not, meaning that the encryption key may be given to the adversary but the decryption key is kept secret. If T is public but not tainted, as typically the case for *digital signatures*, the sealing function is not public and the unsealing function is public, meaning that the signing key is kept secret but the verification key may be given to the adversary.

Although this unified interpretation of cryptography as sealing and unsealing functions is conceptually appealing, it actually exhibits some undesired side-effects when modeling asymmetric cryptography. If the type of a signed message is not public, then the verification key is not public either and cannot be given to the adversary. This is unrealistic, since in most cases verification keys are public even if the message to be signed is not (as in DAA, see §7.1). Moreover, if the type of a message encrypted with a public key is not tainted, then the public key is not public and cannot be given to the adversary. This may be problematic, for instance, when modeling authentication protocols based on public keys as the NSL protocol (see §2), where the type of the encrypted messages is neither public nor tainted.

6.2 Digital Signatures

In this section, we focus on digital signatures and show how union and intersection types can be used to solve the aforementioned problems. The signing key consists of the seal itself and is given type $\mathsf{SigKey}\langle T\rangle \triangleq \mathsf{Seal}\,\langle T\rangle$, as in the original RCF library [14]. The verification key, instead, is encoded as a function that (i) takes the signature x and the signed message t as input; (ii) calls the unsealing function to retrieve the message y bound to x in the secret list; and (iii) returns y if y is equal to t and fails otherwise. In this encoding, the verifier has to know the signed message in order to verify the signature. This is reasonable as, for efficiency reasons, one usually signs a hash of the message as opposed to the message .

Symbolic implementation of signing-verification key pair

$$
\begin{array}{l}
\mathit{mkSigPair} : \forall\alpha.\,\mathsf{unit} \to \mathsf{SigKey}\langle\alpha\rangle * \mathsf{VerKey}\langle\alpha\rangle \\
\mathit{mkSigPair} = \Lambda\alpha.\,\lambda u : \mathsf{unit}. \\
\quad \mathsf{let}\ (\mathit{seal}, \mathit{unseal}) = \mathit{mkSeal}\,\langle\alpha\rangle\ \mathsf{in} \\
\quad \mathsf{let}\ vk = \lambda x : \mathsf{Un}.\,\mathsf{for}\ \beta\ \mathsf{in}\ \top; \mathsf{Un}.\ \lambda t : \beta. \\
\qquad\qquad \mathsf{if}\ t = (\mathit{unseal}\ x)\ \mathsf{as}\ z\ \mathsf{then}\ z\ \mathsf{else}\ \mathit{failwith}\ \text{“verification failed”} \\
\quad \mathsf{in}\ (k, vk)
\end{array}
$$

The type $\mathsf{VerKey}\langle T\rangle$ of a verification key is defined as $\mathsf{Un} \to \big((x : \top \to \{y : T \mid x = y\}) \wedge (\mathsf{Un} \to \mathsf{Un})\big)$. The verification key takes the signature of type Un as first argument. The second part of this type is an intersection of two types: The type $x : \top \to \{y : T \mid x = y\}$ is used to type-check honest callers: the signed message x has any type (top type) and the message y returned by the unsealing function has the stronger type T, which means that the unsealing function casts the type of the signed message from $\top$ down to T. This is safe since the sealing function is not public and can only be used to sign messages of type T. The type $\mathsf{Un} \to \mathsf{Un}$ makes $\mathsf{VerKey}\langle T\rangle$ always public.[7] Hence, in contrast to [14], we can reason about protocols where the signing key is used to sign private messages while the verification key is public (e.g., in DAA [20]). Finally, we present the typed interface of the functions to create and check signatures:

$$
\begin{array}{l}
\mathit{sign} : \forall\alpha.\ (x_{sk} : \mathsf{SigKey}\langle\alpha\rangle \to \alpha \to \mathsf{Un}) \wedge \mathsf{Un} \\
\mathit{check} : \forall\alpha.\ (x_{vk} \colon \mathsf{VerKey}\langle\alpha\rangle \to \mathsf{Un} \to \top \to \alpha) \wedge \mathsf{Un}
\end{array}
$$

We type-check *sign* and *check* twice, to give them intersection types whose right-hand side is Un. While making these functions available to the adversary is not necessary (the attacker can directly use the signing and verification keys to which he has access), this is convenient for the encoding of zero-knowledge we describe in §7 (dishonest verifier cases).

6.3 Public-Key Encryption

For public-key encryption we simply use a seal of type $\mathsf{Seal}\,\langle T \vee \mathsf{Un}\rangle$, i.e., $\mathsf{PrivKey}\langle T\rangle \triangleq \mathsf{Seal}\,\langle T \vee \mathsf{Un}\rangle$ and $\mathsf{PubKey}\langle T\rangle \triangleq (T \vee \mathsf{Un}) \to \mathsf{Un}$. This allows us to obtain the types described in §2.2. In contrast to [14], the encryption key is always public, even if the type T of the encrypted message is not tainted.[8]

[7] A type of the form $\mathsf{Un} \to (T_1 \wedge T_2)$ is public if T_1 or T_2 are public, and in our case $T_2 = \mathsf{Un} \to \mathsf{Un}$ is public.

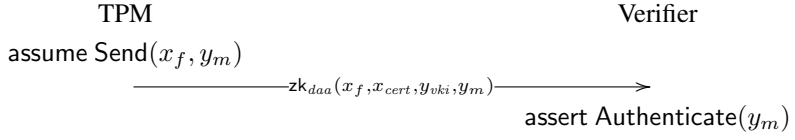
[8] A type of the form $(T_1 \vee T_2) \to \mathsf{Un}$ is public if T_1 or T_2 is tainted, and in our case $T_2 = \mathsf{Un}$ is tainted.

7 Encoding of Zero-knowledge

This section describes how we automatically generate the symbolic implementation of non-interactive zero-knowledge proofs, starting from a high-level specification. Intuitively, this implementation resembles an oracle that provides three operations: one for creating zero-knowledge proofs, one for verifying such proofs, and one for obtaining the public values used to create the proofs. Some of the values used to create a zero-knowledge proof are revealed by the proof to the verifier and to any eavesdropper, while the others (which we call witnesses) are kept secret. A zero-knowledge proof does not reveal any information about these witnesses, other than the validity of the statement being proved.

7.1 Illustrative Example: Simplified DAA

We are going to illustrate our technique on a simplified version[9] of the Direct Anonymous Attestation (DAA) protocol [20]. The goal of the DAA protocol is to enable the TPM to sign arbitrary messages and to send them to an entity called the verifier in such a way that the verifier will only learn that a valid TPM signed that message, but without revealing the TPM's identity. The DAA protocol is composed of two sub-protocols: the *join protocol* and the *DAA-signing protocol*. The join protocol allows a TPM to obtain a certificate x_{cert} from an entity called the issuer. This certificate is just a signature on the TPM's secret identifier x_f. The DAA-signing protocol enables a TPM to authenticate a message y_m by proving to the verifier the knowledge of a valid certificate, but without revealing the TPM's identifier or the certificate. In this section, we focus on the DAA-signing protocol and we assume that the TPM has already completed the join protocol and received the certificate from the issuer. In the DAA-signing protocol the TPM sends to the verifier a zero-knowledge proof.



The TPM proves the knowledge of a certificate x_{cert} of its identifier x_f that can be verified with the verification key y_{vki} of the issuer. Note that although the payload message y_m does not occur in the statement, the proof guarantees non-malleability so an attacker cannot change y_m without redoing the proof. Before sending the zero-knowledge proof, the TPM assumes $\mathsf{Send}(x_f, y_m)$. After verifying the zero-knowledge proof, the verifier asserts $\mathsf{Authenticate}(y_m)$. The authorization policy we consider for the DAA-sign protocol is

$$\mathsf{assume}\ \forall x_f, x_{cert}, y_m.\ \mathsf{Send}(x_f, y_m) \wedge \mathsf{OkTPM}(x_f) \Rightarrow \mathsf{Authenticate}(y_m)$$

where the predicate $\mathsf{OkTPM}(x_f)$ is assumed by the issuer before signing x_f.

7.2 High-level Specification

Our high-level specification of non-interactive zero-knowledge proofs is similar in spirit to the symbolic representation of zero-knowledge proofs in a process calculus [10, 12].

[9] The long version describes the general code generation routine in more detail [9]

For a specification the user needs to provide: (1) variables representing the witnesses and public values of the proof, (2) a Boolean formula over these variables representing the statement of the proof, (3) types for the variables, and, if desired, (4) a promise, i.e., a logical formula that is conveyed by the proof only if the prover is honest.

High-level specification of simplified DAA

$$
\begin{array}{l}
\mathsf{zkdef}\ daa = \\
\quad \mathsf{witness} = [x_f : T_{vki}, x_{cert} : \mathsf{Un}] \\
\quad \mathsf{matched} = [y_{vki} : \mathsf{VerKey}\langle T_{vki}\rangle] \\
\quad \mathsf{public} = [y_m : \mathsf{Un}] \\
\quad \mathsf{statement} = [x_f = \mathsf{check}\langle T_{vki}\rangle\ y_{vki}\ x_{cert}\ x_f] \\
\quad \mathsf{promise} = [\mathsf{Send}(x_f, y_m)] \\
\text{where } T_{vki} = \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\}
\end{array}
$$

Variables. The variables x_f and x_{cert} stand for witnesses. The value of y_{vki} is matched against the signature verification key of the issuer, which the verifier of the zero-knowledge proof already knows. The payload message y_m is returned to the verifier of the proof, so it is public.

Statement. The statement conveyed by a zero-knowledge proof is in general a positive Boolean formula over equality checks. In our example this is $x_f = \mathsf{check}\langle T_{vki}\rangle\ y_{vki}\ x_{cert}\ x_f$.

Types. The user also needs to provide types for the variables. The DAA-sign protocol does not preserve the secrecy of the signed message, so y_m has type Un. On the other hand, the TPM identifier x_f is given a secret and untainted type $T_{vki} = \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\}$. This type ensures that x_f is not known to the attacker and that the predicate $\mathsf{OkTPM}(x_f)$ holds. The verification key of the issuer is used to check signed messages of type T_{vki}, so it is given type $\mathsf{VerKey}\langle T_{vki}\rangle$. Finally the certificate x_{cert} is a signature, so it has type Un. Even though it has type Un, it would break the anonymity of the user to make the certificate a public value, since the verifier could then always distinguish if two consecutive requests come from the same user or not.

Promise. The user can additionally specify a *promise*: an arbitrary authorization logic formula that holds in the typing environment of the prover. If the statement is strong enough to identify the prover as an honest (type-checked) protocol participant (signature proofs of knowledge such as DAA-signing have this property), then the promise can be safely transmitted to the typing environment of the verifier. In the DAA example we have the promise $\mathsf{Send}(x_f, y_m)$, since this predicate holds in the typing environment of a honest TPM.

7.3 Automatic Code Generation

We automatically generate both a typed interface and a symbolic implementation for the oracle corresponding to a zero-knowledge specification.

Generated typed interface for simplified DAA

$$
\begin{array}{l}
\mathsf{create}_{daa} : T_{daa} \vee \mathsf{Un} \to \mathsf{Un} \qquad\qquad \mathsf{public}_{daa} : \mathsf{Un} \to \mathsf{Un} \\
\mathsf{verify}_{daa} : \mathsf{Un} \to ((y_{vki} : \mathsf{VerKey}\langle T_{vki}\rangle \to U_{daa}) \wedge \mathsf{Un} \to \mathsf{Un}) \\
\text{where } T_{daa} = y_{vki} : \mathsf{VerKey}\langle T_{vki}\rangle * y_m : \mathsf{Un} * x_f : T_{vki} * \ x_{cert} : \mathsf{Un} * \{\mathsf{Send}(x_f, y_m)\} \\
\text{and } U_{daa} = \{y_m : \mathsf{Un} \mid \exists x_f, x_{cert}.\ \mathsf{OkTPM}(x_f) \wedge \mathsf{Send}(x_f, y_m)\}
\end{array}
$$

The *generated interface* for DAA contains three functions that share a hidden seal of type $T_{daa} \vee \mathsf{Un}$. The function create_{daa} is used to create zero-knowledge proofs. It takes as argument a tuple containing values for all variables of the proof, or an argument of type Un if it is called by the adversary. In case a protocol participant calls this function, we check that the values have the specified types. Additionally, we check that the promise $\mathsf{Send}(x_f, y_m)$ holds in the typing environment of the prover. The returned zero-knowledge proof is given type Un so that it can be sent over the public network.

The function public_{daa} is used to read the public values of a proof, so it takes as input the sealed proof of type Un and returns y_m, also at type Un.

The function verify_{daa} is used for verifying zero-knowledge proofs. Because of the second part of the intersection type, this function can be called by the attacker, in which case it returns a value of type Un. When called by a protocol participant, however, it takes as argument a candidate zero-knowledge proof of type Un and the verification key of the issuer with type $\mathsf{VerKey}\langle T_{daa}\rangle$. On successful verification, verify_{daa} returns y_m, the only public variable, but with a stronger type than in public_{daa}. The function guarantees that the formula $\exists x_f, x_{cert}.\ \mathsf{OkTPM}(x_f) \wedge \mathsf{Send}(x_f, y_m)$ holds, where the witnesses are existentially quantified. The first conjunct, $\mathsf{OkTPM}(x_f)$, guarantees that if verification succeeds then the statement indeed holds, no matter what the origin of the proof is. This predicate is automatically extracted from the return type of the *check*$\langle T_{vki}\rangle$ function (see §6.2). The second conjunct $\mathsf{Send}(x_f, y_m)$ is the promise of the proof.

The *generated implementation* for this interface creates a fresh seal k_{daa} for values of type $T_{daa} \vee \mathsf{Un}$. The sealing function of k_{daa} is directly used to implement the create_{daa} function. The unsealing function of k_{daa} is used to implement the public_{daa} and verify_{daa} functions. The implementation of public_{daa} is very simple: since the zero-knowledge proof is just a sealed value, public_{daa} unseals it and returns y_m. The witnesses are discarded, and the validity of the statement is not checked.

The implementation of the verify_{daa} function is more interesting. This function takes a candidate zero-knowledge proof z of type Un as input, and a value for the matched variable y_{vki}. Since the type of verify_{daa} contains an intersection type we use a for construct to introduce this intersection type. If the proof is verified by the attacker we can assume that the y_{vki} has type Un and need to type the return value to Un. On the other hand, if the proof is verified by a protocol participant we can assume that y_{vki} has the type $\mathsf{VerKey}\langle T_{vki}\rangle$. In general, it is the strong types of the matched values that allow us to guarantee the strong types of the returned public values, as well as the promise.

Generated symbolic implementation for simplified DAA

$\mathsf{verify}_{daa} = \lambda z : \mathsf{Un}.$
  for α in Un; $\mathsf{VerKey}\langle T_{vki}\rangle$. $\lambda y'_{vki} : \alpha.$
  let $z' = (\mathsf{snd}\ k_{daa})\ z$ in (1)
  case $z'' = z' : \mathsf{Un} \vee T_{daa}$ in (2)
  let $(y_{vki}, y_m, x_f, x_{cert}, \_) = z''$ in (3)
  if $y_{vki} = y'_{vki}$ as y''_{vki} then (4)
    if $x_f = \mathsf{check}\langle T_{vki}\rangle\ y''_{vki}\ x_{cert}\ x_f$ then y_m (5)
    else *failwith* "statement not valid"
  else *failwith* "y_{vki} does not match"

The generated verify_{daa} function performs the following five steps: (1) it unseals z using "snd k_{daa}" and obtains z'; (2) since z' has a union type, it does case analysis on it, and assigns its value to z''; (3) it splits the tuple z'' into the public values (y_{vki} and y_m) and the witnesses (x_f and x_{cert}). (4) it tests if the matched variable y_{vki} is equal to the

argument y'_{vki}, and in case of success assigns the value to the variable y''_{vki} – since y''_{vki} has a stronger type than y'_{vki} and y_{vki} we use this new variable to stand for y_{vki} in the following; (5) it tests if the statement is true by applying the $\mathit{check}\langle T_{vki}\rangle$ function, and checking the result for equality with the value of x_f. In general, this last step is slightly complicated by the fact that the statement can contain conjunctions and disjunctions, so we use decision trees. However, for the DAA example the decision tree has a trivial structure with only one node.

Since the automatically generated implementation of zero-knowledge proofs relies on types and formulas provided by the user, which may both be wrong, the generated implementation is not guaranteed to fulfill its interface. We use our type-checker to check whether this is indeed the case. If type-checking the generated code against its interface succeeds, then this code can be safely used in protocol implementations. Note that because of the for and case constructs the body of verify_{daa} is type-checked four times, corresponding to the following four scenarios: honest prover / honest verifier, honest prover / dishonest verifier, dishonest prover / honest verifier, and dishonest prover / dishonest verifier. In DAA the most interesting case is dishonest prover / honest verifier, when z'' and hence x_f are given type Un, while the result of the signature verification is of type T_{vki}. Since $\vdash \mathsf{NonDisj}\ \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\}\ \mathsf{Un} \rightsquigarrow \mathsf{false}$ by rules (ND Refine) and (ND Private Un), false is added to the environment in which y_m is type-checked. The variable y_m has type Un in this environment, but since this environment is inconsistent y_m can also be given type U_{daa}.

8 Implementation

We have implemented a complete tool-chain for $\mathrm{RCF}^{\forall}_{\wedge\vee}$: it includes a type-checker for the type system described in §4, the automatic code generator for zero-knowledge described in §7, an interpreter, and a visual debugger.

The type-checker supports an extended syntax with respect to the one from §3, including: a simple module system, algebraic data types, recursive functions, type definitions, and mutable references. We use first-order logic with equality as the authorization logic and the type-checker invokes the Z3 SMT solver [27] to discharge proof obligations. The type-checker performed very well in our experiments: it type-checks all our symbolic libraries and samples totaling more than 1.5kLOC in around 12 seconds, on a normal laptop. The type-checker produces an XML log file containing the complete type derivation in case of success, and a partial derivation that leads to the typing error in case of failure. This can be inspected using our visualizer to easily detect and fix flaws in the protocol implementation. The type-checker also performs very limited type inference: it can infer the instantiation of some polymorphic functions from the type of the arguments, however, the user has to provide all the other typing annotations – we would like to improve the amount of type inference in the future.

The type-checker, the code generator for zero-knowledge, and the interpreter are command-line tools implemented in F#, while the GUIs of the visual debugger and the visualizer for type derivations are specified using WPF (Windows Presentation Foundation). The type-checker consists of around 2.5kLOC, while the whole tool-chain has over 5kLOC. All the tools and samples are available at [9].

9 Related Work on Unions and Intersections

The for construct for explicitly alternating type annotations was introduced by Pierce [42, 43] as a generalization of an idea Reynolds [44] used in Forsythe for giving intersection types to annotated lambda abstractions of the form $\lambda x{:}\tau_1..\tau_n.\, e$. In a Church-style system, however, the for construct does not have a clear operational semantics. Compagnoni [25] gives an operational semantics to function application expressions of the form $((\mathsf{for}\ \alpha\ \mathsf{in}\ T; U.\ \lambda x{:}V.\, e_1)\ e_2)$ by pushing the application inside the for – i.e., this expression reduces in one step to $(\mathsf{for}\ \alpha\ \mathsf{in}\ T; U.\ ((\lambda x{:}V.\, e_2)\ e_2))$. It is unclear if this can be generalized to anything other than function applications. Moreover, this reduction rule does not respect the value restriction for the introduction of intersection types (our rule (Val And*) in §4). As discovered by Davies and Pfenning [26] the value restriction on intersection introduction is crucial for soundness in the presence of side-effects. The counterexample they give is in fact very similar to the one used to illustrate the unsoundness of ML, in the absence of the value restriction, due to the interaction of polymorphism with side-effects [36]. Moreover, Davies and Pfenning [26] observed that some standard distributivity laws of subtyping are unsound in a setting with side-effects, since they basically allow one to circumvent the value restriction. We obtain all the benefits of the for construct in $\text{RCF}^{\forall}_{\wedge\vee}$, but erase it completely when translating values into Formal-$\text{RCF}^{\forall}_{\wedge\vee}$, and use the value restriction on both levels to ensure soundness.

The case construct for eliminating union types was introduced by Pierce [42] as a way to make type-checking more efficient, by asking the programmer to annotate the position in the code where union elimination should occur. Dunfield and Pfenning [30] later pointed out that unrestricted elimination of union types is unsound in the presence of non-determinism. This observation is crucial for us, since our calculus, as opposed to the one studied by Dunfield and Pfenning, is in fact non-deterministic. They propose an evaluation context restriction that recovers soundness, but this is not enough to make type-checking efficient. In recent work, Dunfield [29], shows that carefully transforming programs into let-normal form improves efficiency. This is encouraging, since our expressions are already in let normal form, so we can hope to replace the case construct by a normal let in the future, and still preserve efficient type-checking.

10 Conclusions and Future Work

We have presented a new type system that combines refinement types with union types, intersection types, and polymorphic types. A novelty of the type system is its ability to reason statically about the disjointness of types. This extends the scope of the existing type-based analyses of protocol implementations to important classes of cryptographic protocols that were not covered so far, including protocols based on zero-knowledge proofs. Our type system comes with a mechanized proof of correctness and an efficient implementation.

As future work, we plan to investigate the automated generation of concrete cryptographic implementations of zero-knowledge proofs, and thus to complement the generation of symbolic implementations considered in this paper. Also, we intend to apply our framework to analyze implementations of more complex protocols, such as the Civitas electronic voting system [24].

The type-checker we implemented had very good efficiency in our experiments, however, the amount of typing annotations it requires is at the moment quite high. This issue

is more pronounced in our symbolic cryptography library, where intersection and union types are pervasive. This is less of a problem in the code that links against these libraries, and in the case of zero-knowledge even the code in the library is automatically generated together with all the necessary annotations. In the future we would like to perform more type inference, maybe leveraging some of the recent progress on type inference for refinement types [37, 45]. The good news is that intersection and union types can be very useful when devising precise type inference algorithms [10, 38].

Acknowledgments. We thank Cédric Fournet, Andy Gordon, Jan Schwinghammer, and Pierre-yves Strub for the constructive discussions. Thorsten Tarrach implemented the original F5 prototype. Stefan Lorenz helped us with the cryptographic implementation of the DAA protocol. Joshua Dunfield and Kim Pecina commented on a draft. Cătălin Hriţcu is supported by a fellowship from Microsoft Research and the International Max Planck Research School for Computer Science. Matteo Maffei is partially supported by the initiative for excellence of the German federal government, by DFG Emmy Noether program, and by MIUR project "SOFT".

References

1. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 3(298):387–415, 2003.
2. M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.
3. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. 28th Symposium on Principles of Programming Languages (POPL)*, pages 104–115. ACM Press, 2001.
4. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
5. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, 1993.
6. B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *Proc. 35th Symposium on Principles of Programming Languages (POPL '08)*, pages 3–15, 2008.
7. B. E. Aydemir and S. Weirich. LNgen: Tool support for locally nameless representations. Draft available at `http://www.cis.upenn.edu/~sweirich/papers/lngen/`, 2010.
8. M. Backes, M. P. Grochulla, C. Hriţcu, and M. Maffei. Achieving security despite compromise using zero-knowledge. In *22th IEEE Symposium on Computer Security Foundations (CSF 2009)*. IEEE Computer Society Press, July 2009.
9. M. Backes, C. Hriţcu, and M. Maffei. Union and intersection types for secure protocol implementations. Long version, formalization and implementation available at `http://www.infsec.cs.uni-sb.de/projects/F5/`.
10. M. Backes, C. Hriţcu, and M. Maffei. Type-checking zero-knowledge. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 357–370. ACM Press, 2008.
11. M. Backes, M. Maffei, and K. Pecina. A security API for distributed social networks. In *18th Network and Distributed System Security Conference (NDSS'11)*. IEEE Computer Society Press, 2011. To appear.
12. M. Backes, M. Maffei, and D. Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *Proc. 29th IEEE Symposium on Security and Privacy*, pages 202–215. IEEE Computer Society Press, 2008.

13. M. Backes, M. Maffei, and D. Unruh. Computationally sound verification of source code. In *Proc. 17th ACM Conference on Computer and Communications Security (CCS)*, pages 387–398. ACM Press, 2010.
14. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, pages 17–32. IEEE Computer Society Press, 2008. Long version appeared as MSR-TR-2008-118. November 2010 revision that fixes the problems we pointed out is available at `http://research.microsoft.com/en-us/um/people/adg/Publications/MSR-TR-2008-118-SP2.pdf`.
15. K. Bhargavan, R. Corin, C. Fournet, and E. Zălinescu. Cryptographically verified implementations for TLS. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 459–468. ACM Press, 2008.
16. K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *Proc. 37th Symposium on Principles of Programming Languages (POPL '10)*, pages 445–456, 2010.
17. K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 139–152. IEEE Computer Society Press, 2006.
18. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE Computer Society Press, 2001.
19. D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1998.
20. E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proc. 11th ACM Conference on Computer and Communications Security*, pages 132–145. ACM Press, 2004.
21. F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Formal analysis of Kerberos 5. *Theoretical Computer Science*, 367(1):57–87, 2006.
22. L. Cardelli. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. 1997.
23. S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. Technical report, CMU CyLab, October 2008.
24. M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: A secure voting system. In *Proc. 29th IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society Press, 2008.
25. A. B. Compagnoni. Subject reduction and minimal types for higher order subtyping. Technical Report ECS-LFCS-97-363, LFCS, University of Edinburgh, August 1997.
26. R. Davies and F. Pfenning. Intersection types and computational effects. In *Proc. International Conference on Functional Programming (ICFP 2000)*, pages 198–208, 2000.
27. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of TACAS*, 2008.
28. D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.
29. J. Dunfield. Untangling typechecking of intersections and unions. In *Workshop on Intersection Types and Related Systems (ITRS)*, July 2010.
30. J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Proc. 31th Symposium on Principles of Programming Languages (POPL '04)*, pages 281–292. ACM Press, 2004.
31. F. Eigner. Type-based verification of electronic voting systems. Master's thesis, Saarland University, 2009.
32. D. Fisher. Millions of .Net Passport accounts put at risk. *eWeek*, May 2003. (Flaw detected by Muhammad Faisal Rauf Danka).
33. C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF)*, pages 31–45. IEEE Computer Society Press, 2007.

34. A. D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *6th International Workshop on Higher-order Logic Theorem Proving and its Applications (HUG '93)*, pages 413–425, 1993.
35. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2005)*, pages 363–379. Springer, 2005.
36. B. Harper and M. Lillibridge. ML with callcc is unsound. Post to TYPES mailing list, July 8, 1991, archived at `http://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html`.
37. R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. `http://arxiv.org/abs/1004.2884v1`, Dec. 2010.
38. N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proc. 36th Symposium on Principles of Programming Languages (POPL '09)*, pages 416–428, 2009.
39. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 147–166. Springer-Verlag, 1996.
40. N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159–172, 1991.
41. J. Morris. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.
42. B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.
43. B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.
44. J. C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996. Reprinted in O'Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 173-233, Birkhäuser, 1997.
45. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 159–169, 2008.
46. P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *The Journal of Functional Programming*, 20(1):71–122, 2010.
47. E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375(1-3):169–192, 2007.
48. P. Urzyczyn. Positive recursive type assignment. In *Proc. 20th International Symposium Mathematical Foundations of Computer Science (MFCS'95)*, pages 382–391, 1995.
49. D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, 1996.