Union, Intersection, and Refinement Types and Reasoning About Type Disjointness for Secure Protocol Implementations

Michael Backes[1,2], Cătălin Hriţcu[1,3,4,*], Matteo Maffei[1]

[1]CISPA, Saarland University, Saarbrücken, Germany

[2]MPI-SWS, Saarbrücken and Kaiserslautern, Germany

[3]Inria, Paris-Rocquencourt, France

[4]University of Pennsylvania, Philadelphia, USA

January 16, 2014

Abstract

We present a new type system for verifying the security of reference implementations of cryptographic protocols written in a core functional programming language. 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 reference 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.

Keywords: security protocols, reference implementations, zero-knowledge proofs, type systems, verification, refinement types, union types, intersection types, concurrent lambda-calculus, mechanized metatheory

*Corresponding author; e-mail address: catalin.hritcu@gmail.com

1 Introduction

Many of today's applications rely on complex cryptographic protocols for communicating over the insecure Internet (e.g., online banking, electronic commerce, social networks, mobile applications, etc). Protocol designers struggle to keep pace with the variety of possible security vulnerabilities, which have affected early authentication protocols like Needham-Schroeder [69, 97], carefully designed de facto standards like SSL and PKCS [42, 125], and even widely deployed products like Microsoft Passport [77], Kerberos [55, 58], and the SAML-based Single Sign-On for Google Apps [14]. Since manual security analyses of cryptographic protocols are extremely difficult and error-prone, significant effort has been put in developing automated analysis techniques for cryptographic protocols. As a result, security analysts and protocol developers have today at disposal a number of efficient, push-button tools [23, 24, 32, 40, 43, 44, 63].

Most of the existing protocol analysis tools target, however, abstract protocol models that disregard most implementation details. So, even if one proves that a model of a protocol is secure, there is usually no guarantee that an implementation of the same protocol has no security flaws. On top of that, protocol models are often not executable [7], so it is not always easy when writing an abstract model to ensure not only that the model is secure, but also that the model is functional. (If one is only interested in robust safety properties, then a completely dysfunctional model is the most secure.) On the other hand, a reference implementation can be executed, debugged, and tested for interoperability against other implementations of the same protocol specification. One can thus convincingly argue that the best "model" for a security protocol comes in the form of an executable program. And since the manual security analysis of executable programs is hardly possible, it is of paramount importance to devise automated analysis techniques that can provide security guarantees for protocol implementations and, more generally, for the source code of distributed applications.

Adapting the techniques for analyzing protocol models to checking executable code poses in general some important challenges. While abstract protocol models are usually compact, protocol implementations can be large, so the efficiency and scalability of the analysis is even more important. Additionally, implementation code normally makes use of loops, recursion, state, unbounded data structures, higher-order functions, concurrency etc., and many of these programming language features pose significant problems to state-of-the art protocol verifiers like ProVerif [40] when used as a back end for analyzing protocol implementations [37]. The type systems for (functional) programming

languages, on the other hand, were designed with these features in mind, and the analysis they provide is inherently modular. Furthermore, type systems proved successful in the automated analysis of cryptographic protocol models [5, 6, 17–19, 23, 49–54, 78–80, 99] and, more recently, protocol implementations [33, 36, 48, 76, 81].

Despite these promising features, the type-based analysis of reference protocol implementations still poses significant challenges, two of which are addressed in this paper. The first is that many emerging applications (e.g., anonymous authentication [46], electronic voting [60], privacy-aware digital identity management [1, 3], and decentralized social networks [27]) 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 protocol models, 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 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, challenge 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 [97] 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 [46] and Civitas [60]).

1.1 Contributions

This paper presents a new type system for statically verifying authorization policies on reference implementations of cryptographic protocols. Our type theory combines refinement types [33, 34] 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 reference protocol implementations [33, 36] to important protocol classes that were not covered before. 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 non-interactive zero-knowledge proofs.

We focus our attention on reference implementations written in $\text{RCF}^{\forall}_{\wedge\vee}$ [33, 34], a concurrent lambda-calculus that is expressive enough to encode a considerable fragment of a functional programming language like OCaml, Standard ML, or F#.[1] As in the spi-calculus [8], cryptographic operations are considered fully reliable building blocks via a symbolic abstraction of cryptography. As opposed to the spi-calculus, the cryptographic operations are not primitive in $\text{RCF}^{\forall}_{\wedge\vee}$, but are instead encoded using a dynamic sealing mechanism [33, 106, 118], which is in turn based on standard functional programming constructs. The resulting symbolic cryptographic library is thus type-checked using regular typing rules for functional languages; in particular these typing rules are not specific to cryptography. We use union and intersection types to give stronger and more natural types to the operations for asymmetric cryptography than in the original sealing-based symbolic cryptographic library of Bengtson et al. [33]. At the same time, we do preserve the main advantage of the sealing-based library: adding a new cryptographic operation to the library does not involve changes to the calculus or manual proofs, one has just to find a well-typed encoding of the desired cryptographic operation. In addition to hashes, symmetric cryptography, public-key encryption, and digital signatures, our approach supports non-interactive 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 sealing-based symbolic implementation of the corresponding zero-knowledge primitive.

We have formalized the $\text{RCF}^{\forall}_{\wedge\vee}$ calculus, 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-$\text{RCF}^{\forall}_{\wedge\vee}$, and which is obtained from $\text{RCF}^{\forall}_{\wedge\vee}$ by type erasure and by adopting a locally nameless representation for binders [15]. 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 [23, 33] and to propose and evaluate fixes for the faulty proofs. And although our formal proofs are still partial (the proofs of some helper lemmas are not assert-free), they are done in greater detail than similar published paper proofs [23, 33, 34].

Our type-based analysis is automated, modular, efficient, scalable, and provides secu-

[1] While such functional programming languages are not as mainstream as C and Java, they have a much better studied theory and have successfully been used for producing verified reference implementations of security protocols [35, 38].

rity proofs for an unbounded number of sessions. We have implemented a type-checker that performed very well in our experiments. The type-checker features a user-friendly graphical interface for examining typing derivations. The tool-chain we have developed additionally contains the automatic code generator for zero-knowledge proofs, an interpreter, and a visual debugger. The formalization and the implementation are available online [21].

1.2 Focus and Limitations

Given its explicit focus, this work is not without limitations:

- As explained above, this work focuses on simple reference implementations in a core functional language, not on existing implementations in a mainstream programming language. Moreover, this work focuses on symbolic cryptography and Dolev-Yao attackers [70], and does not attempt to link against a concrete cryptographic library. While this would involve additional engineering effort, previous work on F7 and variants [33, 36] has already shown that similar type systems can be successfully integrated into a functional programming language like F#. One interesting issue that is specific to our work is automatically generating a cryptographic implementation of the involved zero-knowledge proof; we discuss this last point as future work in §12.

- More extensive type inference would be necessary for our type-checker to become easier usable by regular programmers. In this paper we focus on the design of the type system, its soundness proof, and the corresponding type-checker, and leave more extensive type inference for future work (see §12).

1.3 Outline

The remainder of the paper is structured as follows: §2 discusses related work. §3 gives an intuitive overview of our type system and exemplifies the most important concepts on a simple authentication protocol. §4 introduces the syntax of $\mathrm{RCF}^{\forall}_{\wedge\vee}$, the language supported by our type-checker. §5 presents our type system. §6 describes the results of our Coq formalization. In §7 we use union and intersection types to give stronger and more natural types to the dynamic sealing-based encoding of asymmetric cryptography. §8 presents our dynamic sealing-based encoding of zero-knowledge proofs. §9 describes our implementation, while §10 reports on several case studies done by us and

others. §11 discusses related work on union and intersection types. §12 concludes and gives some interesting research directions. Appendix A lists the Formal-RCF$^{\forall}_{\wedge\vee}$ calculus, the erasure function from RCF$^{\forall}_{\wedge\vee}$, the operational semantics and the type system of Formal-RCF$^{\forall}_{\wedge\vee}$; Appendix B provides more technical details about our encoding of zero-knowledge proofs.

2 Related Work

Our type system extends the refinement type system by Bengtson et al. [33] with union, intersection, and polymorphic types, as well as with syntactic reasoning about the disjointness of types. We also provide a novel encoding of 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 [5], but later disappeared in the more advanced type systems for authorization policies. Our extension allows the type system 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 by Bengtson et al. [33] poses on the usage of asymmetric cryptography. For instance, if a key is used to sign a secret message, then the corresponding verification key could not be made public. These limitations were preventing the analysis of many interesting cryptographic applications, such as the Direct Anonymous Attestation protocol [46], which involves digital signatures on secret TPM identifiers.

In independent parallel work, Bhargavan et al. [36] have developed an additional cryptographic library for a simplified version of the type system proposed by Bengtson et al. [33]. This library does not rely on dynamic sealing but on datatype 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 previously established invariants. Therefore, extending a symbolic cryptographic library in the style of [36] to new prim-

itives 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. For instance, Eigner [75] reports encoding the sophisticated cryptographic schemes used in the Civitas [60] electronic voting protocol using dynamic seals, in a relatively short amount of time.

The main simplification Bhargavan et al. [36] propose over the type system by Bengtson et al. [33] 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. The security property guaranteed in the result of Bhargavan et al. [36] is thus different from the widely-accepted robust safety property [33, 80, 85, 86]; in particular the property guaranteed by their type system depends on the type system itself giving the right meaning to the attacker interface and properly enforcing it on the attacker. In contrast, by retaining the kinding relation from [33] we also retain the property that *all* attackers are well-typed with respect to our type system (this property is usually called opponent typability [85]), which allows us to prove that our type system enforces robust safety. Nevertheless, Bhargavan et al. [36] manage to solve some of the problems in the original work of Bengtson et al. [33] without relying on union and intersection types. Moreover, if all refinements are over a common base type, such as bytes, then it is usually possible to represent unions and intersection types using the logical connectives inside the refinement types. It would be interesting future work to better compare our work to this approach, and maybe to try to combine the advantages of both approaches in a unified framework.

Backes et al. [23] 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 [19], union and intersection types are used to infer precise type information about the secret witnesses of zero-knowledge proofs even when protocol participants are compromised. This is captured in a separate "statement-based inference" relation, 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. The general technique we introduce in the current paper for reasoning about type disjointness syntactically was recently also

ported back to the spi calculus setting [92].

Backes et al. [30] have recently established a semantic correspondence for asymmetric cryptography between a library based on sealing and one based on datatype constructors, showing that both libraries enjoy computational soundness guarantees. They establish the computational soundness of symbolic trace properties in RCF by translation to the CoSP framework [20]; these properties can then be established by typing, for instance using our type system, or by any other verification technique.

In another very recent work, Fournet et al. [81], develop a probabilistic variant of RCF, and formalize its type safety in Coq. They develop typed modules and interfaces for MACs, signatures, and encryptions, and establish their authenticity and secrecy properties in the setting of concrete cryptography (i.e., security against chosen plaintext and chosen ciphertext attacks). This allows them to establish computational properties by typing in a modular way, including observational equivalences, such as indistinguishability.

Eigner [75] uses our type system to verify eligibility, inalterability, and individual verifiability for a simple implementation of the Civitas electronic voting protocol [60]. These properties are expressed as authorization policies and verified by our type-checker. The sophisticated cryptographic schemes used by the Civitas protocol (i.e., distributed decryption, plaintext equivalence tests, homomorphic encryptions, mix nets, and a variety of zero-knowledge proofs) are all encoded using dynamic seals.

Maffei and Pecina [100] have recently proposed privacy-aware proof-carrying authorization, a framework for the specification and enforcement of authorization policies in decentralized systems. In proof-carrying authorization the request for access to a sensitive resource comes together with a proof showing that the requester has permissions to access the desired resource according to a decentralized policy. Logical formulas of the form "P says F" where principal P endorses formula F are witnessed by the digital signature of P on F. Such certificates are combined to form proofs of more complicated statements, and then verified by the reference monitor protecting the requested resource. These certificates can, however, leak sensitive information to the reference monitor. In privacy-aware proof-carrying authorization, existential quantification in the authorization logic is used to mark information that must be kept secret, and zero-knowledge proofs are used to transmit such formulas to the reference monitor in a privacy-preserving way. The generated cryptographic protocol between the requester and the reference monitor is modeled in $\mathrm{RCF}^{\forall}_{\wedge\vee}$, and the correctness of the authorization decision is verified using our type system.

Goubault-Larrecq and Parrennes developed a static analysis technique [88] based on pointer analysis and Horn clause resolution for cryptographic protocols implemented in C. The analysis is limited to secrecy properties, assumes that the analyzed C program is memory safe, deals only with standard cryptographic primitives, and 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 [59] 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 at discovering attacks. It was used to check secrecy and authentication properties of the main loop of OpenSSL for configurations of up to three servers and three clients. The analysis only deals with standard cryptographic primitives, relies on the specifications of the called functions being correct, and offers only limited scalability.

Dupressoir et al. [74] have recently proposed to use a general-purpose verifier for analyzing C implementations of cryptographic protocols. The technique can prove both memory safety and security properties for an unbounded number sessions. It uses a general theory of symbolic cryptography, independent of any programming language, developed in the Coq proof assistant – this generalizes the invariants for cryptographic structures introduced for F#/RCF by Bhargavan et al. [36]. Properties of this theory are imported as first-order axioms to the verifier. By using a general-purpose C verifier the authors aim to benefit from economies of scale and future improvements in C verification in general. The main remaining challenge is reducing verification times and the number of user supplied annotations.

Bhargavan et al. [37] proposed a technique for the verification of F# protocol implementations by automatically extracting ProVerif models [40], using an extension of the functions as processes encoding [104]. The technique was successfully used to verify implementations of real-world cryptographic protocols such as TLS [35] and Europay-MasterCard-Visa (EMV) [67]. The underlying analysis using ProVerif is, however, not modular and is less robust and less scalable [36] than type-checking. Furthermore, the considered fragment of F# is quite restrictive: it does not include higher-order functions, and it allows only for a very limited usage of recursion and state.

More recently, Aizatulin et al. [10] and Corin and Manzano [62] have proposed techniques for analyzing C programs by extracting abstract models using symbolic execution. The solution by Aizatulin et al. [10] needs neither a pre-existing protocol description nor manual inspection of source code, and uses existing results for the applied pi

calculus [20] to establish computational soundness. Their current prototype can, however, analyze only a single execution path, so it is limited to protocols with no significant branching. The solution by Corin and Manzano [62] seems to be able to handle branches, but it cannot yet handle security properties.

Swamy et al. [120] propose FINE, a security-typed language for enforcing dynamic, stateful policies for access control and information flow tracking using a combination of refinement and affine types. FINE distinguishes itself from RCF, primarily in its ability to express both stateful authorization[2] and information flow. As opposed to RCF, however, FINE is not concurrent and cannot easily express Dolev-Yao attackers and cryptographic operations. In a very recent work, Swamy et al. [121] address the latter limitation by embedding into F*, a novel programming language which encompasses FINE, as well as a fragment of RCF [33] without kinding, without concurrency, and with restrictions on the use of recursive types.

The more technical discussion about the related work on union and intersection types is postponed to §11.

This paper extends a previously published conference version [26] by providing:

- A mechanized formalization of our calculus, of our type system, and of the important parts of the soundness proof in the Coq proof assistant (§6);
- All the rules of our type system (because of space constraints, the conference version only listed a few selected rules);
- A new subsection in the introduction on focus and limitations (§1.2);
- A new section on experimental evaluation (§10);
- A substantially extended related work section (§2);
- A substantially extended discussion about future work (§12);
- A changed title that also mentions "reasoning about type disjointness", one of the main novelties of the type system we propose. The conclusion now emphasizes that this contribution is of independent interest, beyond analyzing security protocols. Furthermore, the type disjointness judgment was further generalized; rules (ND Conj), (ND Entails), (ND Forms And Type), and (ND Sub) are new;
- Generally improved presentation throughout the paper.

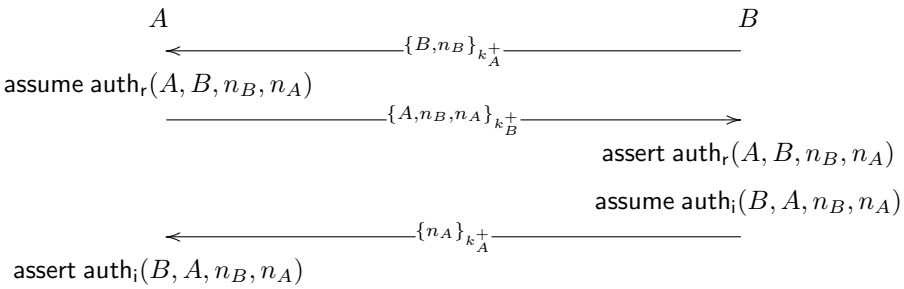
[2] Stateful properties can still be specified and verified within RCF using a refined state monad [45].

3 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 protocol [97] (NSL), which could not be analyzed by the original refinement type system by Bengtson et al. [33]. 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 §4 by standard encodings [33].

3.1 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 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 [33,80], 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.

3.2 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:

$$\textit{encrypt} : \forall\alpha.\ \mathsf{PubKey}\langle\alpha\rangle \to \alpha \vee \mathsf{Un} \to \mathsf{Un}$$
$$\textit{decrypt} : \forall\alpha.\ \mathsf{Un} \to \mathsf{PrivKey}\langle\alpha\rangle \to \alpha \vee \mathsf{Un}$$

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 [33], which severely limits the expressiveness of the type system and prevents the analysis of a number of protocols, including this very simple example.

3.3 Type-checking the NSL Protocol

We first introduce the type definitions[3] 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 Private. Type 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 Private. Type $\mathsf{msg2}[x_B]$ 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 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 msg3 is a refinement type describing a nonce x_{nA} of type 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:

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

We use this notation to represent a disjoint union type with 3 constructors (tags): Msg1, Msg2, and Msg3. In the formal development and our implementation such types are desugared to binary sums, so the type $\mathsf{payload}[x]$ is desugared to $\mathsf{msg1} + (\mathsf{msg2} + \mathsf{msg3})$, Msg1 becomes inl, Msg2 becomes $\mathsf{inr} \circ \mathsf{inl}$, etc. 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 Table 1) and the code of the responder (A) abstract over the principal's identity and they are type-checked independently of each other.

Since library functions such as *encrypt*, *decrypt*, *send* and so on are polymorphic, they are instantiated with concrete types in the code (e.g., the encryptions in the initiator's

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

$\mathsf{init} = \lambda x_B : \mathsf{Un}.\ \lambda x_A : \mathsf{Un}.$
$\quad \lambda k_B : \mathsf{PrivKey}\langle\mathsf{payload}[x_B]\rangle.$
$\quad \lambda pk_A : \mathsf{PubKey}\langle\mathsf{payload}[x_A]\rangle.$
$\quad \lambda ch : \mathsf{Ch}(\mathsf{Un}).$
$\quad \mathsf{let}\ n_B = \mathit{mkPriv}()\ \mathsf{in}$
$\quad \mathsf{let}\ p_1 = (\mathsf{Msg1}\ (x_B, n_B))\ \mathsf{in}$
$\quad \mathsf{let}\ m_1 = \mathit{encrypt}\langle\mathsf{payload}[x_A]\rangle\ pk_A\ p_1\ \mathsf{in}$
$\quad \mathit{send}\langle\mathsf{Un}\rangle\ ch\ m_1;$
$\quad \mathsf{let}\ z = \mathit{recv}\langle\mathsf{Un}\rangle\ ch\ \mathsf{in}$
$\quad \mathsf{let}\ x = \mathit{decrypt}\langle\mathsf{payload}[x_B]\rangle\ k_B\ z\ \mathsf{in}$
$\quad \mathsf{case}\ x_1 = x : \mathsf{payload}[x_B] \vee \mathsf{Un}\ \mathsf{in}$
$\quad \mathsf{match}\ x_1\ \mathsf{with}\ \mathsf{Msg2}\ x_2 \Rightarrow$
$\quad \mathsf{let}\ (y_A, y_{nB}, y_{nA}) = x_2\ \mathsf{in}$
$\quad \mathsf{if}\ y_A = x_A\ \mathsf{then}$
$\quad \mathsf{if}\ y_{nB} = n_B\ \mathsf{then}$
$\quad \mathsf{assert}\ \mathsf{auth_r}(x_A, x_B, y_{nB}, y_{nA});$
$\quad \mathsf{assume}\ \mathsf{auth_i}(x_B, x_A, y_{nB}, y_{nA});$
$\quad \mathsf{let}\ p_3 = (\mathsf{Msg3}\ y_{nA})\ \mathsf{in}$
$\quad \mathsf{let}\ m_3 = \mathit{encrypt}\langle\mathsf{payload}[x_A]\rangle\ pk_A\ p_3\ \mathsf{in}$
$\quad \mathit{send}\langle\mathsf{Un}\rangle\ ch\ m_3$

$\mathsf{resp} = \lambda x_A : \mathsf{Un}.\ \lambda x_B : \mathsf{Un}.$
$\quad \lambda pk_B : \mathsf{PubKey}\langle\mathsf{payload}[x_B]\rangle.$
$\quad \lambda k_A : \mathsf{PrivKey}\langle\mathsf{payload}[x_A]\rangle.$
$\quad \lambda ch : \mathsf{Ch}(\mathsf{Un}).$
$\quad \mathsf{let}\ m_1 = \mathit{recv}\langle\mathsf{Un}\rangle\ ch\ \mathsf{in}$
$\quad \mathsf{let}\ x_1\ \mathsf{in}\ \mathit{decrypt}\langle\mathsf{payload}[x_A]\rangle\ k_A\ m_1$
$\quad \mathsf{case}\ y_1 = x_1 : \mathsf{payload}[x_A] \vee \mathsf{Un}\ \mathsf{in}$
$\quad \mathsf{match}\ y_1\ \mathsf{with}\ \mathsf{Msg1}\ z_1 \Rightarrow$
$\quad \mathsf{let}\ (y_B, x_{nB}) = z_1\ \mathsf{in}$
$\quad \mathsf{if}\ y_B = x_B\ \mathsf{then}$
$\quad \mathsf{let}\ n_A = \mathit{mkPriv}()\ \mathsf{in}$
$\quad \mathsf{assume}\ \mathsf{auth_r}(x_A, x_B, x_{nB}, n_A);$
$\quad \mathsf{let}\ p_2 = \mathsf{Msg2}(x_A, x_{nB}, n_A)\ \mathsf{in}$
$\quad \mathsf{let}\ m_2 = \mathit{encrypt}\langle\mathsf{payload}[x_B]\rangle\ pk_B\ p_2\ \mathsf{in}$
$\quad \mathit{send}\langle\mathsf{Un}\rangle\ ch\ m_2;$
$\quad \mathsf{let}\ m_3 = \mathit{recv}\langle\mathsf{Un}\rangle ch\ \mathsf{in}$
$\quad \mathsf{let}\ x_3 = \mathit{decrypt}\langle\mathsf{payload}[x_A]\rangle\ k_A\ m_3\ \mathsf{in}$
$\quad \mathsf{case}\ y_3 = x_3 : \mathsf{payload}[x_A] \vee \mathsf{Un}\ \mathsf{in}$
$\quad \mathsf{match}\ y_3\ \mathsf{with}\ \mathsf{Msg3}\ y_{nA} \Rightarrow$
$\quad \mathsf{if}\ y_{nA} = n_A\ \mathsf{then}$
$\quad \mathsf{assert}\ \mathsf{auth_i}(x_B, x_A, x_{nB}, n_A)$

Table 1: NSL Initiator Code and Responder Code

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 $\mathsf{case}\ x_1 = x : \mathsf{payload}[x_B] \vee \mathsf{Un}\ \mathsf{in}\ \ldots$.

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 $\{y_{nA} : \mathsf{Private} \mid \exists x_A, x_B, x_{nB}.\ \mathsf{auth_r}(x_A, x_B, x_{nB}, y_{nA}) \wedge \mathsf{auth_i}(x_B, x_A, x_{nB}, y_{nA})\} = \mathsf{msg3}$ 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}.

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

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 $= \lambda x : \mathsf{Un}.$
  let $k = mkPrivKey\langle \mathsf{payload}[x] \rangle$ () in $addToDB\ x\ k; getPubKey\langle \mathsf{payload}[x] \rangle\ k$

startNSL $= \lambda(role : \mathsf{Un})(x_A : \mathsf{Un})(x_B : \mathsf{Un})(c : \mathsf{Un}).$
  let $k_A = getFromDB\ x_A$ in let $pk_A = getPubKey\langle \mathsf{payload}[x_A] \rangle\ k_A$ in
  let $k_B = getFromDB\ x_B$ in let $pk_B = getPubKey\langle \mathsf{payload}[x_B] \rangle\ k_B$ in
  match $role$ with inl _ $\Rightarrow$ (init $x_A\ x_B\ k_A\ pk_B\ c$)
  | inr _ $\Rightarrow$ (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.

For simplicity of presentation, we do not give the attacker the capability to compromise participants, so the famous attack discovered by Lowe [97] is not possible even if we were to drop A's identity from the second message. As recently shown by Sattarzadeh and Fallah [113], the scenario in which the attacker has the ability to compromise participants can also be handled by our type system, but requires more complex type annotations. The two functions above express the capabilities of the attacker for verification purposes, and would not be exposed in a production setting. They can also be used 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, even when executed by an arbitrary attacker. In addition, the two nonces are given type Private and thus they are not revealed to the opponent.

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

The Refined Concurrent FPC (RCF) [33] is a simple programming language extending the Fixed Point Calculus [90] with refinement types [82, 112, 126] and concurrency [8, 105]. This core calculus is expressive enough to encode a considerable fragment of an ML-like programming language [33]. In this paper, we further increase the expressivity

of the calculus by adding intersection types [108], union types [107], parametric polymorphism [83, 109], and the novel ability to reason statically about type disjointness. 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. RCF$^{\forall}_{\wedge\vee}$ 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 §6).

Given a phrase of syntax ϕ, let $\phi\{M/x\}$ denote the substitution of each free occurrence of the variable x in ϕ with the value M. We use $\widetilde{\phi}$ to denote the sequence $\phi_1, \ldots, \phi_n$ for some n. A phrase is closed if it does not have free variables. We write $\mathit{free}(\phi)$ to denote the free names, variables and type variables in a phrase of syntax ϕ.

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 [33].

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

$x : T \to U$	dependent function type (scope of x is U)
$x : T * U$	dependent pair type (scope of x is U)
$T + U$	disjoint sum type
$\mu\alpha.\, T$	iso-recursive type (scope of α is T)
α	type variable
$\{x : T \mid C\}$	refinement type (scope of x is C)
$T \wedge U$	intersection type*
$T \vee U$	union type*
$\top$	top type*
$\forall\alpha.\, T$	polymorphic type* (scope of α is 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 (disjoint union). 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 [33, 82, 112, 126] to associate logical formulas to values. 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 the supertype of all the other types, and contains all well-typed values. The universal type $\forall\alpha.\, T$ [83, 109] 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 and $x \neq y$)
$\mathsf{match}\ M\ \mathsf{with}\ \mathsf{inl}\ x \Rightarrow A \mid \mathsf{inr}\ y \Rightarrow B$	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* (scope of x is A)
$(\nu a \updownarrow T)A$	restriction (scope of a is A)
$A \Rsh B$	fork off parallel expression
$a!M$	scope of M is 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 [33, 83, 107, 109]. 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 of the for construct (see §6). 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 §3. 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 the value M on channel a and returns the unit value $()$. Expression $a?$ blocks until some value 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 list 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 cryptographic 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 §6 and statically enforced by our type system from §5.

Surface syntax of RCF$^{\forall}_{\wedge\vee}$ authorization logic formulas

$C ::=$	authorization logic formula
$p(M)$	predicate symbol

$M = N$	equality
$C_1 \wedge C_2$	conjunction
$C_1 \vee C_2$	disjunction
$\neg C$	negation
$\forall x.\, C$	universal quantifier (scope of x is C)
$\exists x.\, C$	existential quantifier (scope of x is C)
$\exists a.\, C$	existential quantifier over names (scope of a is C)

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. A full account of the underlying authorization logic is provided in §A.5

5 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. [33] with union [107], intersection [108], and polymorphic types [83, 109]. 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 introduce the typing judgments, list all the typing rules and discuss the most important ones.

Typing judgments

$E \vdash \diamond$	E is well-formed
$E \vdash T$	type T is well-formed in E
$E \vdash C$	formula C is entailed from E
$E \vdash T :: k$	type T has kind k in E (where $k \in \{\mathsf{pub}, \mathsf{tnt}\}$)
$E \vdash T <: U$	type T is a subtype of type U in E
$E \vdash M : T$	value M has type T in E
$E \vdash A : T$	expression A has type T in E

5.1 Well-formed Environments 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 $\{C\}$).

Syntax of typing environments

$\mu ::=$	environment entry
α	type variable
$\alpha :: k$	kind-bounded type variable
$a \updownarrow T$	channel name
$x : T$	variable
$\{C\}$	formula*
$E ::= \mu_1, \ldots, \mu_n$	typing environment

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, and E is itself well-formed.

Domain of environment $dom(E)$; free bindings of environment entry $free(\mu)$

$dom(\alpha) = \{\alpha\}$	$free(x : T) = free(T)$
$dom(\alpha :: k) = \{\alpha\}$	$free(a \updownarrow T) = free(T)$
$dom(a \updownarrow T) = \{a\}$	$free(\{C\}) = free(C)$
$dom(x : T) = \{x\}$	$free(\mu) = \emptyset$, otherwise
$dom(\{C\}) = \emptyset$	
$dom(E_1, E_2) = dom(E_1) \cup dom(E_2)$	

Well-formed environments and types

(Env Empty)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env Entry)
$$\frac{E \vdash \diamond \quad free(\mu) \subseteq dom(E) \quad dom(\mu) \cap dom(E) = \emptyset}{E, \mu \vdash \diamond}$$

(Type)
$$\frac{E \vdash \diamond \qquad free(T) \subseteq dom(E)}{E \vdash T}$$

An important 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. If E binds a variable y to a refinement type $\{x : T \mid C\}$, we know that the formula $C\{y/x\}$ is entailed in the system and therefore $E \vdash C\{y/x\}$. In general, the idea is to inspect each of the type bindings in E and to extract the set of formulas occurring within refinement types. For intersection types we take the union of the formulas occurring in the two types,

while for union types we take their component-wise disjunction. Before discharging the proof obligation (using an SMT solver or FOL prover), we erase all type annotations from the formulas in the environment and the formula to be proven. As discussed in §6, this is crucial for the soundness of our type system. The function $[\![\phi]\!]$ removes all type annotations from a phrase of syntax ϕ. (The formal definition of this function is given in §A.2.)

Entailed formulas $E \vdash C$

(Derive)

$$\frac{E \vdash \diamond \quad \mathit{free}(C) \subseteq \mathit{dom}(E) \quad [\![\mathit{forms}(E)]\!] \models [\![C]\!]}{E \vdash C}$$

$\mathit{forms}(y : \{x : T \mid C\}) = \{C\{y/x\}\} \cup \mathit{forms}(y : T)$
$\mathit{forms}(y : T_1 \wedge T_2) = \mathit{forms}(y : T_1) \cup \mathit{forms}(y : T_2)$
$\mathit{forms}(y : T_1 \vee T_2) = \{C_1 \vee C_2 \mid C_1 \in \mathit{forms}(y : T_1), C_2 \in \mathit{forms}(y : T_2)\}$
$\mathit{forms}(\{C\}) = C$
$\mathit{forms}(E_1, E_2) = \mathit{forms}(E_1) \cup \mathit{forms}(E_2)$
$\mathit{forms}(E) = \emptyset$, otherwise

5.2 Subtyping and Kinding

The type system defines a *subtyping* relation on types and allows an expression of a subtype to be used in all contexts that require an expression of a supertype. This preorder provides more flexibility to the type system, since it allows more correct programs to be accepted as well-typed. For instance, 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., values of type $\{x : \mathsf{Un} \mid Ok(x)\}$ or $\mathsf{Un} * \mathsf{Un}$ could not be sent over the network.

Subtyping is commonly 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 flow to the attacker (e.g., 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 flow from the attacker (e.g., 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*. Kinding is defined as a separate judgment that contributes to the subtyping judgment via the (Sub Pub Tnt) rule. While kinding is the only part of the type system that is specific to security (everything

else is just very expressive but otherwise standard types for functional programming), kinding has a big impact on the rest of the type system (e.g., it equates many types that are not syntactically similar, breaking the standard progress property). We list all rules for kinding and subtyping, and then explain the most interesting ones below.

Kinding $E \vdash T :: k$

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

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

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

(Kind Unit)
$$\frac{}{E \vdash \diamond}$$

(Kind Sum)
$$\frac{E \vdash T :: k \quad E \vdash U :: k}{E \vdash (T + U) :: k}$$

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

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

(Kind Var)
$$\frac{E \vdash \diamond \quad (\alpha :: k) \in E}{E \vdash \alpha :: k}$$

(Kind Var False*)
$$\frac{\alpha \in dom(E) \quad E \vdash \mathsf{false}}{E \vdash \alpha :: k}$$

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

(Kind Top Tnt*)
$$\frac{E \vdash \diamond}{E \vdash \top :: \mathsf{tnt}}$$

(Kind Top Pub*)
$$\frac{E \vdash \mathsf{false}}{E \vdash \top :: \mathsf{pub}}$$

Notation: $\overline{\mathsf{pub}} = \mathsf{tnt}$ and $\overline{\mathsf{tnt}} = \mathsf{pub}$

Subtyping $E \vdash T <: U$

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

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

(Sub Pub Tnt)
$$\frac{E \vdash T :: \mathsf{pub} \quad E \vdash U :: \mathsf{tnt}}{E \vdash T <: U}$$

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

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

(Sub Pair)
$$\frac{E \vdash T <: T' \quad E, x : T' \vdash U <: U'}{E \vdash (x : T * U) <: (x : T' * U')}$$

(Sub Arrow)
$$\frac{E \vdash T' <: T \quad E, x : T' \vdash U <: U'}{E \vdash (x : T \to U) <: (x : T' \to U')}$$

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

(Sub Sum)
$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')}$$

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

Refinement Types. 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 $\forall x.\, \mathit{forms}(x : T) \Rightarrow C$ is entailed in the current typing environment, so by subtyping we can only add universally valid formulas. Similarly, a type $\{x : T \mid C\}$ is public when T is public, and tainted when T is tainted and $\forall x.\, \mathit{forms}(x : T) \Rightarrow C$ is entailed in the typing environment. The intuition is that $\{x : T \mid C\} <: T$ by (Sub Refine Left) and (Sub Refl*), so if additionally we have that that T is public ($T <: \mathsf{Un}$), then we can use transitivity of subtyping to conclude that $\{x : T \mid C\}$ is public as well ($\{x : T \mid C\} <: \mathsf{Un}$). Please note, however, that transitivity of subtyping is a property we later prove for the type system, not a subtyping rule.

Function Types. Function types are contravariant in their input and covariant in their output, i.e., in (Sub Arrow) we have that $T \to U$ is a subtype of $T' \to U'$ if T' is a subtype of T and U is a subtype of U'. Intuitively, this means that a function can be used in place of another function if the former is "more liberal" in the types it accepts and "more conservative" in the type it returns [96]. 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). Another way to look at this is if T is tainted (i.e., $\mathsf{Un} <: T$) and U is public (i.e., $U <: \mathsf{Un}$) then $T \to U$ is public, since by transitivity $(T \to U) <: (\mathsf{Un} \to \mathsf{Un}) <:> \mathsf{Un}$. Conversely, $T \to U$ is tainted if T is public and U is tainted.

Union and Intersection Types. The intersection type $T_1 \wedge T_2$ is a[4] 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 $T_1 \vee T_2$ is 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.

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 from (Sub Arrow), (Sub And LB 1), (Sub And LB 2), and (Sub And Greatest) 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 [64], 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 §5.4 and §11).

Polymorphic Types. Our 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. Bounded (or kind-bounded) quantification could easily be added to our language, but so far we found no compelling example in our application domain that would require bounded quantification (bounded quantifiers can also be encoded with normal quantifiers and intersection types [107]). Recent work by Dunfield [71] and others studies more precise subtyping rules for first-class polymorphic types.

Recursive Types. Our rule (Sub Pos Rec*) for subtyping recursive types can be tracked back to Val Tannen et al. [122]. It differs significantly from Cardelli's Amber rule [13, 57], which is more well-known and which is used by the original RCF [33]:

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

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 [33] – in particular proving the transitivity of subtyping in the presence of the Amber rule requires a very complicated inductive argument, which only works for "executable" environments, 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[5], this did not pose problems for us in practice[6], where most of the time only positive recursive types [103, 124] are used. In particular all first-order algebraic data types satisfy the positivity restriction, because only function types introduce negative positions. Moreover, this positivity restriction only affects subtyping, so programs involving negative occurrences of recursion variables that do not require subtyping can still be properly type-checked (e.g., we can still type-check the encodings of fixpoint combinators on expressions [33])

Ligatti et al. [95] have very recently proposed subtyping rules for iso-recursive types that are not only sound, but also complete with respect to type safety. The incompleteness of the Amber rule (Sub Rec) stems from its lack of considering unrolled types. We are not sure, however, if formalizing the transitivity of subtyping proof of Ligatti et al. would

[5] Let $T = \mu\alpha.\, \alpha \to \mathsf{pos}$ and $U = \mu\alpha.\, \alpha \to \mathsf{nat}$; if it wasn't for the positivity restriction, rule (Sub Pos Rec*) would allow us to show that T is a subtype of U. One would then expect that also the unfoldings of T and U are subtypes of each other, i.e., that $T \to \mathsf{pos}$ is a subtype of $U \to \mathsf{nat}$. By the contravariance of function types this is only the case if U is a subtype of T, so only when T and U are equivalent by subtyping, which is clearly not the case.

[6] Val Tannen et al. [123] give $\mu\alpha.\, \mathsf{int} * \{l : \alpha, m : \alpha \to \alpha\} <: \mu\beta.\, \mathsf{int} * \{l : \beta\}$ as an example subtyping that is intuitively valid, but which cannot be handled by rule (Sub Pos Rec*) because of the positivity restriction. Our type system has, however, no record types, and it cannot encode record types that satisfy subtyping in width. The only way we found to write a similar example in our system was to use union or intersection types inside the recursive type, as in $\mu\alpha.\, \mathsf{int} * (\alpha \wedge (\alpha \to \alpha)) <: \mu\beta.\, (\mathsf{int} * \beta)$, but this is by no means a commonly used idiom in practice.

be any easier than for the Amber rule.

5.3 Encoding Types Un and Private in RCF$^{\forall}_{\wedge\vee}$

In RCF [33] type Un is 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 equates many of the types in the system. For instance in RCF all the following types are equivalent by subtyping: 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[7]. A final observation is that, in 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

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

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[8]. 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 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 $\mathsf{Private}_{\mathsf{false}}$, which ensures that the attacker can only obtain private data under a logically inconsistent environment.

[7] For instance type $\top \to \top$ is neither public nor tainted, still $\lambda x : \top.\ x$ and $\lambda x : \mathsf{Un}.\ x$ are two syntactically equal values (after type erasure) that inhabit $\top \to \top$ and $\mathsf{Un} \to \mathsf{Un}$ respectively.

[8] The type $\mathsf{Private}_C$ is also very handy when reasoning about security despite compromise [19].

5.4 Typing Values and Expressions

The main judgments of the type system 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 judgments are mutually-inductively defined, since values and expressions are themselves defined by mutual induction (see §4). We first list the rules of each judgment, and then we explain the most interesting ones and, in particular, the ones that are new with respect to Bengtson et al. [33].

Typing values $E \vdash M : T$

(Val Var)
$$\frac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$$

(Val Subsum)
$$\frac{E \vdash M : T \quad E \vdash T <: T'}{E \vdash M : T'}$$

(Val Refine)
$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x : T \mid C\}}$$

(Val Lam)
$$\frac{E, x : T \vdash A : U}{E \vdash \lambda x : T.\, A : (x : T \to U)}$$

(Val TLam*)
$$\frac{E, \alpha \vdash A : T}{E \vdash \Lambda\alpha.\, A : \forall\alpha.\, T}$$

(Val Pair)
$$\frac{E \vdash M_1 : T_1 \quad E \vdash M_2 : T_2\{M_1/x\}}{E \vdash (M_1, M_2) : (x : T_1 * T_2)}$$

(Val And*)
$$\frac{E \vdash M : T \quad E \vdash M : U}{E \vdash M : T \wedge U}$$

(Val For 1*)
$$\frac{E \vdash M\{\widetilde{T}/\widetilde{\alpha}\} : V}{E \vdash \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}.\, M : V}$$

(Val For 2*)
$$\frac{E \vdash M\{\widetilde{U}/\widetilde{\alpha}\} : V}{E \vdash \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}.\, M : V}$$

(Val Fold)
$$\frac{E \vdash M : T\{\mu\alpha.\, T/\alpha\} \quad E \vdash \mu\alpha.\, T}{E \vdash \mathsf{fold}_{\mu\alpha.\, T}\ M : \mu\alpha.\, T}$$

(Val Unit)
$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}}$$

(Val Inl)
$$\frac{E \vdash M : T}{E \vdash \mathsf{inl}\ M : T + U}$$

(Val Inr)
$$\frac{E \vdash M : U}{E \vdash \mathsf{inr}\ M : T + U}$$

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 [64] the value restriction is crucial for the soundness of this introduction rule in the presence of side-effects (also see §11). 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 [107, 108] and Reynolds [110] 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 Un→Un, so by (Val And*) we can give it the desired intersection type.

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

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

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

(Exp Let)
$$\frac{E \vdash A : T \quad E, x : T \vdash B : U \quad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A\ \mathsf{in}\ B : U}$$

(Exp Assume)
$$\frac{E \vdash \diamond \quad free(C) \subseteq dom(E)}{E \vdash \mathsf{assume}\ C : \{\_ : \mathsf{unit} \mid C\}}$$

(Exp Res)
$$\frac{E, a \updownarrow T \vdash A : U \quad a \notin fn(U)}{E \vdash (\nu a \updownarrow T)A : U}$$

(Exp Send)
$$\frac{E \vdash M : T \quad (a \updownarrow T) \in E}{E \vdash a!M : \mathsf{unit}}$$

(Exp Recv)
$$\frac{E \vdash \diamond \quad (a \updownarrow T) \in E}{E \vdash a? : T}$$

(Exp Split)
$$\frac{\begin{array}{c} E \vdash M : (x : T * U) \\ E, x : T, y : U, \{(x, y) = M\} \vdash A : V \\ \{x, y\} \cap fv(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x, y) = M\ \mathsf{in}\ A : V}$$

(Exp Match)
$$\frac{\begin{array}{c} E \vdash M : T_1 + T_2 \\ E, x : T_1, \{\mathsf{inl}\ x = M\} \vdash A : U \quad x \notin fv(U) \\ E, y : T_2, \{\mathsf{inr}\ y = M\} \vdash B : U \quad y \notin fv(U) \end{array}}{E \vdash \mathsf{match}\ M\ \mathsf{with}\ \mathsf{inl}\ x \Rightarrow A \mid \mathsf{inr}\ y \Rightarrow B : U}$$

(Exp Unfold)
$$\frac{E \vdash M : \mu\alpha.\, T}{E \vdash \mathsf{unfold}_{\mu\alpha.\, T}\ M : T\{\mu\alpha.\, T/\alpha\}}$$

(Exp Fork)
$$\frac{E, \{\overline{A_2}\} \vdash A_1 : T \quad E, \{\overline{A_1}\} \vdash A_2 : U}{E \vdash (A_1 \Rsh A_2) : U}$$

The rule for type-checking $A_1 \Rsh A_2$, relies on an auxiliary function that extracts the top-level formulas from A_2 for type-checking A_1 and vice-versa. The function $\overline{A}$ returns the conjunction of each formula C occurring in a top-level $\mathsf{assume}\ C$ in A, with restricted names existentially quantified.

Formula extraction

$\overline{\mathsf{assume}\ C} = C$ $\qquad$ $\overline{A \Rsh B} = \overline{A} \wedge \overline{B}$ $\qquad$ $\overline{A} = \mathsf{true}$, otherwise

$\overline{(\nu a \updownarrow T)A} = \exists a.\, \overline{A}$ $\qquad$ $\overline{\mathsf{let}\ x = A\ \mathsf{in}\ B} = \overline{A}$

Union types are introduced by subtyping (T_1 is a subtype of $T_1 \vee T_2$ for any well-formed type T_2), and eliminated by a $\mathsf{case}\ x = M : T_1 \vee T_2\ \mathsf{in}\ A$ expression [107] using the (Exp Case*) rule.[9] Given a value M of type $T_1 \vee T_2$, we do not know in general 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 §3). 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 §7.2) and zero-knowledge (see §8). Additionally, if the equality test of the conditional succeeds then the types T_1 and T_2 are not disjoint. However, certain types such as Un and $\mathsf{Private}$ 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 [5] for verifying secrecy properties of nonce handshakes, but later disappeared in the more advanced type systems for authorization policies.

5.5 Non-disjointness of Types

We take this idea a lot further: we inductively define a relation of arity 4, which relates an environment, two types, and a logical formula. If $E \vdash T_1 \between T_2 \leadsto C$ holds and T_1 and T_2 have a common value in environment E, then E has to entail the condition C (i.e., $E \vdash C$). The base case of this relation is $E \vdash \mathsf{Private}_C \between \mathsf{Un} \leadsto C$, in particular $\emptyset \vdash \mathsf{Private} \between \mathsf{Un} \leadsto \mathsf{false}$. We call two types *provably disjoint* if $\emptyset \vdash T_1 \between T_2 \leadsto \mathsf{false}$, so $\mathsf{Private}$ and Un are provably disjoint. Intuitively, two provably disjoint types

[9] As pointed out by Dunfield and Pfenning [73] eliminating union types for expressions that are not in evaluation contexts is unsound in the presence of non-determinism (this is further discussed in §11).

have common values only in an inconsistent environment. 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 5 from §6.

Non-disjointness of types (*) $E \vdash T \between U \rightsquigarrow C$

(ND Private Un)
$$\frac{E \vdash \diamond \quad \mathit{free}(C) \subseteq \mathit{dom}(E)}{E \vdash \mathsf{Private}_C \between \mathsf{Un} \rightsquigarrow C}$$

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

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

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

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

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

(ND True)
$$\frac{E \vdash T_1 \quad E \vdash T_2}{E \vdash T_1 \between T_2 \rightsquigarrow \mathsf{true}}$$

(ND Sym)
$$\frac{E \vdash T_2 \between T_1 \rightsquigarrow C}{E \vdash T_1 \between T_2 \rightsquigarrow C}$$

(ND Conj)
$$\frac{E \vdash T \between U \rightsquigarrow C_1 \quad E \vdash T \between U \rightsquigarrow C_2}{E \vdash T \between U \rightsquigarrow C_1 \wedge C_2}$$

(ND Entails)
$$\frac{E \vdash T_1 \between T_2 \rightsquigarrow C \quad E, C \vdash C'}{E \vdash T_1 \between T_2 \rightsquigarrow C'}$$

(ND Forms And Type)
$$\frac{E \vdash T_1 \quad E \vdash T_2 \quad x \notin \mathit{free}(T_1, T_2)}{E \vdash T_1 \between T_2 \rightsquigarrow \exists x.\ \bigwedge \mathit{forms}(x : T_1 \wedge T_2)}$$

(ND Sub)
$$\frac{E \vdash T \between U \rightsquigarrow C \quad E \vdash U' <: U}{E \vdash T \between U' \rightsquigarrow C}$$

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 Un are provably disjoint. By (ND Sub) we need to show that $\mathsf{tree}\langle\mathsf{Private}\rangle$ and $\mathsf{tree}\langle\mathsf{Un}\rangle$, since $\mathsf{tree}\langle\mathsf{Un}\rangle$ and Un are equivalent by subtyping. 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).

Rule (ND True) gives the non-disjointness judgment a trivial base case which allows us to always infer the true formula. Rule (ND Sym) allows us to swap the two type arguments, since type disjointness is symmetric. Rule (ND Conj) allows us to take two instances of the non-disjointness judgment and combine their results using logical conjunction. Rule (ND Entails) allows us to weaken the formula in the non-disjointness judgment to any other formula that is entailed by it in the current typing environment. This rule together with (ND True) allow us to copy all formulas of the environment into the output formula, as done by the derived rule (ND Forms Env) below. Rule (ND Forms And Type) allows us to gather the formulas from the two types, conjoin them together, and require that there exists a term for which they all hold. Intuitively, if there exists a term that belongs to the intersection of the two types, that term will also satisfy the formulas gathered from both types. This rule allows us to derive rule (ND Forms Empty), which implies that $\{x : \mathsf{unit} \mid \mathsf{false}\}$ (the bottom type) overlaps other types only in an inconsistent environment. It also allows us to derive rule (ND Refine Exists) below, which implies that two refinement types that have contradicting formulas are disjoint. Rule (ND Sub) allows us to replace the types in the non-disjointness judgment by any of their subtypes. Together with rule (Sub Refine Left) this allows us to add refinement types in derived rule (ND Refine) (to drop refinement types if reading the rule backwards).

Derived non-disjointness rules

(ND Forms Env)

$$\frac{E \vdash T_1 \quad E \vdash T_2}{E \vdash T_1 \between T_2 \rightsquigarrow \bigwedge forms(E)}$$

(ND Forms Empty)

$$\frac{E \vdash T_1 \quad E \vdash T_2 \quad E, x : T_1 \vdash \mathsf{false}}{E \vdash T_1 \between T_2 \rightsquigarrow \mathsf{false}}$$

(ND Refine)

$$\frac{E \vdash T_1 \between T_2 \rightsquigarrow C \qquad E \vdash \{x : T_1 \mid C_1\}}{E \vdash \{x : T_1 \mid C_1\} \between T_2 \rightsquigarrow C}$$

(ND Refine Exists)

$$\frac{E \vdash T_1 \between T_2 \rightsquigarrow C \qquad E \vdash \{x : T_1 \mid C_1\} \quad E \vdash \{x : T_2 \mid C_2\}}{E \vdash \{x : T_1 \mid C_1\} \between \{x : T_2 \mid C_2\} \rightsquigarrow C \wedge \exists x.\, C_1 \wedge C_2}$$

6 Results of the Formalization

We have formalized the metatheory of RCF$^{\forall}_{\wedge\vee}$ in the Coq proof assistant [2]. We achieve this by defining Formal-RCF$^{\forall}_{\wedge\vee}$, a core calculus where terms are deeply embedded in Coq using a *locally nameless representation* [15, 84]: 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 [65]. 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-RCF$^{\forall}_{\wedge\vee}$ and RCF$^{\forall}_{\wedge\vee}$ is that in Formal-RCF$^{\forall}_{\wedge\vee}$ *all type annotations from values, expressions and formulas are erased*.

Type erasure for selected values and expressions

$$\|\lambda x : T.\, A\| = \mathsf{v\_lam}\ (\mathsf{close}_x\ \|A\|) \qquad \|\Lambda\alpha.\, A\| = \mathsf{v\_tlam}\ \|A\|$$

$$\|\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}.\, M\| = \|M\| \qquad \|M\langle T\rangle\| = \mathsf{e\_inst}\ \|M\|$$

$$\|\mathsf{case}\ x = M : T \vee U\ \mathsf{in}\ A\| = \mathsf{e\_let}\ \|M\|\ (\mathsf{close}_x\ \|A\|)$$

Notation: v_lam , e_inst , etc. are the constructors for the corresponding Formal-RCF$^{\forall}_{\wedge\vee}$ values and expressions. The function $\mathsf{close}_x\ e$ turns the free variable x into de Bruijn index 0, and shifts all existing de Bruijn indices up by one in Formal-RCF$^{\forall}_{\wedge\vee}$ expression e.

While this erasure process is straightforward (see §A.2), 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-RCF$^{\forall}_{\wedge\vee}$ very close to the original RCF [33], which is also extrinsically typed. In particular the operational se-

mantics of Formal-RCF$^{\forall}_{\wedge\vee}$[10] 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 ($e \to e'$). These two relations are formally defined in §A.4. 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-RCF$^{\forall}_{\wedge\vee}$.

As proposed by Aydemir et al. [15], 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. 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.

Two of the rules using cofinite quantification

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

Notation: The function $\mathsf{open}_x\ e$ turns the de Bruijn index 0 into free variable x, and shifts all existing de Bruijn indices down by one in Formal-RCF$^{\forall}_{\wedge\vee}$ expression e.

For instance the heating rule above allows heating below e_new binders by turning the de Bruijn index 0 in e_1 and e_2, bound by the top-level e_new constructs, into a sufficiently fresh free name a, using the open function. The name a has to avoid the finite set L, that is chosen arbitrarily when instantiating the rule. Similarly, the typing rule for v_tlam quantifies over a type variable name α avoiding an arbitrary finite set L.

We have proved that the typing judgments of RCF$^{\forall}_{\wedge\vee}$ are preserved by type erasure. This proof relies on standard renaming lemmas [15] for the Formal-RCF$^{\forall}_{\wedge\vee}$ judgments (we use $\Vdash$ to denote Formal-RCF$^{\forall}_{\wedge\vee}$ judgments).

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

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

The main result we have proved for the type system is that well-typed expressions are robustly safe. As in previous work [33], this property follows from the subject-reduction property of the type system. We also present a couple of important lemmas and theorems used in the proof.

The first such lemma states that in a logically inconsistent environment any well-formed expression e has any well-formed type T.

Lemma 3 (Inconsistent Environment)**.** *If* $E \Vdash \mathsf{false}$*,* $E \Vdash T$ *and* $free(e) \subseteq dom(E)$ *then* $E \Vdash e : T$*.*

The subtyping relation from §5.2 is transitive, and this result is used for proving various inversion lemmas for the typing judgments.

Lemma 4 (Transitivity of Subtyping)**.** *If* $E \Vdash T_1 <: T_2$ *and* $E \Vdash T_2 <: T_3$ *then* $E \Vdash T_1 <: T_3$*.*

As explained in §5.5, we show that if the judgment $E \Vdash T_1 \between T_2 \rightsquigarrow F$ is derivable and T_1 and T_2 have a common value v in environment E, then E has to entail the formula F. By choosing $F = \mathsf{f\_false}$ we obtain that two provably disjoint types have common values only in a logically inconsistent environment.

Theorem 5 (Non-disjoint)**.** *If* $E \Vdash T_1 \between T_2 \rightsquigarrow F$ *and* v *is a closed value so that* $E \Vdash v : T_1$ *and* $E \Vdash v : T_2$*, then* $E \Vdash F$*.*

The results above are all necessary for showing that in Formal-RCF$^{\forall}_{\wedge\vee}$ reduction preserves typing.

Theorem 6 (Reduction Preserves Types)**.** *If* $fv(e) = \emptyset$*,* $E \Vdash e : T$*, and* $e \to e'$ *then* $E \Vdash e' : T$*.*

We use this result to show that in Formal-RCF$^{\forall}_{\wedge\vee}$ all well-typed expressions are safe, and all expressions of type $\mathsf{t\_unit}$ are robustly safe. Intuitively, a program is safe if in all executions all active assertions are entailed by the active assumptions, and a program is robustly safe if it is safe when run in parallel with an arbitrary opponent [33, 80, 85, 86].

Definition 7 (Safety)**.** A closed expression e is safe if and only if, in all evaluations of e, all assertions succeed.

Theorem 8 (Safety)**.** *If* $\emptyset \Vdash e : T$ *then* e *is safe.*

Definition 9 (Opponent)**.** An opponent is an expression e that does not contain asserts, free variables or names.

Lemma 10 (Opponent Typability)**.** *If O is an opponent, $E \Vdash \diamond$, and E only contains variables and channels with type* $\mathsf{t\_unit}$ *then then $E \Vdash O : \mathsf{t\_unit}$.*

Robust safety follows immediately from safety and opponent typability.

Definition 11 (Robust Safety)**.** An expression e is robustly safe if the application $O\ e$ is safe for any opponent O.

Theorem 12 (Robust Safety for Formal-RCF$^{\forall}_{\wedge\vee}$)**.**
If $\emptyset \Vdash e : [\![\mathsf{t\_unit}]\!]$ then e is robustly safe.

Corollary 13 (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. [33] (which is, however, a property of contexts, not of values), we define a notion of *robustly private values*.

Definition 14 (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 12 (Robust Safety) that every value of type $\mathsf{Private}_C$ is robustly private in e unless C, for any well-typed expression e.

Theorem 15 (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 [2], together with most of the necessary lemmas. A notable exception is Theorem 2 (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. We remark that although the proofs of some helper lemmas are not assert-free, our formal proofs are done in greater detail than similar published paper proofs [23, 33, 34].

Our Coq formalization [21] totals more than 14kLOC,[11] out of which more than 1.5kLOC are just definitions. We used Ott [115] to generate a large part of these definitions from a 1kLOC long Ott specification, but for the more complex rules we often

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

needed to patch the output of Ott. We used LNgen [16] 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 relatively small problems in the paper proofs for the original RCF [33].[12] 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.

Lemma 16 (Replacing Tainted Bounds)**.**
If $E, x : T', E' \vdash U :: k$*, and* $E \vdash T$*, and* $E \vdash T' :: \mathsf{tnt}$ *then* $E, x : T, E' \vdash U :: k$*.*

Second, in the original RCF opponents can contain free names, so the proof of Theorem 12 (Robust Safety) used Theorem 8 (Safety) for a non-empty environment; however, safety was proved only for empty environments. We fixed this by not allowing the opponent to contain free names, since it can already generate names using the $(\nu a \updownarrow T)A$ expression and use such names for generating communication channels that are passed as arguments to the protocol code. Finally, the proof of the "Strengthening" lemma in the original RCF [33], and also in other refinement type systems for security [23], is wrong, and the status of the lemma in its original form is still unclear.

Claim 17 (Strengthening)**.**
If $E, \mu, E' \vdash \mathcal{J}$ *and* $dom(\mu) \cap (\mathit{free}(E') \cup \mathit{free}(\mathcal{J})) = \emptyset$ *and* $E, E' \vdash \mathit{forms}(\mu)$*, then* $E, E' \vdash \mathcal{J}$*.*

The proof is claimed to be by induction on the depth of the derivation of $E, \mu, E' \vdash \mathcal{J}$, however, in the (Exp Subsum) case the induction does not go through. In this case we know that $E, \mu, E' \vdash A : T$ and $E, \mu, E' \vdash T <: T'$, and need to show that $E, E' \vdash A : T'$. Additionally we know that $dom(\mu) \cap (\mathit{free}(E') \cup \mathit{free}(A, T')) = \emptyset$ and $E, E' \vdash \mathit{forms}(\mu)$. However, in order to apply the induction hypothesis for $E, \mu, E' \vdash A : T$ we would need as a freshness condition that $dom(\mu) \cap \mathit{free}(T) = \emptyset$, which we do not know since T and T' do not necessarily share the same free variables and names. The solution the RCF authors proposed is to weaken the claim of the lemma to only cover type variable and "anonymous" variable bindings [34] (which in our work we replaced by formula bindings). This is enough for the other results to go through, while avoiding the problems with the freshness condition.

[12] The journal version of the paper on RCF [34] incorporates these three fixes.

7 Implementation of Symbolic Cryptography

In contrast to process calculi for cryptographic protocols [7, 8], $\text{RCF}^{\forall}_{\wedge\vee}$ does not have any built-in construct to model cryptography. Cryptographic primitives are instead encoded using a dynamic sealing mechanism [106], 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 to the calculus or to the soundness proofs: one has just to find a well-typed encoding of the desired cryptographic primitive. In addition, Backes et al. [30] have recently 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. §7.1 overviews the dynamic sealing mechanism used by Bengtson et al. [33] to encode symbolic cryptography, while §7.2 and §7.3 show how our expressive type system can be used to improve this encoding and extend the class of supported protocols.

7.1 Dynamic Sealing

The notion of *dynamic sealing* was initially introduced by Morris [106] as a protection mechanism for programs. Later, Sumii and Pierce [117, 118] studied the semantics of dynamic sealing in a λ-calculus, observing a close correspondence with symmetric encryption. So the original spi-calculus [8], which baked in symmetric encryption, can essentially be seen as the pi-calculus [105] with dynamic sealing.

In RCF [33] 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 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 §8.1). 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 §3), where the type of the encrypted messages is neither public nor tainted.

7.2 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 [33]. 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 associated 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 in plain.

Symbolic implementation of signing-verification key pair

```
mkSigPair : ∀α. unit → SigKey⟨α⟩ ∗ VerKey⟨α⟩
mkSigPair = Λα. λu : unit.
    let (seal, unseal) = mkSeal ⟨α⟩ in
    let vk =  λx : Un. for β in ⊤; Un. λy : β.
                  if y = (unseal x) as z then z else failwith "verification failed"
    in (seal, vk)
```

The type $\mathsf{VerKey}\langle T\rangle$ of a verification key is defined as $\mathsf{Un} \to \big((y : \top \to \{z : T \mid y = z\}) \wedge (\mathsf{Un} \to \mathsf{Un})\big)$. 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 $y : \top \to \{z : T \mid y = z\}$ is used to type-check honest callers: the signed value y has any type (top type) and the value 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[13]. Hence, in contrast to [33], 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 [46]).

When type-checking the *mkSigPair* function above, the for causes the if to be type-checked twice using rule (Exp If*), the first time with y of type $\top$, and the second time with y of type Un. In case y has type $\top$ variable z has type $\top \wedge \alpha$, which is equivalent to T. Additionally, on the then branch we also know that $z = t$, which justifies the refinement type $\{z : T \mid t = z\}$ for the result z. In case y has type Un variable z has type $\mathsf{Un} \wedge \alpha$ which is a subtype of Un.

Finally, we present the typed interface of the functions to create and check signatures:

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

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 strictly 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 §8 (dishonest verifier cases).

[13] 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.

7.3 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 §3.2. In contrast to [33], the encryption key is always public, even if the type T of the encrypted message is not tainted.[14]

8 Encoding of Zero-knowledge

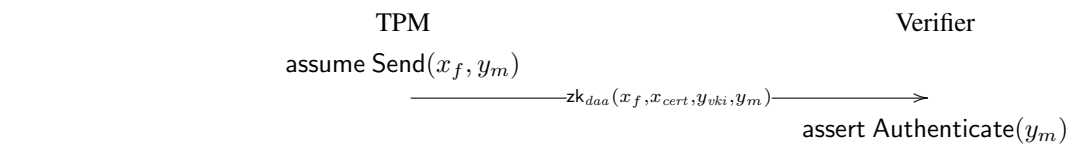
This section describes how we automatically generate the symbolic implementation of non-interactive zero-knowledge proofs, starting from a high-level specification. We recall that a zero-knowledge proof scheme can be seen as a proof system with the additional property that nothing is disclosed to the verifier about the *witnesses* used to create the proof other than the validity of the statement being proved. Intuitively, our symbolic 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* explicitly mentioned in the statement and, thus, revealed to the verifier.

8.1 Illustrative Example: Simplified DAA-sign

We are going to illustrate our technique on a simplified variant of the Direct Anonymous Attestation (DAA) protocol [46]. 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

[14] 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.

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.

8.2 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 [23, 29]. For a specification the user needs to provide: (1) variables representing the witnesses and public values of the proof, (2) a positive 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

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

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 is already known to the verifier

of the zero-knowledge proof. The payload message y_m is *returned* to the verifier of the proof.

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

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 [46, 98]), 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.

8.3 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

$\mathsf{create}_{daa} : T_{daa} \vee \mathsf{Un} \to \mathsf{Un}$ $\qquad$ $\mathsf{public}_{daa} : \mathsf{Un} \to \mathsf{Un}$

$\mathsf{verify}_{daa} : \mathsf{Un} \to ((y_{vki} : \mathsf{VerKey}\langle T_{vki}\rangle \to U_{daa}) \wedge (\mathsf{Un} \to \mathsf{Un}))$

where $T_{daa} = y_{vki} : \mathsf{VerKey}\langle T_{vki}\rangle * y_m : \mathsf{Un} * x_f : T_{vki} *\ x_{cert} : \mathsf{Un} * \{\mathsf{Send}(x_f, y_m)\}$

and $U_{daa} = \{y_m : \mathsf{Un} \mid \exists x_f, x_{cert}.\ \mathsf{OkTPM}(x_f) \wedge \mathsf{Send}(x_f, y_m)\}$

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 returned 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 $\mathit{check}\langle T_{vki}\rangle$ function (see §7.2). 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 public 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

```
verify_daa = λz : Un.
    for α in Un; VerKey⟨T_vki⟩. λy'_vki : α.
    let z' = (snd k_daa) z in                                   (1)
    case z'' = z' : Un ∨ 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 = check⟨T_vki⟩ 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 "$\mathsf{snd}\ k_{daa}$" 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 *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 our simple 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 called from 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 $\emptyset \vdash \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\} \between \mathsf{Un} \rightsquigarrow \mathsf{false}$ 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}.

9 Implementation

We have implemented a complete tool-chain for $\text{RCF}^{\forall}_{\wedge\vee}$: it includes a type-checker for the type system described in §5, the automatic code generator for zero-knowledge described in §8, an interpreter, a visual debugger, and a visualizer for (complete or partial) type derivations.

The type-checker supports an extended syntax with respect to the one from §4, 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 [66] or the E theorem prover [114] to discharge proof obligations. The type-checker supports the TPTP syntax for the generated proof obligations, which is a standard supported by most first-order theorem provers [119].

The type-checker produces a 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 reference protocol implementation. The type-checker also performs very limited type inference: in particular, it can infer the instantiation of some polymorphic functions from the type of the arguments. The user has to provide all the other typing annotations. We leave the development of a more powerful type inference algorithm as a future work.

The type-checker, the code generator for zero-knowledge, and the interpreter are command-line tools implemented in F#, while the graphical user interfaces 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 are available online [21].

10 Experiments

In this section we present several experiments using our type-checker and our code generator for symbolic zero-knowledge proofs. We also report on three independent cases studies, in which our tools were used "out of the box" in other research projects. The F5 type-checker performed well in these experiments, despite the lack of a considerable number of possible optimizations.

In a first experiment we used F5 to type-check a symbolic cryptographic library, including the sealing-based encodings of digital signatures (§7.2) and public-key en-

cryption (§7.3), as well as symbolic encodings for hashes and blind signatures. This cryptographic library encompasses 812 LOC and was type-checked by F5 in around 39 seconds on a Windows Vista laptop powered by an Intel Core2 Duo processor and 4GB of RAM. This 5 years old machine was used for all experiments presented in this section; we expect a newer machine would perform much better. For obtaining the results below we used the E equational prover [114] to discharge proof obligations.

Symbolic cryptographic library

File	Size (LOC)	Time (s)	Notes
base	69	1	bools, ints, options, lists, strings, references, etc.
blind	240	10	blind signatures
hash	79	2	hashes
pk-enc	55	9	public-key encryption (see §7.3)
seals	176	3	dynamic sealing (see §7.1)
sign	51	13	digital signatures (see §7.2)
old	142	1	variants for some of the above
Total	812	39	

In a second experiment we used F5 to type-check several non-cryptographic examples, including a suite of unit tests used to prevent regressions in F5 itself. F5 type-checked these 418 LOC in around 20 seconds.

Non-cryptographic examples

File	Size (LOC)	Time (s)	Notes
encoding-unions	113	9	union types using intersections [107]
fileprotocol	33	1	exercising assumes and asserts
for-and-side-effects	68	2	interaction between interaction and references
ternary-sum	36	1	encoding ternary sum with binary sum
RegressionSuite	168	7	unit tests
Total	418	20	

A third experiment involved basic cryptographic examples that do not involve zero-knowledge proofs. This included showing authentication for two variants of the NSL protocol (including the one from §3), and several other simple handshakes. This code is 602 LOC and takes around 169 seconds to type-check with F5.

Basic cryptographic examples (no zero-knowledge)

File	Size (LOC)	Time (s)	Notes
blind-sign-example	34	10	simple protocol with blind signatures
ciph-out-hash-in	34	8	authenticity using encryption and hashes
encrypt-then-sign	70	13	standard nonce handshake
nsl	187	37	NSL variants (see §3)
pk-enc-handshake	51	12	authenticity using public-key encryption
sign-example	71	25	authenticity using digital signatures
sign-hash	25	1	authenticity using signatures and hashes
sign-then-encrypt	130	63	standard nonce handshake and variants
Total	602	169	

Our final experiments involved type-checking three cryptographic protocols that use zero-knowledge proofs: the DAA-sign protocol [46], the simplified variant of DAA-sign from §8.1, and a protocol inspired by previous work on strengthening protocols despite compromised participants using zero-knowledge proofs [19]. For each of these case studies we used our generator to produce a symbolic implementation of the involved zero-knowledge proofs, type-checked this symbolic implementation against its generated interface, and type-checked the code of the protocol against the same interface. As the table below shows, more than 60% of the code is automatically generated. This generated code makes crucial use of the for and case constructs to provide rather sophisticated type annotations for each of the 4 considered cases (since both the prover and the verifier can be dishonest). The code that is written by hand is, however, much simpler (comparable in complexity to previous spi calculus models [19, 23]) and does not necessitate sophisticated type annotations.

Cryptographic protocols using zero-knowledge proofs

Dir	Size (LOC)	Generated	Time (s)	Notes
DAA-sign	317	193	43	DAA-sign protocol [46]
Simpl-DAA-sign	125	81	28	Simplified DAA-sign (see §8.1)
Encrypt-then-sign	184	125	62	inspired by [19]
Total	626	399	133	

The code for the experiments above is available part of the F5 tool-chain. The tool-chain was also used independently and "out of the box" in three other research projects, which we briefly describe below.

Eigner [75] used F5 to type-check the inalterability and eligibility properties of the Civitas remote e-voting protocol [60]. She uses dynamic seals to encode the sophisticated cryptographic schemes employed by Civitas: randomized public key encryption with re-encryption and plaintext equivalence tests, as well as encryption with distributed decryption. She also uses our code generator for obtaining sealing-based encodings of zero-knowledge proofs. Her development is around 2500 LOC and takes around 102 seconds to type-check with F5.[15]

Sattarzadeh and Fallah [113] have extended our model of the NSL protocol to compromised participants. They additionally use F5 to verify authentication for variants of the Woo-Lam, Splice/AS, and Otway-Ree protocols in the presence of compromised participants. Their development is 834 LOC and takes around 116 seconds to type-check with F5.[16]

Examples by Sattarzadeh and Fallah [113]

File	Size (LOC)	Time (s)	Notes
sym	34	1	sealing-based encoding of symmetric encryption
V-NSL	149	20	vulnerable NSL variant
T-NSL	163	56	secure NSL variant
T-WL	147	2	Woo-Lam variant from [86]
T-SPAS	175	35	modified Splice/AS protocol
T-OR	166	2	Otway-Ree's variant from [86]
Total	834	116	

Maffei and Pecina [100] used F5 to verify the correctness of the authorization decision for a series of simple privacy-aware proof-carrying authorization protocols. They also use our code generator for encoding zero-knowledge proofs symbolically. Their development is 478 LOC, mostly automatically generated with complex type annotations, and takes around 260 seconds to type-check with F5.[17]

[15] Code available at: `http://www.infsec.cs.uni-saarland.de/projects/F5/browser/trunk/Samples/Examples/Fabienne`

[16] Code available at: `http://ceit.aut.ac.ir/formalsecurity/tasp/f5/index.htm`

[17] Code available at: `http://www.infsec.cs.uni-saarland.de/projects/F5/browser/trunk/Samples/authorization`

11 Related Work on Unions and Intersections

The for construct for explicitly alternating type annotations was introduced by Pierce [107,108] as a generalization of an idea Reynolds [110] used in Forsythe for giving intersection types to annotated lambda abstractions of the form $\lambda x{:}\tau_1..\tau_n.\, e$. The for construct does not have a clear operational semantics. Compagnoni [61] 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 §5). As discovered by Davies and Pfenning [64] 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 [91]. Moreover, Davies and Pfenning [64] 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 [107] 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 [73] 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 [72], 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.

Zeilberger [127] tries to explain why phenomena such as the value and evaluation context restrictions can arise synthetically from a logical view of refinement typing.

12 Conclusion and Future Work

We have presented a new type system that combines refinement types with union types, intersection types, and polymorphic types. An important 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 [21].

The technique for syntactically reasoning about type disjointness we introduce in this work is general and should be of independent interest beyond analyzing security protocols. The only rule that is specific to the security setting is (ND Private Un); in a language without security kinding one can replace this base case with one that deems disjoint any two types that have different top-level type constructors (excluding refinement, union and intersection types). The way we inductively lift disjointness reasoning to many of the types in our system is completely general. Rules (ND Rec), (ND Pair), (ND Sum), (ND And), (ND Or), (ND Refine), (ND Sym), and (ND Sub) can easily be specialized to a completely standard lambda calculus:

Type disjointness for standard lambda calculus $E \vdash T \# U$

(Disj Base)
$$\frac{T \text{ and } U \text{ have different top-level type constructors}}{E \vdash T \# U}$$

(Disj Rec)
$$\frac{E \vdash (T\{\alpha/\mu\alpha.\,T\}) \# (U\{\beta/\mu\beta.\,U\})}{E \vdash (\mu\alpha.\,T) \# (\mu\beta.\,U)}$$

(Disj Pair 1)
$$\frac{E \vdash T_1 \# U_1}{E \vdash (T_1 * T_2) \# (U_1 * U_2)}$$

(Disj Pair 2)
$$\frac{E \vdash T_2 \# U_2}{E \vdash (T_1 * T_2) \# (U_1 * U_2)}$$

(Disj Sum)
$$\frac{E \vdash T_1 \# U_1 \quad E \vdash T_2 \# U_2}{E \vdash (T_1 + T_2) \# (U_1 + U_2)}$$

(Disj And 1)
$$\frac{E \vdash T_1 \# U}{E \vdash (T_1 \wedge T_2) \# U}$$

(Disj And 2)
$$\frac{E \vdash T_2 \# U}{E \vdash (T_1 \wedge T_2) \# U}$$

(Disj Or)
$$\frac{E \vdash T_1 \# U \quad E \vdash T_2 \# U}{E \vdash (T_1 \vee T_2) \# U}$$

(Disj Refine)
$$\frac{E \vdash T_1 \# T_2 \quad E \vdash \{x : T_1 \mid C_1\}}{E \vdash \{x : T_1 \mid C_1\} \# T_2}$$

(Disj Sym)
$$\frac{E \vdash T_2 \between T_1 \rightsquigarrow C}{E \vdash T_1 \between T_2 \rightsquigarrow C}$$

(Disj Sub)
$$\frac{E \vdash T \# U \quad E \vdash U' <: U}{E \vdash T \# U'}$$

It would be interesting to look for other contexts where syntactic reasoning about type disjointness is used, and try to see if our ideas are useful there. Reasoning about type disjointness is, for instance, used for preventing ambiguous overloading in the Fortress programming language [11].

In this work, we have focused on the automated *symbolic* implementation of a zero-knowledge proof system starting from a high-level specification of the statement. While this symbolic implementation is useful for verification and debugging purposes, the actual cryptographic implementation of the zero-knowledge proofs still has to be provided by the user. It would be interesting to integrate our work with the existing compilers for zero-knowledge proofs (e.g., those based on sigma-protocols [12, 31, 102] and on bilinear maps [9,28,89,101]), providing symbolic implementations that faithfully abstract the specific properties of these schemes (e.g., the malleability property of the Groth-Sahai proof system [89]) and are suitable to automated security verification.

The type-checker we implemented proved efficient 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 [93, 111]. The good news is that intersection and union types can be very useful when devising precise type inference algorithms [23, 94].

The original work of Abadi on secrecy by typing [4], as well as some of the follow up work by Abadi and Blanchet [5, 6] dealt with a strong notion of secrecy based on observational equivalence. Later type systems usually considered a weaker trace-based notion of secrecy that only prevents direct flows to the attacker [87]. For instance, the original type system for RCF [33] only considers weak secrecy for contexts (under the name of robust secrecy), while in this work we study the weak secrecy of values (under the name of robustly private values, see §6). In recent work Fournet et al. [81] use parametricity [41, 106, 117, 118] to prove strong secrecy by typing for a probabilistic variant of RCF. It would be interesting to apply this work to zero-knowledge proofs, since the zero-knowledge property is a strong secrecy property we do not currently capture.

More generally, it would be interesting to adapt some of the general techniques for establishing observational equivalences such as logical relations and bisimulations to $\mathrm{RCF}^{\forall}_{\wedge\vee}$. This would enable reasoning about privacy [22, 68, 100] and anonymity properties [25, 39, 46, 56] not only for abstract protocol models, but also for protocol implementations. It is often the case that such properties are achieved using zero-knowledge proofs.

Acknowledgments. We thank all those who provided helpful feedback at various stages

of this work: Joshua Dunfield, François Dupressoir, Cédric Fournet, Deepak Garg, Andy Gordon, Kim Pecina, Jan Schwinghammer, Pierre-yves Strub, Dominique Unruh, as well as the anonymous reviewers. Kudos to Thorsten Tarrach who implemented the original F5 prototype. Stefan Lorenz helped us with the cryptographic implementation of the DAA protocol. Cătălin Hriţcu was supported by a fellowship from Microsoft Research and the International Max Planck Research School for Computer Science. Matteo Maffei was supported by the German Research Foundation (DFG) through the Emmy Noether program and the Cluster of Excellence on Multimodal Computing and Interaction (MMCI), and by the German Federal Ministry of Education and Research (BMBF) through the Center for IT-Security, Privacy, and Accountability (CISPA).

A Formal-RCF$^{\forall}_{\wedge\vee}$ Calculus

A.1 Syntax

Formal-RCF$^{\forall}_{\wedge\vee}$ values, formulas and expressions

$c ::=$	name
name_b n	bound name (de Bruijn)
name_f a	free name (named)
$v, u ::=$	value
v_var_b n	bound variable (de Bruijn)
v_var_f x	free variable (named)
v_unit	unit
v_lam e	function
v_pair v_1 v_2	pair
v_inx h v	constructor
v_fold v	recursive value
v_tlam e	polymorphic value
$F ::=$	formula
f_pred P v	predicate symbol
f_eq v_1 v_2	equality
f_and F_1 F_2	conjunction
f_or F_1 F_2	disjunction
f_not F	negation
f_forall F	universal quantification
f_exists F	existential quantification
$e ::=$	expression

e_val v	value
e_app v_1 v_2	function application
e_inst v	instantiation
e_let e_1 e_2	let
e_first v	split first
e_second v e	split second
e_match v e_1 e_2	pattern matching
e_unfold v	use recursive value
e_if v_1 v_2 e_1 e_2	equality with type cast
e_new e	name restriction
e_fork e_1 e_2	fork off process
e_send c v	send v on channel c
e_recv c	receive on channel c
e_assume F	add formula F to log
e_assert F	formula F must hold

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

$T, U, V ::=$	types
t_unit	unit type
t_arrow T U	dependent function type
t_pair T U	dependent pair type
t_sum T U	disjoint sum type
t_rec T	iso-recursive type
t_var_b n	bound type variable (de Bruijn)
t_var_f α	free type variable (named)
t_refine T C	refinement type
t_and T U	intersection type
t_or T U	union type
t_top	top type
t_univ T	polymorphic type

Formal-RCF$^{\forall}_{\wedge\vee}$ syntax of environment entries

$\mu ::=$	environment entry
ee_tvar α	type variable
ee_kind α k	kind-bounded type variable
ee_var x T	variable x of type T

ee_chan $a\ T$	name a of type T
ee_ok F	assumed formula

Formal-RCF$^{\forall}_{\wedge\vee}$ syntax of variances

$\eta ::=$	variance
vnc_covar	covariant
vnc_contr	contravariant

A.2 Erasure from RCF$^{\forall}_{\wedge\vee}$ to Formal-RCF$^{\forall}_{\wedge\vee}$

Erasure for values

$\llbracket x \rrbracket = \mathsf{v\_var\_f}\ x$
$\llbracket () \rrbracket = \mathsf{v\_unit}$
$\llbracket \lambda x : T.\ A \rrbracket = \mathsf{v\_lam}\ (\mathsf{close}_x\ \llbracket A \rrbracket)$
$\llbracket (M, N) \rrbracket = \mathsf{v\_pair}\ \llbracket M \rrbracket\ \llbracket N \rrbracket$
$\llbracket h\ M \rrbracket = \mathsf{v\_inx}\ h\ \llbracket M \rrbracket$
$\llbracket \mathsf{fold}_{\mu\alpha.\,T}\ M \rrbracket = \mathsf{v\_fold}\ \llbracket M \rrbracket$
$\llbracket \Lambda\alpha.\ A \rrbracket = \mathsf{v\_tlam}\ \llbracket A \rrbracket$
$\llbracket \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}.\ M \rrbracket = \llbracket M \rrbracket$

Erasure for formulas

$\llbracket P(M) \rrbracket = \mathsf{f\_pred}\ P\ \llbracket M \rrbracket$
$\llbracket M = N \rrbracket = \mathsf{f\_eq}\ \llbracket M \rrbracket\ \llbracket N \rrbracket$
$\llbracket C_1 \wedge C_2 \rrbracket = \mathsf{f\_and}\ \llbracket C_1 \rrbracket\ \llbracket C_2 \rrbracket$
$\llbracket C_1 \vee C_2 \rrbracket = \mathsf{f\_or}\ \llbracket C_1 \rrbracket\ \llbracket C_2 \rrbracket$
$\llbracket \neg C \rrbracket = \mathsf{f\_not}\ \llbracket C \rrbracket$
$\llbracket \forall x.\ C \rrbracket = \mathsf{f\_forall}\ (\mathsf{close}_x\ \llbracket C \rrbracket)$
$\llbracket \exists x.\ C \rrbracket = \mathsf{f\_exists}\ (\mathsf{close}_x\ \llbracket C \rrbracket)$

Erasure for expressions

$\llbracket M\ N \rrbracket = \mathsf{e\_app}\ \llbracket M \rrbracket\ \llbracket N \rrbracket$
$\llbracket M\langle T \rangle \rrbracket = \mathsf{e\_inst}\ \llbracket M \rrbracket$
$\llbracket \mathsf{let}\ x = A\ \mathsf{in}\ B \rrbracket = \mathsf{e\_let}\ \llbracket A \rrbracket\ (\mathsf{close}_x\ \llbracket B \rrbracket)$

$\lVert \mathsf{let}\ (x, y) = M\ \mathsf{in}\ A \rVert =$
  $\mathsf{e\_let}\ (\mathsf{e\_first}\ \lVert M \rVert)$
    $(\mathsf{e\_second}\ \lVert M \rVert\ (\mathsf{close}_y\ (\mathsf{close}_x\ \lVert A \rVert)))$
$\lVert \mathsf{match}\ M\ \mathsf{with}\ \mathsf{inl}\ x \Rightarrow A \mid \mathsf{inr}\ y \Rightarrow B \rVert =$
  $\mathsf{e\_match}\ \lVert M \rVert\ (\mathsf{close}_x\ \lVert A \rVert)\ (\mathsf{close}_y\ \lVert B \rVert)$
$\lVert \mathsf{unfold}_{\mu\alpha.\,T}\ M \rVert = \mathsf{e\_unfold}\ \lVert M \rVert$
$\lVert \mathsf{case}\ x = M : T \vee U\ \mathsf{in}\ A \rVert = \mathsf{e\_let}\ \lVert M \rVert\ (\mathsf{close}_x\ \lVert A \rVert)$
$\lVert \mathsf{if}\ M = N\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B \rVert =$
  $\mathsf{e\_if}\ \lVert M \rVert\ \lVert N \rVert\ (\mathsf{close}_x\ \lVert A \rVert)\ \lVert B \rVert$
$\lVert (\nu a \updownarrow T) A \rVert = \mathsf{e\_new}\ (\mathsf{close}_a\ \lVert A \rVert)$
$\lVert A \Rsh B \rVert = \mathsf{e\_fork}\ \lVert A \rVert\ \lVert B \rVert$
$\lVert a!N \rVert = \mathsf{e\_send}\ a\ \lVert N \rVert$
$\lVert a? \rVert = \mathsf{e\_recv}\ a$
$\lVert \mathsf{assume}\ C \rVert = \mathsf{e\_assume}\ \lVert C \rVert$
$\lVert \mathsf{assert}\ C \rVert = \mathsf{e\_assert}\ \lVert C \rVert$

Erasure for types

$\lVert \mathsf{unit} \rVert = \mathsf{t\_unit}$
$\lVert x : T \to U \rVert = \mathsf{t\_arrow}\ \lVert T \rVert\ (\mathsf{close}_x\ \lVert U \rVert)$
$\lVert x : T * U \rVert = \mathsf{t\_pair}\ \lVert T \rVert\ (\mathsf{close}_x\ \lVert U \rVert)$
$\lVert T + U \rVert = \mathsf{t\_sum}\ \lVert T \rVert\ \lVert U \rVert$
$\lVert \mu\alpha.\, T \rVert = \mathsf{t\_rec}\ (\mathsf{close}_x\ \lVert T \rVert)$
$\lVert \alpha \rVert = \mathsf{t\_var\_f}\ \alpha$
$\lVert \{x : T \mid C\} \rVert = \mathsf{t\_refine}\ T\ (\mathsf{close}_x\ \lVert C \rVert)$
$\lVert T \wedge U \rVert = \mathsf{t\_and}\ \lVert T \rVert\ \lVert U \rVert$
$\lVert T \vee U \rVert = \mathsf{t\_or}\ \lVert T \rVert\ \lVert U \rVert$
$\lVert \top \rVert = \mathsf{t\_top}$
$\lVert \forall\alpha.\, T \rVert = \mathsf{t\_univ}\ (\mathsf{close}_\alpha\ \lVert T \rVert)$

Erasure for typing environments

$\lVert \alpha \rVert = \mathsf{ee\_tvar}\ \alpha$
$\lVert \alpha :: k \rVert = \mathsf{ee\_kind}\ \alpha\ k$
$\lVert a \updownarrow T \rVert = \mathsf{ee\_chan}\ a\ \lVert T \rVert$
$\lVert x : T \rVert = \mathsf{ee\_var}\ x\ \lVert T \rVert$
$\lVert \{C\} \rVert = \mathsf{ee\_ok}\ \lVert C \rVert$
$\lVert \mu_1, \ldots, \mu_n \rVert = \lVert \mu_1 \rVert, \ldots, \lVert \mu_n \rVert$

A.3 Local Closure

Locally closed values, formulas and expressions

$$\frac{}{\mathsf{lc}\ (\mathsf{v\_var\_f}\ x)} \quad \frac{}{\mathsf{lc}\ (\mathsf{v\_unit})} \quad \frac{\forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e)}{\mathsf{lc}\ (\mathsf{v\_lam}\ e)} \quad \frac{\mathsf{lc}\ e}{\mathsf{lc}\ (\mathsf{v\_tlam}\ e)} \quad \frac{\mathsf{lc}\ v_1 \quad \mathsf{lc}\ v_2}{\mathsf{lc}\ (\mathsf{v\_pair}\ v_1\ v_2)}$$

$$\frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{v\_inx}\ h\ v)} \quad \frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{v\_fold}\ v)} \quad \frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{f\_pred}\ P\ v)} \quad \frac{\mathsf{lc}\ v_1 \quad \mathsf{lc}\ v_2}{\mathsf{lc}\ (\mathsf{f\_eq}\ v_1\ v_2)} \quad \frac{\mathsf{lc}\ F_1 \quad \mathsf{lc}\ F_2}{\mathsf{lc}\ (\mathsf{f\_and}\ F_1\ F_2)}$$

$$\frac{\mathsf{lc}\ F_1 \quad \mathsf{lc}\ F_2}{\mathsf{lc}\ (\mathsf{f\_or}\ F_1\ F_2)} \quad \frac{\mathsf{lc}\ F}{\mathsf{lc}\ (\mathsf{f\_not}\ F)} \quad \frac{\forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ F)}{\mathsf{lc}\ (\mathsf{f\_forall}\ F)} \quad \frac{\forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ F)}{\mathsf{lc}\ (\mathsf{f\_exists}\ F)}$$

$$\frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{e\_val}\ v)} \quad \frac{\mathsf{lc}\ v_1 \quad \mathsf{lc}\ v_2}{\mathsf{lc}\ (\mathsf{e\_app}\ v_1\ v_2)} \quad \frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{e\_inst}\ v)} \quad \frac{\mathsf{lc}\ e_1 \quad \forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e_2)}{\mathsf{lc}\ (\mathsf{e\_let}\ e_1\ e_2)} \quad \frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{e\_first}\ v)}$$

$$\frac{\mathsf{lc}\ v \quad \forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e)}{\mathsf{lc}\ (\mathsf{e\_second}\ v\ e)} \quad \frac{\mathsf{lc}\ v \quad \forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e_1) \quad \forall y.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ y)}\ e_2)}{\mathsf{lc}\ (\mathsf{e\_match}\ v\ e_1\ e_2)}$$

$$\frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{e\_unfold}\ v)} \quad \frac{\mathsf{lc}\ F}{\mathsf{lc}\ (\mathsf{e\_assert}\ F)} \quad \frac{\mathsf{lc}\ F}{\mathsf{lc}\ (\mathsf{e\_assume}\ F)}$$

$$\frac{\mathsf{lc}\ v_1 \quad \mathsf{lc}\ v_2 \quad \forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e_1) \quad \mathsf{lc}\ e_2}{\mathsf{lc}\ (\mathsf{e\_if}\ v_1\ v_2\ e_1\ e_2)} \quad \frac{\forall a.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{name\_f}\ a)}\ e)}{\mathsf{lc}\ (\mathsf{e\_new}\ e)} \quad \frac{\mathsf{lc}\ e_1 \quad \mathsf{lc}\ e_2}{\mathsf{lc}\ (\mathsf{e\_fork}\ e_1\ e_2)}$$

$$\frac{}{\mathsf{lc}\ (\mathsf{name\_f}\ a)} \quad \frac{\mathsf{lc}\ c \quad \mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{e\_send}\ c\ v)} \quad \frac{\mathsf{lc}\ c}{\mathsf{lc}\ (\mathsf{e\_recv}\ c)}$$

Locally closed types

$$\frac{}{\mathsf{lc}\ (\mathsf{t\_unit})} \quad \frac{\mathsf{lc}\ T \quad \forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U)}{\mathsf{lc}\ (\mathsf{t\_arrow}\ T\ U)} \quad \frac{\mathsf{lc}\ T \quad \forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U)}{\mathsf{lc}\ (\mathsf{t\_pair}\ T\ U)}$$

$$\frac{\mathsf{lc}\ T \quad \mathsf{lc}\ U}{\mathsf{lc}\ (\mathsf{t\_sum}\ T\ U)} \quad \frac{\forall \alpha.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ T)}{\mathsf{lc}\ (\mathsf{t\_rec}\ T)} \quad \frac{}{\mathsf{lc}\ (\mathsf{t\_var\_f}\ \alpha)} \quad \frac{\mathsf{lc}\ T \quad \forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ F)}{\mathsf{lc}\ (\mathsf{t\_refine}\ T\ F)}$$

$$\frac{\mathsf{lc}\ T \quad \mathsf{lc}\ U}{\mathsf{lc}\ (\mathsf{t\_and}\ T\ U)} \quad \frac{\mathsf{lc}\ T \quad \mathsf{lc}\ U}{\mathsf{lc}\ (\mathsf{t\_or}\ T\ U)} \quad \frac{}{\mathsf{lc}\ (\mathsf{t\_top})} \quad \frac{\forall \alpha.\ \mathsf{lc}\ (\mathsf{open}_{\mathsf{t\_var\_f}\ \alpha}\ T)}{\mathsf{lc}\ (\mathsf{t\_univ}\ T)}$$

Locally closed environment entries

$$\frac{}{\mathsf{lc}\ (\mathsf{ee\_tvar}\ \alpha)} \quad \frac{}{\mathsf{lc}\ (\mathsf{ee\_kind}\ \alpha\ k)} \quad \frac{\mathsf{lc}\ T}{\mathsf{lc}\ (\mathsf{ee\_var}\ x\ T)} \quad \frac{\mathsf{lc}\ T}{\mathsf{lc}\ (\mathsf{ee\_chan}\ a\ T)} \quad \frac{\mathsf{lc}\ F}{\mathsf{lc}\ (\mathsf{ee\_ok}\ F)}$$

A.4 Operational Semantics

Heating Relation: $e_1 \Rrightarrow e_2$

(heat refl)
$$\frac{\mathsf{lc}\ e}{e \Rrightarrow e}$$

(heat trans)
$$\frac{e_1 \Rrightarrow e_2 \quad e_2 \Rrightarrow e_3}{e_1 \Rrightarrow e_3}$$

(heat let)
$$\frac{\mathsf{lc}\ (\mathsf{e\_let}\ e_1\ e_2) \quad e_1 \Rrightarrow e_1'}{\mathsf{e\_let}\ e_1\ e_2 \Rrightarrow \mathsf{e\_let}\ e_1'\ e_2}$$

(heat res)
$$\frac{\forall a \notin L.\ \mathsf{open}_{(\mathsf{name\_f}\ a)}\ e_1 \Rrightarrow \mathsf{open}_{(\mathsf{name\_f}\ a)}\ e_2}{\mathsf{e\_new}\ e_1 \Rrightarrow \mathsf{e\_new}\ e_2}$$

(heat fork 1)
$$\frac{e_1 \Rrightarrow e_1' \quad \mathsf{lc}\ e_2}{\mathsf{e\_fork}\ e_1\ e_2 \Rrightarrow \mathsf{e\_fork}\ e_1'\ e_2}$$

(heat fork 2)
$$\frac{e_2 \Rrightarrow e_2' \quad \mathsf{lc}\ e_1}{\mathsf{e\_fork}\ e_1\ e_2 \Rrightarrow \mathsf{e\_fork}\ e_1\ e_2'}$$

(heat fork unit 1)
$$\frac{\mathsf{lc}\ e}{\mathsf{e\_fork}\ (\mathsf{e\_val\ v\_unit})\ e \Rrightarrow e}$$

(heat fork unit 2)
$$\frac{\mathsf{lc}\ e}{e \Rrightarrow \mathsf{e\_fork}\ (\mathsf{e\_val\ v\_unit})\ e}$$

(heat msg unit)
$$\frac{\mathsf{lc}\ c \qquad \mathsf{lc}\ v}{\mathsf{e\_send}\ c\ v \Rrightarrow \mathsf{e\_fork}\ (\mathsf{e\_send}\ c\ v)\ (\mathsf{e\_val\ v\_unit})}$$

(heat assume unit)
$$\frac{\mathsf{lc}\ F}{\mathsf{e\_assume}\ F \Rrightarrow \mathsf{e\_fork}\ (\mathsf{e\_assume}\ F)\ (\mathsf{e\_val\ v\_unit})}$$

(heat res fork 1)
$$\frac{\mathsf{lc}\ e_1 \qquad \mathsf{lc}\ (\mathsf{e\_new}\ e_2)}{\mathsf{e\_fork}\ e_1\ (\mathsf{e\_new}\ e_2) \Rrightarrow \mathsf{e\_new}\ (\mathsf{e\_fork}\ e_1\ e_2)}$$

(heat res fork 2)
$$\frac{\mathsf{lc}\ (\mathsf{e\_new}\ e_1) \qquad \mathsf{lc}\ e_2}{\mathsf{e\_fork}\ (\mathsf{e\_new}\ e_1)\ e_2 \Rrightarrow \mathsf{e\_new}\ (\mathsf{e\_fork}\ e_1\ e_2)}$$

(heat res let)
$$\frac{\mathsf{lc}\ (\mathsf{e\_let}\ (\mathsf{e\_new}\ e_1)\ e_2)}{\mathsf{e\_let}\ (\mathsf{e\_new}\ e_1)\ e_2 \Rrightarrow \mathsf{e\_new}\ (\mathsf{e\_let}\ e_1\ e_2)}$$

(heat fork assoc 1)
$$\frac{\mathsf{lc}\ e_1 \qquad \mathsf{lc}\ e_2 \qquad \mathsf{lc}\ e_3}{\mathsf{e\_fork}\ (\mathsf{e\_fork}\ e_1\ e_2)\ e_3 \Rrightarrow \mathsf{e\_fork}\ e_1\ (\mathsf{e\_fork}\ e_2\ e_3)}$$

(heat fork assoc 2)
$$\frac{\mathsf{lc}\ e_1 \qquad \mathsf{lc}\ e_2 \qquad \mathsf{lc}\ e_3}{\mathsf{e\_fork}\ e_1\ (\mathsf{e\_fork}\ e_2\ e_3) \Rrightarrow \mathsf{e\_fork}\ (\mathsf{e\_fork}\ e_1\ e_2)\ e_3}$$

(heat fork comm)
$$\frac{\mathsf{lc}\ e_1 \qquad \mathsf{lc}\ e_2 \qquad \mathsf{lc}\ e_3}{\mathsf{e\_fork}\ (\mathsf{e\_fork}\ e_1\ e_2)\ e_3 \Rrightarrow \mathsf{e\_fork}\ (\mathsf{e\_fork}\ e_2\ e_1)\ e_3}$$

(heat fork let 1)

$$\frac{\mathsf{lc}\ (\mathsf{e\_let}\ (\mathsf{e\_fork}\ e_1\ e_2)\ e_3) \quad \mathsf{lc}\ e_1 \quad \mathsf{lc}\ e_2}{\mathsf{e\_let}\ (\mathsf{e\_fork}\ e_1\ e_2)\ e_3 \Rrightarrow \mathsf{e\_fork}\ e_1\ (\mathsf{e\_let}\ e_2\ e_3)}$$

(heat fork let 2)

$$\frac{\mathsf{lc}\ (\mathsf{e\_let}\ e_2\ e_3) \quad \mathsf{lc}\ e_1 \quad \mathsf{lc}\ e_2}{\mathsf{e\_fork}\ e_1\ (\mathsf{e\_let}\ e_2\ e_3) \Rrightarrow \mathsf{e\_let}\ (\mathsf{e\_fork}\ e_1\ e_2)\ e_3}$$

Reduction Relation: $e_1 \to e_2$

(red beta)

$$\frac{\mathsf{lc}\ (\mathsf{v\_lam}\ e) \quad \mathsf{lc}\ v}{\mathsf{e\_app}\ (\mathsf{v\_lam}\ e)\ v \to \mathsf{open}_v\ e}$$

(red inst)

$$\frac{\mathsf{lc}\ e}{\mathsf{e\_inst}\ (\mathsf{v\_tlam}\ e) \to e}$$

(red first)

$$\frac{\mathsf{lc}\ v_1 \qquad \mathsf{lc}\ v_2}{\mathsf{e\_first}\ (\mathsf{v\_pair}\ v_1\ v_2) \to \mathsf{e\_val}\ v_1}$$

(red second)

$$\frac{\mathsf{lc}\ (\mathsf{e\_second}\ (\mathsf{v\_pair}\ v_1\ v_2)\ e)}{\mathsf{e\_second}\ (\mathsf{v\_pair}\ v_1\ v_2)\ e \to \mathsf{e\_val}\ v_2}$$

(red match inl)

$$\frac{\mathsf{lc}\ (\mathsf{e\_match}\ (\mathsf{v\_inx\ inl}\ v)\ e_1\ e_2)}{\mathsf{e\_match}\ (\mathsf{v\_inx\ inl}\ v)\ e_1\ e_2 \to \mathsf{open}_v\ e_1}$$

(red match inr)

$$\frac{\mathsf{lc}\ (\mathsf{e\_match}\ (\mathsf{v\_inx\ inr}\ v)\ e_1\ e_2)}{\mathsf{e\_match}\ (\mathsf{v\_inx\ inr}\ v)\ e_1\ e_2 \to \mathsf{open}_v\ e_2}$$

(red unfold)

$$\frac{\mathsf{lc}\ v}{\mathsf{e\_unfold}\ (\mathsf{v\_fold}\ v) \to \mathsf{e\_val}\ v}$$

(red if true)

$$\frac{v_1 = v_2 \quad \mathsf{lc}\ (\mathsf{e\_if}\ v_1\ v_2\ e_1\ e_2)}{\mathsf{e\_if}\ v_1\ v_2\ e_1\ e_2 \to \mathsf{open}_v\ e_1}$$

(red if false)

$$\frac{v_1 \neq v_2 \quad \mathsf{lc}\ (\mathsf{e\_if}\ v_1\ v_2\ e_1\ e_2)}{\mathsf{e\_if}\ v_1\ v_2\ e_1\ e_2 \to e_2}$$

(red comm)

$$\frac{\mathsf{lc}\ c \qquad \mathsf{lc}\ v}{\mathsf{e\_fork}\ (\mathsf{e\_send}\ c\ v)\ (\mathsf{e\_recv}\ c) \to \mathsf{e\_val}\ v}$$

(red assert)

$$\frac{\mathsf{lc}\ F}{\mathsf{e\_assert}\ F \to \mathsf{e\_val\ v\_unit}}$$

(red let val)

$$\frac{\mathsf{lc}\ (\mathsf{e\_let}\ (\mathsf{e\_val}\ v)\ e)}{\mathsf{e\_let}\ (\mathsf{e\_val}\ v)\ e \to \mathsf{open}_v\ e}$$

(red let)

$$\frac{e_1 \to e_1' \quad \mathsf{lc}\ (\mathsf{e\_let}\ e_1\ e_2)}{\mathsf{e\_let}\ e_1\ e_2 \to \mathsf{e\_let}\ e_1'\ e_2}$$

(red res)

$$\frac{\forall a \notin L. \quad \mathsf{open}_{(\mathsf{name\_f}\ a)}\ e \to \mathsf{open}_{(\mathsf{name\_f}\ a)}\ e'}{\mathsf{e\_new}\ e \to \mathsf{e\_new}\ e'}$$

(red fork 1)

$$\frac{e_1 \to e_1' \qquad \mathsf{lc}\ e_2}{\mathsf{e\_fork}\ e_1\ e_2 \to \mathsf{e\_fork}\ e_1'\ e_2}$$

(red fork 2)

$$\frac{e_2 \to e_2' \qquad \mathsf{lc}\ e_1}{\mathsf{e\_fork}\ e_1\ e_2 \to \mathsf{e\_fork}\ e_1\ e_2'}$$

(red heat)

$$\frac{e_1 \Rrightarrow e_2 \quad e_2 \to e_3 \quad e_3 \Rrightarrow e_4}{e_1 \to e_4}$$

A.5 Properties of the Authorization Logic

Properties of Deducibility $S \models F$

(multiset)
$$\frac{(S_1 +\!+ S_2) \models F}{(S_2 +\!+ S_1) \models F}$$

(axiom)
$$\frac{\mathsf{lc}\ F}{(F :: \mathsf{nil}) \models F}$$

(mon)
$$\frac{\mathsf{lc}\ F' \quad S \models F}{(F' :: S) \models F}$$

(subst)
$$\frac{\mathsf{lc}\ v \quad S \models F}{S\{v/x\} \models F\{v/x\}}$$

(cut)
$$\frac{S \models F \quad (F :: S) \models F'}{S \models F'}$$

(and intro)
$$\frac{S \models F_1 \quad S \models F_2}{S \models (\mathsf{f\_and}\ F_1\ F_2)}$$

(and elim l)
$$\frac{S \models (\mathsf{f\_and}\ F_1\ F_2)}{S \models F_1}$$

(and elim r)
$$\frac{S \models (\mathsf{f\_and}\ F_1\ F_2)}{S \models F_2}$$

(or intro l)
$$\frac{S \models F_1 \quad \mathsf{lc}\ F_2}{S \models (\mathsf{f\_or}\ F_1\ F_2)}$$

(or intro r)
$$\frac{\mathsf{lc}\ F_1 \quad S \models F_2}{S \models (\mathsf{f\_or}\ F_1\ F_2)}$$

(or elim*)
$$\frac{S \models (\mathsf{f\_or}\ F_1\ F_2) \quad (F_1 :: S) \models F \quad (F_2 :: S) \models F}{S \models F}$$

(eq)
$$\frac{\mathsf{lc}\ v}{\mathsf{nil} \models (\mathsf{f\_eq}\ v\ v)}$$

(ineq)
$$\frac{\mathsf{lc}\ v_1 \quad \mathsf{lc}\ v_2 \quad v_1 \neq v_2 \quad fv(v_1, v_2) = \emptyset}{\mathsf{nil} \models (\mathsf{f\_not}\ (\mathsf{f\_eq}\ v_1\ v_2))}$$

(ineq inx)
$$\frac{\mathsf{lc}\ v \qquad \nexists v'.\ \mathsf{v\_inx}\ h\ v' = v \qquad fv(v) = \emptyset}{\mathsf{nil} \models (\mathsf{f\_forall}\ (\mathsf{f\_not}\ (\mathsf{f\_eq}\ (\mathsf{v\_inx}\ h\ (\mathsf{v\_var\_b}\ 0))\ v)))}$$

(ineq fold)
$$\frac{\mathsf{lc}\ v \qquad \nexists v'.\ \mathsf{v\_fold}\ v' = v \qquad fv(v) = \emptyset}{\mathsf{nil} \models (\mathsf{f\_forall}\ (\mathsf{f\_not}\ (\mathsf{f\_eq}\ (\mathsf{v\_fold}\ (\mathsf{v\_var\_b}\ 0))\ v)))}$$

(exists elim)
$$\frac{S \models (\mathsf{f\_exists}\ F) \quad \forall x \notin L \cup fv(F') \cup fv(S).\ ((\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ F) :: S) \models F'}{S \models F'}$$

(exists intro)
$$\frac{\mathsf{lc}\ v \quad S \models (\mathsf{open}_v\ F)}{S \models (\mathsf{f\_exists}\ F)}$$

(false*)
$$\frac{\mathsf{lc}\ F \quad S \models \mathsf{f\_false}}{S \models F}$$

(contra*)
$$\frac{S \models \mathsf{f\_not}\ F \quad S \models F}{S \models \mathsf{f\_false}}$$

(ineq exists*)
$$\frac{\mathsf{lc}\ (\mathsf{f\_exists}\ (\mathsf{f\_eq}\ v_1\ v_2)) \quad fv(v_1, v_2) = \emptyset \quad \nexists v.\ \mathsf{open}_v\ v_1 = \mathsf{open}_v\ v_2}{\mathsf{nil} \models (\mathsf{f\_not}\ (\mathsf{f\_exists}\ (\mathsf{f\_eq}\ v_1\ v_2)))}$$

A.6 Typing Judgments

Well-formed environments $E \Vdash \diamond$

(wfe empty)

$$\frac{}{\mathsf{nil} \Vdash \diamond}$$

(wfe entry)

$$\frac{\begin{array}{c}\mathsf{lc}\ \mu \quad E \Vdash \diamond \quad \mathsf{fv\_ee}\ \mu \subseteq \mathsf{dom\_v}\ E \\ \mathsf{fn\_ee}\ \mu \subseteq \mathsf{dom\_n}\ E \quad \mathsf{ftv\_ee}\ \mu \subseteq \mathsf{dom\_tv}\ E \\ (\mathsf{dom\_v\_ee}\ \mu) \cap (\mathsf{dom\_v}\ E) = \emptyset \\ (\mathsf{dom\_n\_ee}\ \mu) \cap (\mathsf{dom\_n}\ E) = \emptyset \\ (\mathsf{dom\_tv\_ee}\ \mu) \cap (\mathsf{dom\_tv}\ E) = \emptyset\end{array}}{\mu :: E \Vdash \diamond}$$

Well-formed types $E \Vdash T$

(wft type)

$$\frac{\mathsf{lc}\ T \quad E \Vdash \diamond \quad \mathsf{fv\_type}\ T \subseteq \mathsf{dom\_v}\ E \quad \mathsf{fn\_type}\ T \subseteq \mathsf{dom\_n}\ E \quad \mathsf{ftv\_type}\ T \subseteq \mathsf{dom\_tv}\ E}{E \Vdash T}$$

Entailed formula $E \Vdash F$

(entails derive)

$$\frac{E \Vdash \diamond \quad \mathsf{fv\_form}\ F \subseteq \mathsf{dom\_v}\ E \quad \mathsf{fn\_form}F \subseteq \mathsf{dom\_n}\ E \quad (\mathsf{forms\_env}\ E) \models F}{E \Vdash F}$$

Kinding $E \Vdash T :: k$

(kind var)

$$\frac{E \Vdash \diamond \quad (\mathsf{ee\_kind}\ \alpha\ k) \in E}{E \Vdash (\mathsf{t\_var\_f}\ \alpha) :: k}$$

(kind unit)

$$\frac{E \Vdash \diamond}{E \Vdash \mathsf{t\_unit} :: k}$$

(kind arrow)

$$\frac{E \Vdash T :: \overline{k} \quad \forall x \notin L.\ ((\mathsf{ee\_var}\ x\ T) :: E) \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U) :: k}{E \Vdash (\mathsf{t\_arrow}\ T\ U) :: k}$$

(kind pair)

$$\frac{E \Vdash T :: k \quad \forall x \notin L.\ ((\mathsf{ee\_var}\ x\ T) :: E) \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U) :: k}{E \Vdash (\mathsf{t\_pair}\ T\ U) :: k}$$

(kind sum)

$$\frac{E \Vdash T :: k \quad E \Vdash U :: k}{E \Vdash (\mathsf{t\_sum}\ T\ U) :: k}$$

(kind rec)

$$\frac{\forall \alpha \notin L.\ ((\mathsf{ee\_kind}\ \alpha\ k) :: E) \Vdash (\mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ T) :: k}{E \Vdash (\mathsf{t\_rec}\ T) :: k}$$

(kind refine pub)

$$\frac{E \Vdash (\mathsf{t\_refine}\ T\ F) \quad E \Vdash T :: \mathsf{pub}}{E \Vdash (\mathsf{t\_refine}\ T\ F) :: \mathsf{pub}}$$

(kind refine tnt)

$$\frac{E \Vdash T :: \mathsf{tnt} \quad \forall x \notin L.\ ((\mathsf{ee\_var}\ x\ T) :: E) \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ F)}{E \Vdash (\mathsf{t\_refine}\ T\ F) :: \mathsf{tnt}}$$

(kind var false)

$$\frac{\alpha \in \mathsf{dom\_tv}\ E \quad E \Vdash \mathsf{f\_false}}{E \Vdash (\mathsf{t\_var\_f}\ \alpha) :: k}$$

(kind top tnt)

$$\frac{E \Vdash \diamond}{E \Vdash \mathsf{t\_top} :: \mathsf{tnt}}$$

(kind top pub)

$$\frac{E \Vdash \mathsf{f\_false}}{E \Vdash \mathsf{t\_top} :: \mathsf{pub}}$$

(kind and pub 1)

$$\frac{E \Vdash T :: \mathsf{pub} \qquad E \Vdash U}{E \Vdash (\mathsf{t\_and}\ T\ U) :: \mathsf{pub}}$$

(kind and pub 2)

$$\frac{E \Vdash T \quad E \Vdash U :: \mathsf{pub}}{E \Vdash (\mathsf{t\_and}\ T\ U) :: \mathsf{pub}}$$

(kind and tnt)

$$\frac{E \Vdash T :: \mathsf{tnt} \quad E \Vdash U :: \mathsf{tnt}}{E \Vdash (\mathsf{t\_and}\ T\ U) :: \mathsf{tnt}}$$

(kind or pub)

$$\frac{E \Vdash T :: \mathsf{pub} \quad E \Vdash U :: \mathsf{pub}}{E \Vdash (\mathsf{t\_or}\ T\ U) :: \mathsf{pub}}$$

(kind or tnt 1)

$$\frac{E \Vdash T :: \mathsf{tnt} \quad E \Vdash U}{E \Vdash (\mathsf{t\_or}\ T\ U) :: \mathsf{tnt}}$$

(kind or tnt 2)

$$\frac{E \Vdash T \quad E \Vdash U :: \mathsf{tnt}}{E \Vdash (\mathsf{t\_or}\ T\ U) :: \mathsf{tnt}}$$

(kind univ)

$$\frac{\forall \alpha \notin L.\ ((\mathsf{ee\_tvar}\ \alpha) :: E) \Vdash (\mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ T) :: k}{E \Vdash (\mathsf{t\_univ}\ T) :: k}$$

Subtyping $E \Vdash T <: U$

(sub refl)

$$\frac{E \Vdash T}{E \Vdash T <: T}$$

(sub pub tnt)

$$\frac{E \Vdash T :: \mathsf{pub} \quad E \Vdash U :: \mathsf{tnt}}{E \Vdash T <: U}$$

(sub arrow)

$$\frac{E \Vdash T' <: T \quad \forall x \notin L.\ ((\mathsf{ee\_var}\ x\ T') :: E) \Vdash \mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U <: \mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U'}{E \Vdash (\mathsf{t\_arrow}\ T\ U) <: (\mathsf{t\_arrow}\ T'\ U')}$$

(sub pair)

$$\frac{E \Vdash T <: T' \quad \forall x \notin L.\ ((\mathsf{ee\_var}\ x\ T) :: E) \Vdash \mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U <: \mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U'}{E \Vdash (\mathsf{t\_pair}\ T\ U) <: (\mathsf{t\_pair}\ T'\ U')}$$

(sub sum)

$$\frac{E \Vdash T <: T' \quad E \Vdash U <: U'}{E \Vdash (\mathsf{t\_sum}\ T\ U) <: (\mathsf{t\_sum}\ T'\ U')}$$

(sub top)

$$\frac{E \Vdash T}{E \Vdash T <: \mathsf{t\_top}}$$

(sub refine left)

$$\frac{E \Vdash (\mathsf{t\_refine}\ T\ F) \quad E \Vdash T <: T'}{E \Vdash (\mathsf{t\_refine}\ T\ F) <: T'}$$

(sub refine right)

$$\frac{E \Vdash T <: T' \quad \forall x \notin L.\ ((\mathsf{ee\_var}\ e\ T) :: E) \Vdash \mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ F}{E \Vdash T <: (\mathsf{t\_refine}\ T'\ F)}$$

(sub and lb 1)
$$\frac{E \Vdash T <: T' \quad E \Vdash U}{E \Vdash (\mathsf{t\_and}\ T\ U) <: T'}$$

(sub and lb 2)
$$\frac{E \Vdash T \quad E \Vdash U <: T'}{E \Vdash (\mathsf{t\_and}\ T\ U) <: T'}$$

(sub and greatest)
$$\frac{E \Vdash T <: T_1 \quad E \Vdash T <: T_2}{E \Vdash T <: (\mathsf{t\_and}\ T_1\ T_2)}$$

(sub or least)
$$\frac{E \Vdash T_1 <: T \quad E \Vdash T_2 <: T}{E \Vdash (\mathsf{t\_or}\ T_1\ T_2) <: T}$$

(sub or ub 1)
$$\frac{E \Vdash T' <: T \quad E \Vdash U}{E \Vdash T' <: (\mathsf{t\_or}\ T\ U)}$$

(sub or ub 2)
$$\frac{E \Vdash T \quad E \Vdash T' <: U}{E \Vdash T' <: (\mathsf{t\_or}\ T\ U)}$$

(sub univ)
$$\frac{\forall \alpha \notin L.\ ((\mathsf{ee\_tvar}\ \alpha) :: E) \Vdash \mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ T <: \mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ U}{E \Vdash \mathsf{t\_univ}\ T <: \mathsf{t\_univ}\ U}$$

(sub pos rec)
$$\frac{\begin{array}{l}\forall \alpha \notin L. \quad (\mathsf{ee\_tvar}\ \alpha) :: E \Vdash \mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ T <: \mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ U \\ \qquad\quad \mathsf{has\_variance}\ \alpha\ \mathsf{vnc\_covar}\ \mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ T \\ \qquad\quad \mathsf{has\_variance}\ \alpha\ \mathsf{vnc\_covar}\ \mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ U\end{array}}{E \Vdash \mathsf{t\_rec}\ T <: \mathsf{t\_rec}\ U}$$

α **has variance** η **in** T $\quad (\mathsf{has\_variance}\ \alpha\ \eta\ T)$

(hv var eq)
$$\frac{}{\mathsf{has\_variance}\ \alpha\ \mathsf{vnc\_covar}\ (\mathsf{t\_var\_f}\ \alpha)}$$

(hv var neq)
$$\frac{\alpha \neq \beta}{\mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{t\_var\_f}\ \beta)}$$

(hv unit)
$$\frac{}{\mathsf{has\_variance}\ \alpha\ \eta\ \mathsf{t\_unit}}$$

(hv arrow)
$$\frac{\mathsf{has\_variance}\ \alpha\ (\mathsf{neg\_vnc}\ \eta)\ T \quad \forall x \notin L.\ \mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U)}{\mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{t\_arrow}\ T\ U)}$$

(hv pair)
$$\frac{\mathsf{has\_variance}\ \alpha\ \eta\ T \quad \forall x \notin L.\ \mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U)}{\mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{t\_pair}\ T\ U)}$$

(hv sum)
$$\frac{\mathsf{has\_variance}\ \alpha\ \eta\ T \quad \mathsf{has\_variance}\ \alpha\ \eta\ U}{\mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{t\_sum}\ T\ U)}$$

(hv rec)
$$\frac{\forall \beta \notin L.\ \mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{open}_{(\mathsf{t\_var\_f}\ \beta)}\ T)}{\mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{t\_rec}\ T)}$$

(hv refine)
$$\frac{\mathsf{lc}\ (\mathsf{t\_refine}\ T\ C) \quad \mathsf{has\_variance}\ \alpha\ \eta\ T}{\mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{t\_refine}\ T\ C)}$$

(hv and)
$$\frac{\mathsf{has\_variance}\ \alpha\ \eta\ T \quad \mathsf{has\_variance}\ \alpha\ \eta\ U}{\mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{t\_and}\ T\ U)}$$

(hv or)
$$\frac{\mathsf{has\_variance}\ \alpha\ \eta\ T \quad \mathsf{has\_variance}\ \alpha\ \eta\ U}{\mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{t\_or}\ T\ U)}$$

(hv top)
$$\frac{}{\mathsf{has\_variance}\ \alpha\ \eta\ \mathsf{t\_top}}$$

(hv univ)

$$\frac{\forall \beta \notin L.\ \mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{open}_{(\mathsf{t\_var\_f}\ \beta)}\ T)}{\mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{t\_univ}\ T)}$$

Typing values $E \Vdash v : T$

(tval var)

$$\frac{E \Vdash \diamond \quad (\mathsf{ee\_var}\ x\ T) \in E}{E \Vdash (\mathsf{v\_var\_f}\ x) : T}$$

(tval unit)

$$\frac{E \Vdash \diamond}{E \Vdash \mathsf{v\_unit} : \mathsf{t\_unit}}$$

(tval lam)

$$\frac{\forall x \notin L.\ ((\mathsf{ee\_var}\ x\ T) :: E) \Vdash \mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e : \mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U}{E \Vdash (\mathsf{v\_lam}\ e) : \mathsf{t\_arrow}\ T\ U}$$

(tval tlam)

$$\frac{\forall \alpha \notin L.\ ((\mathsf{ee\_tvar}\ \alpha) :: E) \Vdash e : \mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ T}{E \Vdash (\mathsf{v\_tlam}\ e) : (\mathsf{t\_univ}\ T)}$$

(tval pair)

$$\frac{E \Vdash v_1 : T_1 \quad E \Vdash v_2 : (\mathsf{open}_{v_1}\ T_2)}{E \Vdash (\mathsf{v\_pair}\ v_1\ v_2) : (\mathsf{t\_pair}\ T_1\ T_2)}$$

(tval inl)

$$\frac{E \Vdash v : T \quad E \Vdash U}{E \Vdash (\mathsf{v\_inx\ inl}\ v) : (\mathsf{t\_sum}\ T\ U)}$$

(tval inr)

$$\frac{E \Vdash v : U \quad E \Vdash T}{E \Vdash (\mathsf{v\_inx\ inr}\ v) : (\mathsf{t\_sum}\ T\ U)}$$

(tval fold)

$$\frac{E \Vdash v : (\mathsf{open}_{(\mathsf{t\_rec}\ T)}\ T) \quad E \Vdash (\mathsf{t\_rec}\ T)}{E \Vdash (\mathsf{v\_fold}\ v) : (\mathsf{t\_rec}\ T)}$$

(tval refine)

$$\frac{E \Vdash v : T \quad E \Vdash (\mathsf{open}_v\ F)}{E \Vdash v : (\mathsf{t\_refine}\ T\ F)}$$

(tval subsum)

$$\frac{E \Vdash v : T \quad E \Vdash T <: T'}{E \Vdash v : T'}$$

(tval and)

$$\frac{E \Vdash v : T \quad E \Vdash v : U}{E \Vdash v : (\mathsf{t\_and}\ T\ U)}$$

Typing expressions $E \Vdash e : T$

(texp val)

$$\frac{E \Vdash v : T}{E \Vdash (\mathsf{e\_val}\ v) : T}$$

(texp subsum)

$$\frac{E \Vdash e : T \quad E \Vdash T <: T'}{E \Vdash e : T'}$$

(texp appl)

$$\frac{E \Vdash v_1 : (\mathsf{t\_arrow}\ T\ U) \quad E \Vdash v_2 : T}{E \Vdash (\mathsf{e\_app}\ v_1\ v_2) : (\mathsf{open}_{v_2}\ U)}$$

(texp inst)

$$\frac{E \Vdash v : (\mathsf{t\_univ}\ U) \quad E \Vdash T}{E \Vdash \mathsf{e\_inst}\ v : (\mathsf{open}_T\ U)}$$

(texp first)

$$\frac{E \Vdash v : (\mathsf{t\_pair}\ T\ U)}{E \Vdash (\mathsf{e\_first}\ v) : \mathsf{t\_refine}\ T\ (\mathsf{f\_exists}\ (\mathsf{f\_eq}\ (\mathsf{v\_pair}\ (\mathsf{v\_var\_b}\ 1)\ (\mathsf{v\_var\_b}\ 0))\ v))}$$

(texp second)

$$\frac{\begin{array}{c} E \Vdash v : (\mathsf{t\_pair}\ T\ U) \\ \mu = (\mathsf{ee\_ok}\ (\mathsf{f\_eq}\ (\mathsf{v\_pair}\ (\mathsf{v\_var\_f}\ x)\ (\mathsf{v\_var\_f}\ y))\ v)) \\ E' = \mu :: (\mathsf{ee\_var}\ y\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U)) :: (\mathsf{ee\_var}\ x\ T) :: E \\ \forall x \neq y \notin L.\ E' \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ y)}\ e) : V \end{array}}{E \Vdash (\mathsf{e\_second}\ v\ e) : V}$$

(texp match)

$$\frac{\begin{array}{c} E \Vdash v : (\mathsf{t\_sum}\ T\ U) \\ \mu_1 = (\mathsf{ee\_ok}\ (\mathsf{f\_eq}\ \mathsf{v\_inx}\ \mathsf{inl}\ (\mathsf{v\_var\_f}\ x)\ v)) \\ \forall x \notin L.\ \mu_1 :: (\mathsf{ee\_var}\ x\ T) :: E \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e_1) : V \\ \mu_2 = (\mathsf{ee\_ok}\ (\mathsf{f\_eq}\ \mathsf{v\_inx}\ \mathsf{inr}\ (\mathsf{v\_var\_f}\ y)\ v)) \\ \forall y \notin L.\ \mu_2 :: (\mathsf{ee\_var}\ y\ T) :: E \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ y)}\ e_2) : V \end{array}}{E \Vdash (\mathsf{e\_match}\ v\ e_1\ e_2) : V}$$

(texp unfold)

$$\frac{E \Vdash v : (\mathsf{t\_rec}\ T)}{E \Vdash (\mathsf{e\_unfold}\ v) : (\mathsf{open}_{(\mathsf{t\_rec}\ T)}\ T)}$$

(texp if)

$$\frac{\begin{array}{c} E \Vdash v_1 : T_1 \quad E \Vdash v_2 : T_2 \quad E \Vdash T_1 \between T_2 \rightsquigarrow F \\ F' = (\mathsf{f\_and}\ (\mathsf{f\_and}\ (\mathsf{f\_eq}\ (\mathsf{v\_var\_f}\ x)\ v_1)\ (\mathsf{f\_eq}\ v_1\ v_2))\ F) \\ \forall x \notin L.\ (\mathsf{ee\_ok}\ F') :: (\mathsf{ee\_var}\ x\ (\mathsf{t\_and}\ T_1\ T_2)) :: E \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e_1) : U \end{array}}{E \Vdash (\mathsf{e\_if}\ v_1\ v_2\ e_1\ e_2) : U}$$

(texp assume)

$$\frac{E \Vdash \diamond \quad \mathsf{lc}\ F \quad \mathsf{fv\_form}\ F \subseteq \mathsf{dom\_v}\ E \quad \mathsf{fn\_form} F \subseteq \mathsf{dom\_n}\ E}{E \Vdash (\mathsf{e\_assume}\ F) : (\mathsf{t\_refine}\ \mathsf{t\_unit}\ F)}$$

(texp assert)

$$\frac{E \Vdash F}{E \Vdash (\mathsf{e\_assert}\ F) : \mathsf{t\_unit}}$$

(texp let)

$$\frac{E \Vdash e_1 : T_1 \quad \forall x \notin L.\ ((\mathsf{ee\_var}\ x\ T_1) :: E) \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e_2) : T_2}{E \Vdash \mathsf{e\_let}\ e_1\ e_2 : T_2}$$

(texp case)

$$\frac{\begin{array}{c} E \Vdash e_1 : (\mathsf{t\_or}\ T_1\ T_2) \\ \forall x \notin L.\ ((\mathsf{ee\_var}\ x\ T_1) :: E) \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e_2) : U \\ \forall x \notin L.\ ((\mathsf{ee\_var}\ x\ T_2) :: E) \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e_2) : U \end{array}}{E \Vdash \mathsf{e\_let}\ e_1\ e_2 : U}$$

(texp res)

$$\frac{\forall a \notin L \cup (\mathsf{fn\_type}\ U).\ (\mathsf{ee\_chan}\ a\ T) :: E \Vdash \mathsf{open}_{(\mathsf{name\_f}\ a)}\ e : U}{E \Vdash (\mathsf{e\_new}\ e) : U}$$

(texp send)

$$\frac{(\mathsf{ee\_chan}\ a\ T) \in E \quad E \Vdash v : T}{E \Vdash (\mathsf{e\_send}\ (\mathsf{name\_f}\ a)\ v) : \mathsf{t\_unit}}$$

(texp recv)

$$\frac{(\mathsf{ee\_chan}\ a\ T) \in E \quad E \Vdash \diamond}{E \Vdash (\mathsf{e\_recv}\ (\mathsf{name\_f}\ a)) : T}$$

(texp fork)

$$\frac{((\mathsf{ee\_ok}\ (\mathsf{extr}\ e_2)) :: E) \Vdash e_1 : T_1 \quad ((\mathsf{ee\_ok}\ (\mathsf{extr}\ e_1)) :: E) \Vdash e_2 : T_2}{E \Vdash (\mathsf{e\_fork}\ e_1\ e_2) : T_2}$$

Non-disjointness of types $E \Vdash T \between U \rightsquigarrow F$

(nd private un)

$$\frac{E \Vdash \mathsf{Private}_F}{E \Vdash [\![\mathsf{Private}_F]\!] \between \mathsf{t\_unit} \rightsquigarrow F}$$

(nd rec)

$$\frac{E \Vdash (\mathsf{open}_{(\mathsf{t\_rec}\ T)}\ T) \between (\mathsf{open}_{(\mathsf{t\_rec}\ U)}\ U) \rightsquigarrow F}{E \Vdash (\mathsf{t\_rec}\ T) \between (\mathsf{t\_rec}\ U) \rightsquigarrow F}$$

(nd pair)

$$\frac{E \Vdash T_1 \between U_1 \rightsquigarrow F_1 \quad E \Vdash T_2 \between U_2 \rightsquigarrow F_2}{E \Vdash (\mathsf{t\_pair}\ T_1\ T_2) \between (\mathsf{t\_pair}\ U_1\ U_2) \rightsquigarrow (\mathsf{f\_and}\ F_1\ F_2)}$$

(nd sum)

$$\frac{E \Vdash T_1 \between U_1 \rightsquigarrow F_1 \quad E \Vdash T_2 \between U_2 \rightsquigarrow F_2}{E \Vdash (\mathsf{t\_sum}\ T_1\ T_2) \between (\mathsf{t\_sum}\ U_1\ U_2) \rightsquigarrow (\mathsf{f\_or}\ F_1\ F_2)}$$

(nd and)

$$\frac{E \Vdash T_1 \between U \rightsquigarrow F_1 \quad E \Vdash T_2 \between U \rightsquigarrow F_2}{E \Vdash (\mathsf{t\_and}\ T_1\ T_2) \between U \rightsquigarrow (\mathsf{f\_and}\ F_1\ F_2)}$$

(nd or)

$$\frac{E \Vdash T_1 \between U \rightsquigarrow F_1 \quad E \Vdash T_2 \between U \rightsquigarrow F_2}{E \Vdash (\mathsf{t\_or}\ T_1\ T_2) \between U \rightsquigarrow (\mathsf{f\_or}\ F_1\ F_2)}$$

(nd true)

$$\frac{E \Vdash T_1 \quad E \Vdash T_2}{E \Vdash T_1 \between T_2 \rightsquigarrow \mathsf{f\_true}}$$

(nd sym)

$$\frac{E \Vdash T_2 \between T_1 \rightsquigarrow F}{E \Vdash T_1 \between T_2 \rightsquigarrow F}$$

(nd conj)

$$\frac{E \Vdash T \between U \rightsquigarrow C_1 \quad E \Vdash T \between U \rightsquigarrow C_2}{E \Vdash T \between U \rightsquigarrow \mathsf{f\_and}\ C_1\ C_2}$$

(nd entails)

$$\frac{E \Vdash T_1 \between T_2 \rightsquigarrow C \quad E, C \Vdash C'}{E \Vdash T_1 \between T_2 \rightsquigarrow C'}$$

(nd forms and type)

$$\frac{E \Vdash T_1 \quad E \Vdash T_2 \quad x \notin \mathit{free}(T_1, T_2)}{E \Vdash T_1 \between T_2 \rightsquigarrow \mathsf{f\_exists}\ (\mathsf{close}_x\ (\bigwedge \mathit{forms}(x : T_1 \wedge T_2)))}$$

(nd sub)

$$\frac{E \Vdash T \between U \rightsquigarrow C \quad E \Vdash U' <: U}{E \Vdash T \between U' \rightsquigarrow C}$$

B Technical Details of the Symbolic Encoding of Zero-knowledge Proofs in $\mathrm{RCF}^{\forall}_{\wedge\vee}$

We implement a zero-knowledge oracle in $\mathrm{RCF}^{\forall}_{\wedge\vee}$ as three public functions that share a secret seal. In order to create a zero-knowledge proof the first function seals the witnesses and public values provided by the caller all together and returns a sealed value representing the non-interactive zero-knowledge proof, which can be sent to the verifier. The verification function unseals the sealed values, and checks if they indeed satisfy the statement by performing the corresponding cryptographic and logical operations. If it succeeds the result of the verification function is a tuple containing the returned public values of the proof. The public values (both matched and returned ones) can also be obtained with the third function, without checking the validity of the proof.

B.1 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 [23, 29]. For a specification $\mathscr{S}$ the user needs to provide: (1) variables representing the witnesses and public values of the proof, (2) a positive 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.

Variables. We use variables to stand for the *witnesses* and public values of a zero-knowledge proof. The witnesses are (usually secret) values that are never revealed by the proof, and are represented by *witness variables*. On the other hand, the public values are revealed by the proof. For the purpose of typing, we further make a distinction between the public values that are checked for equality by the verifier – represented by *matched* public variables, and the ones that are obtained as the result of the verification – represented by *returned* public variables.

In the DAA example, the variables x_f and x_{cert} stand for witnesses ($sort_{daa}(x_f) = sort_{daa}(x_{cert}) = \mathsf{witness}$). 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 ($sort_{daa}(y_{vki}) = \mathsf{matched}$). The payload message y_m is returned to the verifier of the proof, so $sort_{daa}(y_m) = \mathsf{returned}$.

In the following we assume a function $sort_{\mathscr{S}}$ that for each variable x of specification $\mathscr{S}$ assigns: matched if the value of x is revealed by the proof and the verifier checks the

value of x for equality with a known value, returned if x has a public value obtained by the verifier after checking the proof, witness if the value of x is not revealed by the proof.

Statement. We assume that the statement conveyed by a zero-knowledge proof for specification $\mathscr{S}$ is a positive Boolean formula $S_{\mathscr{S}}$. Statements are formed using equalities between variables and $\text{RCF}^{\forall}_{\wedge\vee}$ functions applied to variables, as well as conjunctions and disjunctions of such basic statements.

Syntax of zero-knowledge statements

$S, R ::=$	statements
$x = f\langle\widetilde{T}\rangle\ x_1 \ldots x_n$	function application
$S_1 \wedge S$	conjunction
$S_1 \vee S_2$	disjunction

Intuitively, a statement is valid for a certain instantiation of the variables if after substituting all variables with the corresponding values and applying all $\text{RCF}^{\forall}_{\wedge\vee}$ functions to their arguments we obtain a valid Boolean formula. We assume that the $\text{RCF}^{\forall}_{\wedge\vee}$ functions occurring in the statement have deterministic behavior[18], i.e., when called twice with the same arguments they return the same value.

For example, the statement of the zero-knowledge proof in the DAA-signing protocol is $S_{daa} = (x_f = \mathsf{check}\langle T_{vki}\rangle\ y_{vki}\ x_{cert}\ x_f)$. This statement is valid for a certain instantiation if the check function returns the value of x_f when the values of y_{vki}, x_{cert}, and x_f are passed as arguments. 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.

Types. The user also needs to provide a type for all specified variables. In the following we assume a function $t_{\mathscr{S}}$ that assigns a type to each variable in specification $\mathscr{S}$. The DAA-sign protocol does not preserve the secrecy of the signed message, so $t_{daa}(y_m) = \mathsf{Un}$. On the other hand, the TPM identifier x_f is given a secret and untainted type: $t_{daa}(x_f) = T_{vki} = \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\}$. This ensures that x_f is not known to the attacker and that it is certified by the issuer (i.e., 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 give the certificate sort returned or matched, since the verifier could then always distinguish if two

[18] In order to model randomized functions one can take the random seed as an explicit argument.

consecutive requests come from the same user or not (as in the pseudonymous version of DAA). While we assume that if $sort_{\mathscr{S}}(x) = \mathsf{returned}$ or $sort_{\mathscr{S}}(x) = \mathsf{matched}$ then $t_{\mathscr{S}}(x)$ has kind public, the converse does not need to be true.

Promise. The user can additionally specify a *promise*: an arbitrary formula in the authorization logic that holds in the typing environment of the prover. If the statement is strong enough to identify the prover as a honest (type-checked) protocol participant[19], then the promise can be safely transmitted to the typing environment of the verifier. For a specification $\mathscr{S}$ we denote the promise by $P_{\mathscr{S}}$. In the DAA example we have that $P_{daa} = \mathsf{Send}(x_f, y_m)$, since this predicate holds true in the typing environment of a honest TPM.

B.2 Automatic Code Generation

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

B.3 Typed Interface

The interface generated for a specification $\mathscr{S}$ contains three functions[20] that share hidden state (a seal for values of type $\tau_{\mathscr{S}}$):

$$
\begin{array}{lcl}
\mathsf{create}_{\mathscr{S}} & : & \tau_{\mathscr{S}} \to \mathsf{Un} \\
\multicolumn{3}{l}{\quad \text{where } \tau_{\mathscr{S}} = \mathsf{t\_or\ Un}\ \sum_{x \in \mathsf{vars}_{\mathscr{S}}} x : t_{\mathscr{S}}(x).\{P_{\mathscr{S}}\}} \\
\mathsf{public}_{\mathscr{S}} & : & \mathsf{Un} \to \mathsf{Un} \\
\mathsf{verify}_{\mathscr{S}} & : & \mathsf{Un} \to (\mathsf{Un} \wedge \prod_{y \in \mathsf{matched}_{\mathscr{S}}} y : t_{\mathscr{S}}(y). \\
 & & \quad \sum_{z \in \mathsf{returned}_{\mathscr{S}}} z : t_{\mathscr{S}}(y).\{ \exists \widetilde{x}. \underset{\widetilde{x} = \mathsf{witness}_{\mathscr{S}}}{} F(S_{\mathscr{S}}, E) \wedge P_{\mathscr{S}} \}\ )
\end{array}
$$

The function $\mathsf{create}_{\mathscr{S}}$ is used to create zero-knowledge proofs for specification $\mathscr{S}$. 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 types provided by the user. Additionally, we check that the promise $P_{\mathscr{S}}$ provided by the user holds in the typing environment of the prover. The returned zero-knowledge proof is given type Un so that it

[19] Signature proofs of knowledge have this property [46, 98].

[20] We use $\sum_{x \in \mathsf{vars}_{\mathscr{S}}} x : t_{\mathscr{S}}(x).\{P_{\mathscr{S}}\}$ to denote the nested dependent pair type $x_1 : t_{\mathscr{S}}(x_1) * \ldots * x_n : t_{\mathscr{S}}(x_n) * \{P_{\mathscr{S}}\}$ where $\widetilde{x} = \mathsf{vars}_{\mathscr{S}}$, and $\prod_{y \in \mathsf{matched}_{\mathscr{S}}} y : t_{\mathscr{S}}(y).\ T$ to denote the dependent function type $y_1 : t_{\mathscr{S}}(y_1) \to \ldots \to y_m : t_{\mathscr{S}}(y_m)$, where $\widetilde{y} = \mathsf{matched}_{\mathscr{S}}$.

can be sent over the public network. For instance, in the DAA example we have that: $\tau_{daa} = \mathsf{Un} \vee ((y_{vki}{:}\mathsf{VerKey}\langle T_{vki}\rangle * y_m{:}\mathsf{Un} * x_f{:}T_{vki} * x_{cert}{:}\mathsf{Un}) * \{\mathsf{Send}(x_f, y_m)\})$, where $T_{vki} = \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\}$.

The function $\mathsf{public}_{\mathscr{S}}$ is used to read the public values of a zero-knowledge proof for $\mathscr{S}$, so it takes as input the sealed proof of type Un and returns the tuple of public values, also at type Un.

The function $\mathsf{verify}_{\mathscr{S}}$ is used for verifying zero-knowledge proofs. 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 values for the matched variables, which have the user-specified types. On successful verification, this function returns a tuple containing the values of the public variables, again with their respective types. The function guarantees that the formula $\exists\widetilde{x}.F(S_{\mathscr{S}}, E) \wedge P_{\mathscr{S}}$ holds, where the public variables are free and the witnesses are existentially quantified. The first conjunct, $F(S_{\mathscr{S}}, E)$, guarantees that if verification succeeds then the statement indeed holds, no matter what the origin of the proof is. Since the statement itself is not a formula in the logic (as it was for instance the case in [23]), we use a transformation function F that computes the formula conveyed by the statement. This transformation is straightforward: it extracts the formulas guaranteed by the dependently-typed cryptographic functions (the post-conditions) and combines them using the corresponding logical connectives of the authorization logic.

The formula conveyed by a statement $F(S, E)$

$$F(x = f\langle\widetilde{U}\rangle\ x_1 \ldots x_n, E) = \wedge_{C \in forms(x:T)} C\{\widetilde{x}/\widetilde{y}\}$$
$$\quad \text{if } f : \forall\widetilde{\alpha}.U \in E \text{ and } U\{\widetilde{U}/\widetilde{\alpha}\} = (\textstyle\prod\widetilde{y} : \widetilde{T}.\ T) \wedge \mathsf{Un}$$
$$F(x = f\langle\widetilde{U}\rangle\ x_1 \ldots x_n, E) = \mathsf{true}, \text{ otherwise}$$
$$F(S_1 \wedge S_2, E) = F(S_1, E) \wedge F(S_2, E)$$
$$F(S_1 \vee S_2, E) = F(S_1, E) \vee F(S_2, E)$$

If the prover is a protocol participant then the second conjunct $P_{\mathscr{S}}$ was already checked when creating the proof, and can be easily justified. However, the attacker can, at least in principle, also create valid zero-knowledge proofs for which the formula $P_{\mathscr{S}}$ does not hold. In order to justify its return type, the implementation of the verification function has in many cases to make sure that this is actually not the case, and the proof can only come from a protocol participant.

For instance, in the DAA example, we have that

$$F(S_{daa}, E_{std}) = F(x_f = \mathsf{check}\langle T_{vki}\rangle\ y_{vki}\ x_{cert}\ x_f, E_{std})$$

As explained in §7, we have that $E_{std} \vdash \mathsf{check}\langle T_{vki}\rangle : xvk : \mathsf{VerKey}\langle T_{vki}\rangle \to z : \mathsf{Un} \to x : (T_{vki} \vee \mathsf{Un}) \to \{y : T_{vki} \mid y = x\}$. So for the first conjunct after applying the corresponding substitutions we obtain the formula: $(x_f = x_f) \wedge \mathsf{OkTPM}(x_f)$. The predicate $\mathsf{OkTPM}(x_f)$ was obtained from the nested refinement type T_{vki}, according to the definition of *forms* from §5. Finally, after removing the trivial equality we obtain that:

$$F(S_{daa}, E_{std}) = \mathsf{OkTPM}(x_f)$$

B.4 Generated Implementation

The generated $\mathsf{mkZK}_{\mathscr{S}}$ function creates a fresh seal k of type $\tau_{\mathscr{S}} = \mathsf{Un} \vee \sum_{x \in \mathsf{vars}(S)} x : t_S(x).\{P_{\mathscr{S}}\}$. The union type is necessary since the values that are sealed can come from the attacker as well as from honest participants. The sealing function of the seal k is directly used to implement the creation of zero-knowledge proofs. The unsealing function is instead passed to two auxiliary functions $pub_{\mathscr{S}}$ and $ver_{\mathscr{S}}$ that return the function for extracting the public values and the zero-knowledge verification function, respectively.

$\mathsf{mkZK}_{\mathscr{S}} = \lambda x : \mathsf{unit}.$
  $\mathsf{let}\ k = mkSeal\langle \tau_{\mathscr{S}}\rangle\ ()\ \mathsf{in}$
  $\mathsf{let}\ (\_, k_{sealing}, k_{unsealing}) = k\ \mathsf{in}$
  $(k_{sealing}, ver_S\ k_{unsealing}, pub_S\ k_{unsealing})$

$$\begin{aligned} pub_S &: (\mathsf{Un} \to \tau_S) \to \mathsf{Un} \to \mathsf{Un} \\ ver_S &: (\mathsf{Un} \to \tau_S) \to \mathsf{Verify}_S \end{aligned}$$

The implementation of pub_S is very simple: since the zero-knowledge proof is just a sealed value, pub_S unseals it using the sealing function received as argument and returns all public and matched witnesses as a tuple $(\widetilde{yz})$. The secret witnesses $\widetilde{x}$ are simply discarded, and the validity of the statement is not checked.

$pub_S = \lambda k_{unsealing} : \mathsf{Un} \to \tau_S.\, \lambda z : \mathsf{Un}.$
  $\mathsf{let}\ z' = k_{unsealing}\ z\ \mathsf{in}$
  $\mathsf{case}\ z'' = z' : \mathsf{Un} \vee \sum_{x \in \mathsf{vars}(S)} x : t_S(x).\{P_S\}\ \mathsf{in}$
  $\mathsf{let}\ (\widetilde{yz}, \widetilde{x}) = z''\ \mathsf{in}\ (\widetilde{yz})$

The case construct is necessary since τ_S is a union type. In case z' has type Un then the declared return type Un is trivial to justify. In case z' has type $\sum_{x \in \mathsf{vars}(S)} x :$

$t_S(x).\{P_S\}$ we rely on the earlier assumption that all public and matched variables have a public type, in order to give the returned tuple $(\widetilde{y})$ type Un.

The type and the implementation of the ver_S function are more involved. The function inputs the unsealing function $k_{unsealing}$ of type $\mathsf{Un} \to \tau_S$, a candidate zero-knowledge proof z of type Un, and values for the matched variables. Since the type Verify_S contains an intersection type (Un is one of the branches and this makes the type Verify_S public) we use a for construct to introduce this intersection type. If the proof is verified by the attacker we can assume that for all $y' \in \mathsf{matched}(S)$ we have $y' : \mathsf{Un}$ 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 for all $y' \in \mathsf{matched}(S)$ we have $y' : t_S(y')$, and need to give the returned value type $\sum_{y \in \mathsf{returned}(S)} y : t_S(y).\{\exists_{\widetilde{x}=\mathsf{witness}(S)}\widetilde{x}.\ P_S \wedge F(S,E)\}$. Intuitively, the strong types of the matched values allow us to guarantee the strong types of the returned public values, as well as the two formulas P_S and $F(S,E)$.

The generated ver_S function performs the following five steps (the first three ones are the same as for the pub_S function): (1) it unseals z using $k_{unsealing}$ 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 matched witnesses $\widetilde{y}$, the public ones $\widetilde{z}$, and the secret ones $\widetilde{x}$; (4) it tests if the matched witnesses $\widetilde{y}$ are equal to the values $\widetilde{y'}$ received as arguments, and in case of success assigns the equal values to the variables $\widetilde{y''}$ – since $\widetilde{y''}$ have stronger types than $\widetilde{y}$ and $\widetilde{y'}$ we use these variables to stand for the matched witnesses in the following; (5) it tests if the statement is true by applying the functions in S and checking the results for equality with the corresponding witnesses. This last step (denoted by "$\exp(\text{prime}(S), \{\widetilde{y''}/\widetilde{y}\})$") is slightly complicated by the fact that the statement can contain disjunctions and is discussed in more detail below.

$ver_S = \lambda k_{unsealing} : \mathsf{Un} \to \tau_S.\ \lambda z : \mathsf{Un}.$
$\quad \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{\mathsf{Un}};\ \widetilde{t_S(y)}.$
$\quad \lambda y'_1 : \alpha_1. \ldots \lambda y'_n : \alpha_n.$
$\quad\quad (*1*)\ \ \mathsf{let}\ z' = k_{unsealing}\ z\ \mathsf{in}$
$\quad\quad (*2*)\ \ \mathsf{case}\ z'' = z' : \mathsf{Un} \vee \sum_{x \in \mathsf{vars}(S)} x : t_S(x).\{P_S\}\ \mathsf{in}$
$\quad\quad (*3*)\ \ \mathsf{let}\ (\widetilde{y}, \widetilde{z}, \widetilde{x}) = z''\ \mathsf{in}$
$\quad\quad (*4*)\ \ \mathsf{if}\ (\widetilde{y}) = (\widetilde{y'})\ \mathsf{as}\ (\widetilde{y''})\ \mathsf{then}$
$\quad\quad (*5*)\ \ \quad \text{“}\exp(\text{prime}(S), \{\widetilde{y''}/\widetilde{y}\})\text{”}$
$\quad\quad\quad\quad \mathsf{else}\ \mathit{failwith}\ \text{“variables do not match”}$

In order to convert a statement into the corresponding succession of tests, we first break the statement S into the corresponding atomic statements of the form $R = (x = f\langle\widetilde{T}\rangle\ x_1 \dots x_n)$. By slightly abusing notation, we denote this decomposition as $S\,[R_1, \dots, R_n]$. We then convert $S\,[R_1, \dots, R_n]$ into a decision tree. Decisions trees are defined by the following grammar:

$$D ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ x = f\langle\widetilde{T}\rangle\ x_1 \dots x_n\ \mathsf{then}\ D_1\ \mathsf{else}\ D_2$$

We implement this as a function called prime, that given a decomposed statement $S\,[R_1, \dots, R_n]$ produces its prime tree, i.e., an ordered and reduced decision tree; we refer the interested reader to [47, 116] for the details.

Finally, the decision tree prime$(S\,[R_1, \dots, R_n])$ is converted into an $\text{RCF}^{\forall}_{\wedge\vee}$ expression using a function called exp.

Converting Decision Trees to Expressions

$\exp(\mathsf{true}, \sigma) = (\sigma(z_1), \dots, \sigma(z_n))$, where $\widetilde{z} = \mathsf{returned}(S)$
$\exp(\mathsf{false}, \sigma) = \mathit{failwith}$ "statement not valid"
$\exp(\mathsf{if}\ x = f\langle\widetilde{T}\rangle\ x_1 \dots x_n\ \mathsf{then}\ D_1\ \mathsf{else}\ D_2, \sigma) =$
    $\mathsf{if}\ \sigma(x) = f\langle\widetilde{T}\rangle\ \sigma(x_1) \dots \sigma(x_n)\ \mathsf{as}\ y\ \mathsf{then}$
        $\exp(D_1, \sigma\{y/x\})\ \mathsf{else}\ \exp(D_2, \sigma)$
Note: Variables y, y_1, and y_2 are always freshly chosen.

Other than the decision tree, this function takes as argument a substitution σ that records which is the variable with the strongest type that corresponds to each witness. Initially this substitution is $\{\widetilde{y'}/\widetilde{y}\}$, i.e., it maps the matched variables $\widetilde{y}$ to the values $\widetilde{y'}$ taken as arguments (remember that since $\widetilde{y}$ and $\widetilde{y'}$ were tested for equality in the previous step and $\widetilde{y'}$ have the stronger types). After checking each atomic statement the conversion introduces new variables that stand for some of the witnesses and updates the substitution accordingly. The conversion works as follows. The leaves of the decision tree marked with true are converted into expressions that return the tuple $(\sigma(x_1), \dots, \sigma(x_n))$, i.e., a tuple containing the public witnesses with their strongest type. The leaves marked with false are converted into an expression that indicates a verification error. The inner nodes of the decision tree are converted into if statements. More precisely, a node "if $x = f\langle\widetilde{T}\rangle\ x_1 \dots x_n$ then D_1 else D_2" in the tree is converted into an application on the function $f\langle\widetilde{T}\rangle$ to the arguments $\sigma(x_1) \dots \sigma(x_n)$. The result is then checked for equality with $\sigma(x)$, using an if statement with an "as y" clause, where y is a fresh variable. In order to generate the tree corresponding to a successful check we recursively invoke exp

on D_1 and the substitution updating σ to match x to y. The else branch is generated by recursively calling $\exp(D_2, \sigma)$.

In the DAA example the decision tree has a very simple structure:
if $x_f = \mathsf{check}\langle T_{vki}\rangle\ y_{vki}\ x_{cert}\ x_f$ then true else false.

B.5 Checking the Generated Implementation

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.

In general, there are two situations in which type-checking the generated implementation fails. First, the types provided by the user for the the public witnesses are not public. In this case the implementation of pub_S cannot match its defined type $\mathsf{Un} \to \mathsf{Un}$. Second, the formula P_S is not justified by the statement and the types of the witnesses. In this case ver_S cannot match its defined type.

References

[1] IBM identity governance project. `http://www.zurich.ibm.com/security/idemix/`.

[2] The Coq proof assistant, 2009. Version 8.2.

[3] Microsoft U-Prove, Community Technology Preview R2, Feb. 2011. `http://www.microsoft.com/u-prove`.

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

[5] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 3(298):387–415, 2003.

[6] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.

[7] 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.

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

[9] M. Abe, G. Fuchsbauer, J. Groth, K. Haralambiev, and M. Ohkubo. Structure-preserving signatures and commitments to group elements. In *Advances in Cryptology - CRYPTO 2010*, pages 209–236. Springer-Verlag, 2010.

[10] M. Aizatulin, A. D. Gordon, and J. Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. Draft, 2011.

[11] E. E. Allen, J. Hilburn, S. Kilpatrick, V. Luchangco, S. Ryu, D. Chase, and G. L. Steele Jr. Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance. In *Proc. 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA 2011)*, pages 973–992. ACM Press, 2011.

[12] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols. In *Proc. 15th European Symposium on Research in Computer Security (ESORICS)*, pages 151–167. Springer-Verlag, 2010.

[13] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, 1993.

[14] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google Apps. In *Proc. of 6th ACM workshop on Formal methods in security engineering (FMSE '08)*, pages 1–10. ACM Press, 2008.

[15] 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.

[16] 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.

[17] M. Backes, A. Cortesi, R. Focardi, and M. Maffei. A calculus of challenges and responses. In *Proc. 5th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 101–116. ACM Press, 2007.

[18] M. Backes, A. Cortesi, and M. Maffei. Causality-based abstraction of multiplicity in cryptographic protocols. In *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF)*, pages 355–269. IEEE Computer Society Press, 2007.

[19] M. Backes, M. P. Grochulla, C. Hriţcu, and M. Maffei. Achieving security despite compromise using zero-knowledge. In *Proc. 22th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, July 2009.

[20] M. Backes, D. Hofheinz, and D. Unruh. CoSP: A general framework for computational soundness proofs. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS)*, pages 66–78, 2009.

[21] M. Backes, C. Hriţcu, and M. Maffei. Union and intersection types for secure protocol implementations. Formalization and implementation available at `http://www.infsec.cs.uni-saarland.de/projects/F5/`.

[22] M. Backes, C. Hriţcu, and M. Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *21th IEEE Symposium on Computer Security Foundations (CSF 2008)*, pages 195–209. IEEE Computer Society Press, June 2008.

[23] 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.

[24] M. Backes, S. Lorenz, M. Maffei, and K. Pecina. The CASPA tool: Causality-based abstraction for security protocol analysis. In *Proceedings of the 20th international conference on Computer Aided Verification (CAV'08)*, Lecture Notes in Computer Science, pages 419–422. Springer-Verlag, 2008.

[25] M. Backes, S. Lorenz, M. Maffei, and K. Pecina. Anonymous webs of trust. In *Proc. 10th Privacy Enhancing Technologies Symposium (PETS'10)*, volume 6205 of *Lecture Notes in Computer Science*, pages 130–148. Springer-Verlag, 2010.

[26] M. Backes, M. Maffei, and C. Hriţcu. Union and Intersection Types for Secure Protocol Implementations. In *Proc. Theory of Security and Applications (TOSCA)*, Lecture Notes in Computer Science, pages 1–28. Springer-Verlag, 2011.

[27] M. Backes, M. Maffei, and K. Pecina. A security API for distributed social networks. In *18th Annual Network & Distributed System Security Symposium (NDSS'11)*, pages 35–51. Internet Society, 2011.

[28] M. Backes, M. Maffei, and K. Pecina. Automated synthesis of privacy-preserving distributed applications. In *19th Annual Network & Distributed System Security Symposium (NDSS'12)*. The Internet Society, 2012.

[29] 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.

[30] 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.

[31] E. Bangerter, T. Briner, W. Henecka, S. Krenn, A.-R. Sadeghi, and T. Schneider. Automatic generation of sigma-protocols. In *6th European Workshop on Public Key Infrastructures, Services and Applications (EuroPKI 2009)*, pages 67–82, 2009.

[32] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 2004.

[33] 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.

Superseded by [34]. Long version appeared as MSR-TR-2008-118; November 2010 revision available at `http://research.microsoft.com/en-us/um/people/adg/Publications/MSR-TR-2008-118-SP2.pdf`.

[34] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(2):8:1–8:45, 2011.

[35] K. Bhargavan, R. Corin, C. Fournet, and E. Zălinescu. Cryptographically verified implementations for TLS. In *Proc. 15th ACM Conference on Computer and Communications Security (CCS)*, pages 459–468. ACM Press, 2008.

[36] 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.

[37] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1):5:1–5:61, 2008.

[38] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy*, pages 445–459. IEEE Computer Society, 2013.

[39] P. Bichsel, J. Camenisch, T. Groß, and V. Shoup. Anonymous credentials on a standard Java Card. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS)*, pages 600–610, 2009.

[40] 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.

[41] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, Feb.–Mar. 2008.

[42] 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.

[43] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[44] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Static analysis for the pi-calculus with applications to security. *Information and Computation*, 168(1):68–92, 2001.

[45] J. Borgström, A. D. Gordon, and R. Pucella. Roles, stacks, histories: A triple for hoare. *Journal of Functional Programming*, 21(02):159–207, 2011.

[46] E. F. 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.

[47] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[48] M. Bugliesi, S. Calzavara, F. Eigner, and M. Maffei. Logical foundations of secure resource management. In *Proc. 2nd Conference on Principles of Security and Trust (POST 2013)*, pages 105–125. Lecture Notes in Computer Science, 2011.

[49] M. Bugliesi, S. Calzavara, F. Eigner, and M. Maffei. Resource-aware authorization policies for statically typed cryptographic protocols. In *Proc. 24th IEEE Symposium on Computer Security Foundations (CSF)*, pages 83–98. IEEE Computer Society Press, 2011.

[50] M. Bugliesi, R. Focardi, and M. Maffei. Principles for entity authentication. In *Proceedings of 5th International Conference Perspectives of System Informatics (PSI 2003)*, volume 2890 of *Lecture Notes in Computer Science*, pages 294–307. Springer-Verlag, July 2003.

[51] M. Bugliesi, R. Focardi, and M. Maffei. Authenticity by tagging and typing. In *Proc. 2nd ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 1–12. ACM Press, 2004.

[52] M. Bugliesi, R. Focardi, and M. Maffei. Compositional analysis of authentication protocols. In *Proc. 13th European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 140–154. Springer-Verlag, 2004.

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

[54] M. Bugliesi, R. Focardi, and M. Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563–617, 2007.

[55] 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.

[56] J. Camenisch and E. V. Herreweghen. Design and implementation of the idemix anonymous credential system. In *Proc. 9th ACM Conference on Computer and Communications Security*, pages 21–30, 2002.

[57] L. Cardelli. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[58] I. Cervesato, A. D. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key Kerberos. *Information and Computation*, 206:402–424, February 2008.

[59] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. Technical report, CMU CyLab, October 2008.

[60] 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.

[61] A. B. Compagnoni. Subject reduction and minimal types for higher order subtyping. Technical Report ECS-LFCS-97-363, LFCS, University of Edinburgh, August 1997.

[62] R. Corin and F. A. Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In *Proc. 3rd International Symposium on Engineering Secure Software and Systems*, volume 6542 of *LNCS*, pages 58–72. Springer-Verlag, 2011.

[63] C. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference (CAV'08)*, Lecture Notes in Computer Science, pages 414–418. Springer-Verlag, 2008.

[64] R. Davies and F. Pfenning. Intersection types and computational effects. In *Proc. International Conference on Functional Programming (ICFP 2000)*, pages 198–208, 2000.

[65] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381 – 392, 1972.

[66] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of TACAS*, 2008.

[67] J. de Ruiter and E. Poll. Formal analysis of the emv protocol suite. In *Theory of Security and Applications (TOSCA 2011)*, volume 6993 of *Lecture Notes in Computer Science*, pages 113–129. Springer-Verlag, 2011.

[68] S. Delaune, S. Kremer, and M. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[69] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

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

[71] J. Dunfield. Greedy bidirectional polymorphism. In *ML Workshop (ML '09)*, pages 15–26, Aug. 2009.

[72] J. Dunfield. Untangling typechecking of intersections and unions. In *Workshop on Intersection Types and Related Systems (ITRS)*, July 2010.

[73] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Proc. 31th Symposium on Principles of Programming Languages (POPL '04)*, pages 281–292. ACM Press, 2004.

[74] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann. Guiding a general-purpose c verifier to prove cryptographic protocols. In *Proc. 24th IEEE Symposium on Computer Security Foundations (CSF)*, pages 3–17. IEEE Computer Society Press, 2011.

[75] F. Eigner. Type-based verification of electronic voting systems. Master's thesis, Saarland University, September 2009.

[76] F. Eigner and M. Maffei. Differential privacy by typing in security protocols. In *Proc. 26th IEEE Symposium on Computer Security Foundations (CSF)*, pages 272–286. IEEE Computer Society Press, 2013.

[77] D. Fisher. Millions of .Net Passport accounts put at risk. *eWeek*, May 2003. (Flaw detected by Muhammad Faisal Rauf Danka).

[78] R. Focardi and M. Maffei. *Types for Security Protocols*, volume 5, chapter 7. IOS Press, 2011.

[79] R. Focardi, M. Maffei, and F. Placella. Inferring authentication tags. In *WITS '05: Proceedings of the 5th Workshop on Issues in the theory of security*, pages 41–49. ACM Press, 2005.

[80] 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.

[81] C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *Proc. 18th ACM Conference on Computer and Communications Security (CCS)*, pages 341–350. ACM Press, 2011.

[82] T. Freeman and F. Pfenning. Refinement types for ML. In *In Programming Language Design and Implementation (PLDI'91)*, pages 268–277. ACM Press, 1991.

[83] J.-Y. Girard. The System F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.

[84] 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.

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

[86] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3-4):435–483, 2004.

[87] A. D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *Proc. 16th International Conference on Concurrency Theory (CONCUR)*, volume 3653, pages 186–201. Springer-Verlag, 2005.

[88] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *Proc. 6th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2005)*, pages 363–379. Springer-Verlag, 2005.

[89] J. Groth and A. Sahai. Efficient non-interactive proof systems for bilinear groups. In *Advances in Cryptology - EUROCRYPT 2008*, pages 415–432, 2008.

[90] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[91] 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`.

[92] C. Hriţcu. *Union, Intersection, and Refinement Types and Reasoning About Type Disjointness for Security Protocol Analysis*. PhD thesis, Saarland University, 2012.

[93] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *Proceedings of CAV*, pages 470–485, 2011.

[94] 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.

[95] J. Ligatti, M. Nachtigal, J. Blackburn, and I. Hernandez. Completely subtyping iso-recursive types. Technical report, Department of Computer Science and Engineering, University of South Florida, Oct. 2011.

[96] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16:1811–1841, 1994.

[97] 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.

[98] L. Lu, J. Han, L. Hu, J. Huai, Y. Liu, and L. M. Ni. Pseudo trust: Zero-knowledge based authentication in anonymous peer-to-peer protocols. In *Proc. 2007 IEEE International Parallel and Distributed Processing Symposium*, page 94. IEEE Computer Society Press, 2007.

[99] M. Maffei. Tags for multi-protocol authentication. In *Proc. 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SECCO '04)*, Electronic Notes on Theoretical Computer Science, pages 55–63. Elsevier Science Publishers Ltd., 2004.

[100] M. Maffei and K. Pecina. Privacy-aware proof-carrying authorization. Position Paper, PLAS 2011, Apr. 2011.

[101] M. Maffei, K. Pecina, and M. Reinert. Security and privacy by declarative design. In *Proc. 26th IEEE Symposium on Computer Security Foundations (CSF)*, pages 81–96. IEEE Computer Society Press, 2013.

[102] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash. In *Proc. 19th USENIX Security Symposium*, pages 193–206, 2010.

[103] 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.

[104] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[105] R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[106] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[107] B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

[108] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.

[109] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[110] 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.

[111] 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.

[112] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[113] B. Sattarzadeh and M. S. Fallah. Typing secure implementation of authentication protocols in environments with compromised principals. *Security and Communication Networks*, Nov. 2013.

[114] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[115] 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.

[116] G. Smolka and C. E. Brown. Introduction to computational logic. Lecture Notes, Saarland University, July 2008. Available at `http://www.ps.uni-saarland.de/courses/cl-ss08/script/icl.pdf`.

[117] E. Sumii and B. C. Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4):521–554, 2003.

[118] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375(1-3):169–192, 2007.

[119] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[120] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *Proc. 19th European Symposium on Programming (ESOP)*. Springer-Verlag, 2010.

[121] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bharagavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proc. 16th ACM SIGPLAN international conference on Functional programming*, pages 266–278. ACM Press, 2011.

[122] V. Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and explicit coercion (preliminary report). In *Proc. 4th IEEE Symposium on Logic in Computer Science (LICS)*, pages 112–129. IEEE Computer Society Press, 1989.

[123] V. Tannen, C. A. Gunter, and A. Scedrov. Denotational semantics for subtyping between recursive types. Technical Report MS-CIS-89-63, University of Pennsylvania, Department of Computer & Information Science, Nov. 1989.

[124] P. Urzyczyn. Positive recursive type assignment. In *Proc. 20th International Symposium Mathematical Foundations of Computer Science (MFCS'95)*, pages 382–391, 1995.

[125] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, 1996.

[126] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. 26th Symposium on Principles of Programming Languages (POPL '99)*, pages 214–227. ACM Press, 1999.

[127] N. Zeilberger. Refinement types and computational duality. In *Proc. 3rd Workshop on Programming Languages Meets Program Verification (PLPV)*, pages 15–26. ACM Press, 2008.