Saarland University

Faculty of Natural Sciences and Technology I

Department of Computer Science

# Union, Intersection, and Refinement Types and Reasoning About Type Disjointness for Security Protocol Analysis

Cătălin Hriţcu

**Dissertation**

zur Erlangung des Grades
des Doktors der Naturwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Eingereicht am: 14. November 2011
Saarbrücken

Thesis for obtaining the title of Doctor of Natural Sciences of the Faculties of Natural Sciences and Technology of Saarland University

Dissertation zur Erlangung des Grades des Doktors der Naturwissenschaften der Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes

Reporters / Berichterstatter:
Prof. Dr. Michael Backes (Saarland University and MPI-SWS)
Prof. Dr. Andrew D. Gordon (Microsoft Research and University of Edinburgh)
Dr. Matteo Maffei (Saarland University)

Examination Board / Prüfungsausschuss:
Prof. Dr. Gert Smolka (Saarland University)
Prof. Dr. Michael Backes (Saarland University and MPI-SWS)
Dr. Matteo Maffei (Saarland University)
Dr. Derek Dreyer (MPI-SWS)

Dean / Dekan:
Prof. Dr. Holger Hermanns

Date of the Colloquium / Tag des Kolloquiums:
January 10, 2012 / 10. Januar 2012

Version from March 24, 2012/ Textfassung vom 24. März 2012

# Abstract

In this thesis we present two new type systems for verifying the security of cryptographic protocol models expressed in a spi-calculus and, respectively, of protocol implementations expressed in a concurrent lambda calculus. The two type systems combine prior work on refinement types with union and intersection 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 cryptographic protocols. In particular, our type systems can statically analyze protocols that are based on zero-knowledge proofs, even in scenarios when certain protocol participants are compromised. The analysis is scalable and provides security proofs for an unbounded number of protocol executions. The two type systems come with mechanized proofs of correctness and efficient implementations.

# Kurzzusammenfassung

In dieser Arbeit werden zwei neue Typsysteme vorgestellt, mit denen die Sicherheit kryptographischer Protokolle, modelliert in einem spi-Kalkül, und Protokollimplementierungen, beschrieben in einem nebenläufigen Lambdakalkül, verifiziert werden kann. Die beiden Typsysteme verbinden vorausgehende Arbeiten zu Verfeinerungstypen mit disjunktiven und konjunktiven Typen, und ermöglichen außerdem, statisch zu folgern, dass zwei Typen disjunkt sind. Die Ausdrucksstärke der Systeme erlaubt die Analyse wichtiger Klassen von Protokollen, die bisher nicht durch typbasierte Protokollanalysen behandelt werden konnten. Insbesondere ist mit den vorgestellten Typsystemen auch die statische Analyse von Protokollen möglich, die auf Zero-Knowledge-Beweisen basieren, selbst unter der Annahme, dass einige Protokollteilnehmer korrumpiert sind. Die Analysetechnik skaliert und erlaubt Sicherheitsbeweise für eine unbeschränkte Anzahl von Protokollausführungen. Die beiden Typsysteme sind formal korrekt bewiesen und effizient implementiert.

# Acknowledgements

I consider myself extraordinarily fortunate, being able to work with three distinguished academic advisors: Michael Backes, Andy Gordon, and Matteo Maffei. I am profoundly indebted to them for the very many things they have taught me. Michael provided guidance and support, promoted a friendly and relaxed atmosphere in his group, and gave me incredible freedom to find and pursue my own way in research. I am grateful to Andy for his infinite kindness, for taking the time for many in-depth discussions about research and life in general, and for the wonderful time I had during my visits in Cambridge. Matteo spent a great amount of time working together with me, and helped me countless times with his wonderful insights, with endless patience and contagious enthusiasm. Grazie mille, Matteo!

Gert Smolka gave me invaluable advice on many occasions. I am particularly grateful for his most inspiring introductions to logic and the semantics of programming languages, and for his suggestion that I work with Jan Schwinghammer in the spring of 2006. The year I worked with Jan reshaped my view of theory and research, and was crucial in my decision to do a PhD. Jan patiently shared with me not only some of his knowledge and ideas, but also some of his enthusiasm, motivation, and perseverance. Thank you Jan, my great friend and best man!

I would also like to thank some of my colleagues at Saarland University: Matthias Berg, Chad Brown, Oana Ciobotaru, Markus Dürmuth, Fabienne Eigner, Sebastian Gerling, Mark Kaminski, Stefan Lorenz, Aniket Kate, Boris Köpf, Esfandiar Mohammadi, Kim Pecina, Raphael Reischuk, and Dominique Unruh. They always had an open door and it was an extreme pleasure to work and discuss with them. I am particularly thankful to Dominique for teaching me what open-mindedness really means, and for helping me many times, often on topics that were foreign to him. One special "thank you" goes to Bettina Balthasar, our group's amazing administrative assistant, for taking care of countless numbers of things without a hitch, for getting me a visa appointment a couple of days before a spring school in Japan, for submitting this thesis on my behalf, and for the many many other things with which she helped me.

Kudos to Stefan Lorenz, Kim Pecina, and Thorsten Tarrach for helping with the implementation of the two type systems presented in this thesis. Andy Gordon and Matteo Maffei provided invaluable feedback on countless drafts of this thesis. Many other people provided helpful feedback at various stages of this work: Joshua Dunfield, François Dupressoir, Cédric Fournet, Deepak Garg, Boris Köpf, Kim Pecina, Jan Schwinghammer, Pierre-yves Strub, Dominique Unruh, as well as many anonymous reviewers. A big thank you to everyone!

My graduate studies in Saarbrücken were supported by fellowships from the International Max Planck Research School for Computer Science and Microsoft Research. These institutions offered me not only continuous financial support for five years and a half, but also many opportunities to meet great people and make many friends.

I thank my new advisor, Benjamin Pierce, for his patience, understanding, and trust in the last six months. My new colleagues in the Penn PL Club also helped making the last months more bearable. I especially thank Marco Gaboardi, Michael Greenberg, Benoît Montagu, and Benoît Valiron for their friendship and encouragements.

My family and friends have been a constant source of support. This thesis is dedicated to my wife, Beate, whose endless love gives sense to my life.

Philadelphia, November 12, 2011 Cătălin Hriţcu

> "Peace. It does not mean to be in a place where there is no noise, trouble or hard work. It means to be in the midst of those things and still be calm in your heart." – unknown (**quotable**mugs)

# Contents

# Chapter 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 [DS81, Low96], carefully designed de facto standards like SSL and PKCS [WS96, Ble98], and even widely deployed products like Microsoft Passport [Fis03], Kerberos [BCJ+06, CJS+08], and the SAML-based Single Sign-On for Google Apps [ACC+08]. Manual security analyses of cryptographic protocols, and even more so protocol implementations, are extremely difficult and error-prone. Therefore, it is important to formalize the intended security properties and devise automated analysis techniques for security protocols and, more challengingly, for the source code of distributed applications.

Logic-based authorization policies constitute a well-established and expressive framework for describing a wide range of security properties of cryptographic protocols, varying from authenticity [WL94, GJ03] to access control policies [Aba03]. Furthermore, type systems constitute particularly salient tools to statically and automatically enforce authorization policies on abstract protocol specifications [FGM07b, FGM07a] and on concrete protocol implementations [BBF+08, BFG10]. Type systems require little human effort and provide security proofs for an unbounded number of protocol executions. Furthermore, the analysis is efficient, scalable, and has a predictable termination behavior.

As with all static analysis techniques, the expressiveness of the analysis is very important, since, in order to be useful in practice, a type system for security protocols has to be able to express and reason about many different sound idioms employed by actual protocols.

In this thesis we increase the expressiveness of the existing type-based analyses for security protocol models and implementations using union and intersection types and syntac-

tic reasoning about type disjointness. This increased expressivity enables the analysis of important protocol classes that were previously out of scope for type systems. In particular, we show that the increased expressivity allows us

1. to handle more complex cryptographic primitives, and in this work we will focus mostly on *non-interactive zero-knowledge proofs*;

2. to statically express the invariants of *protocols where participants can be compromised*;

3. to provide solutions that work both for abstract protocol models and for concrete *protocol implementations*.

We discuss each of these three points in the following three sections.

## 1.1. Type-checking Protocols Based on Zero-knowledge Proofs

An important challenge in analyzing protocols is the ability to statically characterize the security properties guaranteed by complex cryptographic operations. For instance, current analysis techniques support traditional cryptographic primitives such as encryption and digital signatures, but until recently [BMU08] they could not cope with zero-knowledge proofs [Gol01]. A zero-knowledge proof combines two seemingly contradictory properties. First, it is a proof of a statement that cannot be forged, i.e., it is impossible, or at least computationally infeasible, to produce a wrong zero-knowledge proof that can pass verification. Second, a zero-knowledge proof does not reveal any information besides the bare fact that the statement is valid.

Early general-purpose zero-knowledge proofs were primarily designed for showing the existence of such proofs for very large classes of statements [GMW91]. These proofs were very inefficient and consequently of only limited use in practical applications. The recent advent of efficient zero-knowledge proofs for special classes of statements [CDS94, CL02, GS08, AFG+10] is changing this scenario. The unique security features offered by zero-knowledge proofs, combined with the possibility to efficiently implement some of these proofs non-interactively [BFM88, BR93, GS08] have paved the way for their deployment in real applications.

For instance, zero-knowledge proofs can guarantee the verifiability of electronic elections yet guarantee the privacy of the votes, as in the Civitas electronic voting system [CCM08]; they can allow for remote attestation of a trusted platform while preserving the anonymity of the users, as in the Direct Anonymous Attestation (DAA) protocol [BCC04]; and they can allow a user to prove to a third party that she holds a government issued electronic ID card and she is over 18, without revealing her identity and her age, as in the IBM idemix [IBM] and Microsoft U-Prove [UPr11] anonymous credential systems. Other than electronic voting [DGS03,Gro05,CCM08,Adi08,HRT10] anonymous

authentication [BCC04, CHK+06], anonymous credentials and digital identity management [CH02, CL04, BCKL08, Hig08, BCC+09, BCGS09, IBM, UPr11] the proposed applications of zero-knowledge proofs include: e-cash [CHL06, CLM07, BCKL09], electronic auctions [LAN02, PRST08], anonymous trust and reputation [BSS10, BLMP10], distributed social networks [TSGW09, BMP11], risk assurance for hedge funds [Szy05], anonymous electronic ticketing for public transportation [HBCDF06], biometric authentication [BSSM+07, KNON10], privacy-friendly smart metering [RD10], and others. In these applications zero-knowledge proofs provide security properties that go beyond the traditional and well-understood secrecy and authenticity properties, allowing the design of protocols that fulfill seemingly conflicting requirements.

Statically analyzing protocols that use zero-knowledge proofs is conceptually and technically challenging. While the existing techniques for type-checking cryptographic protocols typically rely on the type of keys for typing cryptographic messages, these techniques do not directly apply to zero-knowledge proofs, since zero-knowledge proofs do not necessarily depend on a key infrastructure.

In Chapter 2 of this thesis, we introduce the first type system for statically analyzing the security of protocols based on non-interactive zero-knowledge proofs. We model protocols in a variant of the spi-calculus [AG99, AB05], and show how the safety properties guaranteed by zero-knowledge proofs can be formulated in terms of authorization policies and statically enforced by a type system. Our type system combines prior work on dependent and refinement types for cryptographic protocols [FGM07a], with union types, intersection types, and the novel ability to reason statically about the disjointness of types.

Zero-knowledge proofs are given dependent types where the witnesses kept secret by the proof are existentially quantified in the authorization logic. We express zero-knowledge statements as specific positive Boolean formulas in the authorization logic and we define the type of zero-knowledge proofs using these formulas. The user has the possibility to extend such types with additional logical formulas describing protocol-dependent security properties. Our type-checker ensures that honest protocol participants only construct proofs for which these formulas hold. However, the zero-knowledge proofs received from the untrusted network can also be created by malicious attackers, which are untyped and therefore do not play by the rules of the type system. Justifying the formulas conveyed by a zero-knowledge proof in the common case in which the proof comes from the untrusted network is a challenging task, which we solve as follows.

We start from the statement being proved and from the type of those public components of the proof which the verifier has obtained from a reliable source, and use intersection, union, and refinement types to infer very precise type information about the other arguments of the zero-knowledge proof. For certain protocols the types inferred this way are already strong enough to justify the formulas in the zero-knowledge proof type [BGHM09, BLMP10]. In other protocols [BCC04, LHH+07] an additional insight is needed: if the verifier can somehow deduce that the proof was necessarily constructed by an honest (and thus type-checked) prover, then the verifier knows that the formulas

conveyed by the proof were already checked on the prover's side, so they must be valid. The verifier knows that a zero-knowledge proof must be constructed by a honest prover when the type he infers for one of the secret witnesses of the proof is disjoint from the type of messages possibly known to the attacker.

We devise a novel technique for syntactically reasoning about the disjointness of types efficiently and with sufficient precision. The technique is general and should be of independent interest, beyond type-checking security protocols.

## 1.2. Security Despite Compromise

Another important challenge when designing and analyzing cryptographic protocols is the enforcement of security properties in the presence of compromised participants. In the setting of logic-based authorization policies, the notion of "security despite compromise" [FGM07a] captures the intuition that *an invalid authorization decision by an uncompromised participant should only arise if participants on which the decision logically depends are compromised.* The impact of participant compromise should be thus apparent from the authorization policy, without having to study the details of the protocol.

One of the advantages of analyzing security despite compromise using a type system is that type-checking the protocol once can prove the security of the protocol for a large number of compromise scenarios. With a flexible enough type system we should be able to cover all these scenarios with only one set of typing annotations, which decreases the burden on the user of the type system, who needs to write and maintain these typing annotations. Union types can be very useful for achieving this. If we track which participants are compromised by a $\mathsf{Compromised}$ predicate, and we want to annotate a name with type $T$ if participant $A$ is honest and with the (usually weaker) type $U$ if $A$ is compromised, then this can be precisely captured as a union between two refinement types: $\{x : T \mid \neg\mathsf{Compromised}(A)\} \vee \{x : U \mid \mathsf{Compromised}(A)\}$. Such types are in general not expressible in the original type system by Fournet et al. [FGM07a].

Fournet et al. [FGM07a] also observe that in order to fix a protocol that is not secure despite compromise one can either weaken the authorization policy to document all dependencies between participants or correct the specification of the protocol in order to avoid such dependencies. In another work [Gro09, BGHM09], on which we briefly report in §2.6, we provide support for achieving the latter: we devise a general technique for strengthening cryptographic protocols in order to satisfy authorization policies despite participant compromise. We automatically transform the original cryptographic protocols by adding non-interactive zero-knowledge proofs, so that each participant proves that the messages sent to the other participants are generated in accordance to the protocol. The zero-knowledge proofs are forwarded to ensure the correct behavior of all participants involved in the protocol, without revealing any secret data. Moreover,

the transformation automatically derives type annotations for the strengthened protocol from the type annotations of the original protocol.

Using our type-checker for zero-knowledge to validate the protocols produced by this transformation raises, however, additional technical challenges. As explained in §1.1, our technique for type-checking zero-knowledge crucially relies on honest provers being type-checked, and on honest verifiers being able to infer that a proof comes from an honest prover by deducing that one of the witnesses of the proof has a type whose values are not known to the attacker. In the setting of security despite compromise though, all this reasoning has to be conditioned by certain participants being indeed honest.

In order to address this challenge our type system has no unconditionally secure types. Instead, we give a precise characterization of when a type is compromised in the form of a formula in the authorization logic. We use refinement types that contain such logical formulas together with union types to express type information that is conditioned by a participant not being compromised. Such conditional types are inferred automatically when processing the zero-knowledge statement.

## 1.3. Type-checking Protocol Implementations

Abstract protocol models usually disregard many 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 in a mainstream programming language has no security flaws. On top of that, protocol models are often not executable [Bla01], 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[1]. On the other hand, a reference implementation in a mainstream programming language can be written, compiled, executed and debugged using standard tools, and it can be 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 useful 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 very large, so the efficiency and scalability of the analysis is even more important. Additionally, code in mainstream programming languages 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 [Bla01]

[1] If one is only interested in (robust) safety properties, then a completely dysfunctional model is the most secure.

when used as a back end for analyzing protocol implementations [BFGT08]. The type systems for programming languages, on the other hand, were designed with these features in mind, and the analysis they provide is inherently modular. Consequently, type systems are more efficient and scale better than the state-of-the-art protocol verifiers for the analysis of source code [BFG10].

In Chapter 3 of this thesis, we show that our technique for type-checking protocol models can be adapted to the setting of protocol implementations. We add union, intersection and polymorphic types as well as the ability to reason about type disjointness to the refinement type system proposed by Bengtson et al. [BBF$^+$08, BBF$^+$11]. The increased expressivity allows us to 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.

Protocols are implemented in RCF$^{\forall}_{\wedge\vee}$, a concurrent lambda-calculus that is expressive enough to encode a considerable fragment of an ML-like programming language [BBF$^+$08]. As in the spi-calculus [AG99], cryptographic operations are considered fully reliable building blocks via a symbolic abstraction of cryptography. As opposed to the spi-calculus, the cryptographic operations are not primitive in RCF$^{\forall}_{\wedge\vee}$, but are instead encoded using a dynamic sealing mechanism [Mor73, SP07, BBF$^+$08], which is in turn based on standard functional programming language constructs. The resulting symbolic cryptographic library is thus type-checked using regular typing rules for functional languages; in particular these typing rules are not specific to cryptography. We use union and intersection types to give stronger and more natural types to the operations for asymmetric cryptography than in the original sealing-based symbolic cryptographic library of Bengtson et al. [BBF$^+$08]. 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 out of the box. 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. This symbolic implementation is type-checked using standard typing rules, which are not tailored in any way to zero-knowledge proofs. In achieving this we crucially rely on union and intersection types, as well as on pruning typing derivation branches based on static information about the disjointness of types.

Finally, we show that our type system can easily reason about authenticity and integrity properties achieved by showing the knowledge of secret data, as in the Needham-Schroeder-Lowe public-key protocol [Low96] 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 [BCC04] and Civitas [CCM08]).

This common cryptographic pattern could not be handled by the original type system proposed by Bengtson et al. [BBF+08, BBF+11].

## 1.4. Soundness Proofs and Implementations

We have formalized the two calculi and their operational semantics, our two type systems, and all the important parts of the soundness proofs in the Coq proof assistant[2]. We believe these mechanized formalizations are important, since the powerful combination of union, intersection, and refinement types and reasoning about type disjointness makes the soundness proofs non-trivial, tedious, and potentially error-prone. Indeed, this work has allowed us to discover several relatively small problems in the soundness proofs of prior type systems with refinement types [FGM07a, BBF+08], as well as our own previous manual proofs [BHM08c, BGHM09], and to propose and evaluate fixes for the affected definitions and proofs. Finally, we remark that 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 [BBF+08, BBF+11, BHM08c].

We have also implemented efficient type-checkers for the two type systems presented in this thesis. They rely on first-order logic automated theorem provers or SMT solvers to discharge proof obligations. Both type-checkers performed very well in our experiments. The spi-calculus type-checker can verify the authenticity properties of a model of the Direct Anonymous Attestation protocol (DAA) [BCC04] in less than three seconds, on a normal laptop. The type-checker for $\mathrm{RCF}^{\forall}_{\wedge\vee}$ can verify our symbolic cryptographic library and sample code totaling more than 1500LOC in around 12 seconds. These promising results indicate that our analysis technique has the potential to scale up to very large protocol models and implementations.[3]

## 1.5. Previous Publications

The results presented in this thesis have previously appeared in a series of conference publications that I have coauthored:

- *Type-checking Zero-knowledge.* In 15th ACM Conference on Computer and Communications Security (CCS 2008), pages 357-370, ACM Press, October 2008. Joint work with Michael Backes and Matteo Maffei.

[2] Full disclosure: On 21st of December 2011 we have discovered a flaw affecting the weakening property of the spi-calculus type system presented in this thesis. We hope to fix this problem in future work.

[3] The soundness proofs and implementations of our two type systems are publicly available at `http://www.infsec.cs.uni-saarland.de/projects/zk-typechecker/`, and respectively at `http://www.infsec.cs.uni-saarland.de/projects/F5/`.

- *Achieving Security Despite Compromise Using Zero-knowledge.* In 22th IEEE Symposium on Computer Security Foundations (CSF 2009), pages 308-323, IEEE Computer Society Press, July 2009. Joint work with Michael Backes, Martin Grochulla, and Matteo Maffei.
- *Union and Intersection Types for Secure Protocol Implementations.* To appear in Theory of Security and Applications (TOSCA'11), Invited Paper, April 2011. Joint work with Michael Backes and Matteo Maffei.

Preliminary results were also presented at several workshops:

- *Type-checking Zero-knowledge.* Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08), June 2008. Joint work with Michael Backes and Matteo Maffei.
- *Achieving Security Despite Compromise Using Zero-knowledge.* Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09), March 2009. Joint work with Michael Backes, Martin Grochulla, and Matteo Maffei.
- *Type-checking Implementations of Protocols Based on Zero-knowledge Proofs - Work in Progress.* Workshop on Foundations of Computer Security (FCS 2009), August 2009. Joint work with Michael Backes, Matteo Maffei, and Thorsten Tarrach.

For the present thesis, I have extended and streamlined the presentation of our results. Additionally, for this thesis I have mechanized our proofs in the Coq proof assistant.

## 1.6. Outline

The remainder of the thesis is structured into three chapters. Chapter 2 presents our type system for analyzing abstract models of protocols based on zero-knowledge proofs. Chapter 3 presents our type system for analyzing implementations of security protocols. Chapter 2 and Chapter 3 are reasonably self-contained and can in principle be read in any order. Chapter 4 concludes and discusses several directions for future work.

# Chapter 2

# Analyzing Protocol Models

In this chapter, we introduce the first type system for statically analyzing the security of protocols based on non-interactive zero-knowledge proofs. We model protocols in a variant of the spi-calculus, and show how the safety properties guaranteed by zero-knowledge proofs can be formulated in terms of authorization policies and statically enforced by our type system. Our type system combines prior work on dependent and refinement types for cryptographic protocols [FGM07a], with union types, intersection types, and the novel ability to reason statically about the disjointness of types.

Zero-knowledge proofs are given dependent types where the witnesses kept secret by the proof are existentially quantified in the authorization logic. We express zero-knowledge statements as specific positive Boolean formulas in the authorization logic and we define the type of zero-knowledge proofs using these formulas. The user has the possibility to extend such types with additional logical formulas describing protocol-dependent security properties. Justifying the formulas conveyed by a zero-knowledge proof when the verified proof comes from the untrusted network is the main challenge when type-checking protocols based on zero-knowledge. We use union and intersection types, as well as syntactic reasoning about type disjointness to address this challenge.

We have formalized our type system and all the important parts of the soundness proof in the Coq proof assistant. This has allowed us to discover several relatively small problems in our previous manual proofs [BHM08c,BGHM09], to properly fix the affected definitions and proofs, and, in the end, to obtain a high degree of confidence in the soundness of the type system.

We use two bigger case studies to illustrate the applicability of our type system to real-world protocols. First, we use our type system to verify the security despite compromise of protocols that were automatically strengthened in this respect by adding

non-interactive zero-knowledge proofs [Gro09, BGHM09]. Second, we verify the authenticity properties of the Direct Anonymous Attestation (DAA) protocol [BCC04].

We have implemented a new type-checker that automates the analysis. It relies on first-order logic automated theorem provers [WDF+09, Sch02, RV99] or SMT solvers [dMB08] to discharge proof obligations. The type-checker is very efficient: it can verify the authenticity properties of the DAA protocol [BCC04] in less than three seconds, on a normal laptop.

**Outline** §2.1 discusses related work. §2.2 illustrates our approach on a simple anonymous authentication protocol inspired by DAA-signing. §2.3 introduces the process calculus we use to model security protocols that use zero-knowledge proofs. §2.4 presents our type system for zero-knowledge. §2.5 discusses the soundness of our type system and the machine-checked formalization in Coq. In §2.6 we use our type system to verify the security despite compromise of protocols automatically strengthened using zero-knowledge proofs. In §2.7 we apply our type system to analyze a model of the complete DAA protocol. §2.8 discusses the implementation of our type-checker. §2.9 gives a summary of the chapter. Appendix A lists the rules for typing blind signatures, used by the DAA protocol from §2.7.

## 2.1. Related Work

Dating back to the seminal work by Abadi on secrecy by typing [Aba99, AB03], type systems were successfully employed to analyze a wide range of security properties of cryptographic protocols, ranging from authenticity properties [GJ04, HJ05, HJ06, BFM07, BCFM07], to security despite compromised participants [GJ05, BFM07, FGM07a, BCD+09a], to authorization policies [FGM07b, FGM07a, BBF+08, BCEM11]. Type-checking is efficient, scalable, and has a predictable termination behavior. None of the existing type systems for cryptographic protocols is, however, capable of dealing with zero-knowledge proofs.

Until recently, ProVerif [Bla01, BAF08] has been the only automatic tool that has been applied to the analysis of protocols that use non-interactive zero-knowledge proofs [BMU08, BHM08a, DRS08]. ProVerif is based on Horn-clause resolution and can analyze trace-based security properties as well as selected observational equivalences [BAF08]. The analysis with ProVerif is, however, not compositional and often has unpredictable termination behavior, with seemingly harmless code changes leading to divergence. In terms of expressivity, several type systems for security [Aba99, AB03] can deal with strong secrecy, which is defined as an observational equivalence, while ProVerif can check a slightly more general notion of observational equivalence based on bi-processes [BAF08]. This allows ProVerif to express certain behavioral properties that are out of scope for current type systems, such as vote privacy and coercion-resistance in electronic-voting protocols [DKR09, BHM08a]. ProVerif is, however, restricted to cryptographic primitives

that can be expressed as convergent or linear equational theories. Our type-based analysis does not pose any constraint on the semantics of cryptographic primitives and, as opposed to ProVerif, can deal with authorization policies using arbitrary logical structure (e.g., arbitrarily nested quantifiers).

More recently, Camenisch et al. [CMS10] propose another way to model non-interractive zero-knowledge proofs symbolically and use this to model-check the Identity Mixer anonymous credential system developed at IBM [CL01] using the AVISPA tool [ABB+05].

The symbolic, Dolev-Yao style [DY83], abstraction of non-interactive zero-knowledge proofs we use in this chapter was first proposed by Backes et al. [BMU08]. Backes and Unruh [BU08] later studied the conditions a cryptographic zero-knowledge proof system needs to satisfy beyond the standard ones in order to serve as a computationally sound implementation of this symbolic abstraction. Backes and Mohammadi [Moh09, BM11] have recently shown that non-standard conditions such as non-malleability and extractability can be relaxed if the symbolic abstraction is changed accordingly.

Backes et al. [BMM10] propose a computationally sound abstraction of secure multi-party computation. They use this abstraction together with the type system from this chapter to automatically verify the security of protocols that use secure multi-party computations as a building block. They use our type-checker "out of the box" together with the Vampire theorem prover [RV99] to automatically verify the global security properties of the SIMAP sugar-beet double auction protocol [BCD+09b].

In another work [Gro09,BGHM09], we devise a general technique for strengthening cryptographic protocols in order to satisfy authorization policies despite participant compromise [FGM07a]. We automatically transform the original cryptographic protocols by adding non-interactive zero-knowledge proofs, so that each participant proves that the messages sent to the other participants are generated in accordance to the protocol. The zero-knowledge proofs are forwarded to ensure the correct behavior of all participants involved in the protocol, without revealing any secret data. Moreover, the transformation automatically derives type annotations for the strengthened protocol from the type annotations of the original protocol. We use our type checker to validate that the protocols generated by this transformation are secure despite compromise. Type-checking the generated protocols raised technical challenges that have motivated some of the design choices for our type system; this is further discussed in §2.6.

Tarrach [Tar08] devised a translation from the spi-calculus into a concurrent lambda calculus (RCF) and proved that the translation preserves security typing. The considered security type systems use refinement types [BBF+08], but are much weaker than the ones presented in this thesis. The type system for the spi-calculus is less precise even than the one by Fournet et al. [FGM07a], and doesn't support nested cryptographic types.

Barthe et al. [BHB+10] devise Coq formalizations for a specific class of efficient cryptographic zero-knowledge proof systems called sigma-protocols [CDS94]. Almeida et al. [ABB+10] construct a certifying compiler for sigma-protocols, which given a specification of a sigma-protocol generates an efficient C implementation together with a

soundness proof for the generated protocol in Isabelle/HOL. Both these formalization results are complementary to ours: while Barthe et al. and Almeida et al. focus on the soundness of the cryptographic primitives for zero-knowledge proofs in a very precise computational model, our work focuses on verifying that these primitives are used correctly when constructing larger cryptographic protocols. Our work uses a more abstract model of cryptography in which cryptographic operations are abstracted as symbolic terms. Computational soundness results can show that under certain assumptions such symbolic abstractions are computationally justified, as done for zero-knowledge proofs by Backes et al. [BMU08, BM11]. Formalizing such results in a theorem prover is, however, a challenging task on its own [SB08b].

## 2.2. Illustrative Example: Simplified DAA-signing

This section highlights the fundamental ideas of our type system, which will be elaborated in more detail in the following sections. As a running example, we consider a much simplified version of the Direct Anonymous Attestation (DAA) protocol [BCC04] (the complete DAA protocol is analyzed in §2.7). DAA is a cryptographic protocol that enables the remote authentication of a hardware device called Trusted Platform Module (TPM) [TCG11], while preserving the anonymity of the user of the device. Such TPMs are included in many personal computers and servers. More precisely, the goal of the DAA protocol is to enable the TPM to send an arbitrary message to a verifier in a way that the verifier is convinced that a valid TPM authenticated the message, but neither the verifier nor any other party learn precisely which TPM was involved in the protocol. The DAA protocol heavily relies on zero-knowledge proofs to achieve this form of anonymous authentication.

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 made by the issuer on the TPM's secret identifier $f$, and we represent this symbolically as the spi-calculus term $x_{cert} = \mathsf{sign}(f, k_i)$. For the sake of simplicity, in our example we assume that the TPM has already completed the join protocol and received the certificate $x_{cert}$ from the issuer. Instead we will focus only on the DAA-signing protocol, which enables the TPM to authenticate a message $m$ by proving to the verifier the knowledge of a valid certificate. The TPM sends to the verifier a non-interactive zero-knowledge proof, which shows that the TPM knows a secret TPM identifier $f$ and a valid certificate $x_{cert}$ for $f$ signed by the issuer, but without revealing $f$ or $x_{cert}$ to the verifier.

Following Backes et al. [BMU08], we represent this zero-knowledge proof by the term $\mathsf{zk}_{S_{sdaa}}(f, x_{cert}; y_{vki}, m)$. The arguments $f$ and $x_{cert}$ are kept secret by the proof, while the verification key of the issuer $y_{vki}$ and the message $m$ are revealed to the verifier and to any other party receiving the proof. We express this by marking the variables $x_f$ and

$x_{cert}$ as secret witnesses, and by marking $y_{vki}$ and $y_m$ as public messages in the definition of the statement $S_{sdaa}$:

$$S_{sdaa} = \mathsf{witness}\ x_f, x_{cert}\ \mathsf{public}\ y_{vki}, y_m\ \mathsf{in}\ B_{sdaa}$$

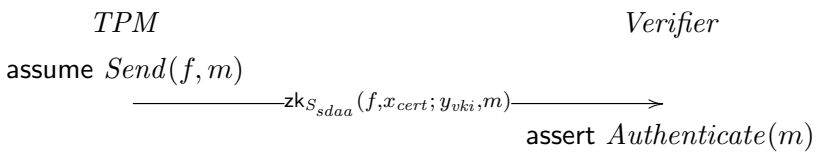
The logical formula of the statement can use these placeholder variables to refer to the actual arguments. In our case this formula is very simple

$$B_{sdaa} = \mathsf{check}(x_{cert}, y_{vki}) \rightsquigarrow x_f,$$

and states that checking the certificate $x_{cert}$ using the verification key of the issuer $y_{vki}$ succeeds and yields the TPM identifier $x_f$. Note that, although the payload message $y_m$ does not occur in the formula, the zero-knowledge proof guarantees non-malleability – the attacker cannot use an existing proof $\mathsf{zk}_{S_{sdaa}}(f, x_{cert};\ y_{vki}, m)$ to produce another proof $\mathsf{zk}_{S_{sdaa}}(f, x_{cert};\ y_{vki}, m')$ without knowing all the arguments, including $f$ and $x_{cert}$.

A zero-knowledge proof is valid if after substituting the placeholder variables with the actual terms passed as arguments we obtain a valid formula. In our case the zero-knowledge proof $\mathsf{zk}_{S_{sdaa}}(f, \mathsf{sign}(f, k_i);\ \mathsf{vk}(k_i), m)$ is valid, since after substituting $x_f$ by $f$, $x_{cert}$ by $\mathsf{sign}(f, k_i)$, and $y_{vki}$ by $\mathsf{vk}(k_i)$ in $B_{sdaa}$, we obtain the formula $\mathsf{check}(\mathsf{sign}(f, k_i), \mathsf{vk}(k_i)) \rightsquigarrow f$, which is valid, since in the semantics of our calculus checking a correct signature for the given verification key yields indeed the message that was originally signed.

In order to express the security property expected from our simplified DAA-signing protocol as an authorization policy, we decorate the protocol with security-related events as follows:




Before sending the zero-knowledge proof, the TPM assumes $\mathit{Send}(f, m)$. If the verification of the received proof succeeds then the verifier asserts $\mathit{Authenticate}(m)$. The authorization policy we consider for the protocol is:

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

where the predicate $\mathit{OkTPM}(f)$ is assumed by the issuer before signing $f$. Since this is the only assumption where the *Authenticate* predicate occurs, the verifier is allowed to authenticate a message $m$, only if $m$ was sent by a TPM whose $f$-value is trusted by the issuer. Everything that is not explicitly allowed by the authorization policy is prohibited.

We are now ready to write a model of our protocol as a spi-calculus process, by adding the missing details to the informal arrow notation above.

**Spi-calculus model of our simplified DAA-sign protocol**

$TPM =$
  *(* abstract away the join protocol *)*
  (new $f$ : Private)
  assume $OkTPM(f)$ |
  let $x_{cert} = \mathsf{sign}(f, k_i)$ in
    *(* simplified DAA-sign protocol *)*
    (new $m$ : Un)
      (assume $Send(f, m)$) |
      $\mathsf{out}(c, \mathsf{zk}_{S_{sdaa}}(f, x_{cert}; y_{vki}, m))$

$SimplifiedDAA =$
  (new $k_i$ : $\mathsf{SigKey}(T_{ki})$)
  let $y_{vki} = \mathsf{vk}(k_i)$ in
    $(TPM \mid Verifier \mid Policy_{sdaa})$

$Verifier =$
  $!\mathsf{in}(c, x_z)$.
  if $\mathsf{ver}_{S_{sdaa}}(x_z, y_{vki}) \Downarrow y_m$ then
    assert $Authenticate(y_m)$

The process *SimplifiedDAA* sets up the protocol. It first creates the signing key of the issuer $k_i$, and it obtains the corresponding verification key $y_{vki}$ by applying the vk constructor to $k_i$. It then runs the *TPM* and *Verifier* processes in parallel, and assumes the authorization policy $Policy_{sdaa}$. Since we consider that the *TPM* has already completed the join protocol, and obtained a valid certificate for its $f$ value, the *TPM* process starts by setting up a variable $x_{cert}$ in which it stores this certificate. It proceeds by creating a message $m$, marking the point when it is about to send $m$ by assuming the $Send(f, m)$ predicate, and finally sending the zero-knowledge proof $\mathsf{zk}_{S_{sdaa}}(f, x_{cert}; y_{vki}, m)$ to the *Verifier* over the public channel $c$. The *Verifier* receives the proof from $c$ in variable $x_z$ and spawns a new process to handle the request (the "!" in front of the in process ensures that the *Verifier* can handle multiple incoming requests). The ver condition of the if-then process checks whether $x_z$ is a valid proof for statement $S_{sdaa}$. Additionally, the ver checks whether the verification key of the issuer $y_{vki}$, which the *Verifier* is assumed to have from a trusted source, matches the first public argument of the proof. If both these checks succeed, then the payload message is returned in variable $y_m$ and the assert $Authenticate(y_m)$ on the then branch is activated.

Intuitively, the assert succeeds if the asserted formula is logically entailed by the previously activated assumes. In our simple example it is easy to see that the assert succeeds when the proof comes indeed from the *TPM*. In this case the $Authenticate(y_m)$ predicate follows from the authorization policy, because the *TPM* assumed $Send(f, m)$ as well as $OkTPM(f)$ (on behalf of the issuer), before sending the message, and the semantics of the verification ensures that $y_m = m$. The property we want for our protocols is, however, much stronger than this. We want that in all executions all asserts succeed even in the presence of an arbitrary Dolev-Yao attacker; following Gordon and Jeffrey [GJ03] we call this stronger properly *robust safety*. The attacker has complete control over the public channels: it can read, block, forward messages, and it can inject new messages constructed from the terms it has already obtained. In particular the attacker can create zero-knowledge proofs from terms it already knows and send them to the *Verifier* without assuming any predicate. If an attacker existed that was able to construct a new proof that could pass verification, our simple protocol would not be robustly safe. Because manually reasoning about the security of protocols under this strong attacker model is

very difficult even for simple protocols like ours, we propose a new type system that can automatically verify if a protocol using zero-knowledge proofs is robustly safe.

Our type system requires that all freshly generated names carry type annotations. For instance, the payload message $m$ generated by the *TPM* has type $\mathsf{Un}$, the type of messages possibly known to the attacker. The attacker can in fact extract $m$ from the zero-knowledge proof sent over the public channel. On the other hand, the *TPM* identifier is given type $\mathsf{Private}$, the type of messages unknown to the attacker. Because of the first assert in the *TPM*'s code, we can in fact give $f$ an even stronger type: the refinement type $T_{ki} = \{x_f : \mathsf{Private} \mid \mathit{OkTPM}(x_f)\}$, which states not only that $f$ is not known to the attacker, but also that the predicate $\mathit{OkTPM}(f)$ holds. We use this refinement type to annotate the signing key of the issuer: we give $k_i$ type $\mathsf{SigKey}(T_{ki})$. Our type system ensures that the key $k_i$ can only be used to sign private messages for which the *OkTPM* predicate holds. The type system has to make sure not only that the code of the protocol respects this restriction, but also that $k_i$ is never leaked to the attacker. By enforcing these restrictions on the usage of the signing key $k_i$, the type system can infer that in case checking a signature with the corresponding verification key $\mathsf{vk}(k_i)$ succeeds, the signed message has to be of type $T_{ki}$, and therefore the predicate *OkTPM* has to hold for it. So, had the *TPM* sent the certificate to the *Verifier* in clear and had the verifier checked the signature directly, the type of the key $k_i$ would have allowed us to "transfer" the predicate *OkTPM* from the *TPM* to the *Verifier*.

In the DAA protocol, however, the *TPM* does not send the certificate to the *Verifier*, the *TPM* only proves that it knows a certificate. So the verifier cannot use the key to directly check a certificate it never receives. In general zero-knowledge proofs do not have to rely on keys – protocols based on zero-knowledge proofs exist that do not use any key whatsoever, such as the PseudoTrust protocol proposed by Lu et al. in [LHH+07].

Our solution is to require the user to provide a type for each statement proved by zero-knowledge in the protocol. In our example the statement $S_{sdaa}$ is associated the type

$$\mathsf{ZKProof}_{S_{sdaa}}(y_{vki} : \mathsf{VerKey}(T_{ki}), y_m : \mathsf{Un};\ \exists x_f, x_{cert}.\ \mathit{Send}(x_f, y_m) \wedge \mathit{OkTPM}(x_f))$$

This dependent type lists the types of the public arguments $y_{vki}$ and $y_m$, and contains an authorization logic formula that is conveyed by the proof, where the witnesses kept secret by the proof $x_f$ and $x_{cert}$ are existentially quantified. In our example the proof is declared to convey the predicates $\mathit{Send}(x_f, y_m)$ and $\mathit{OkTPM}(x_f)$. Our type system ensures that the code of the protocol only constructs zero-knowledge proofs for which these two predicates hold. When type-checking the code of the *TPM* that creates the zero-knowledge proof $\mathsf{zk}_{S_{sdaa}}(f, x_{cert};\ y_{vki}, m)$ it is very easy to check that these predicates hold, since they are both assumed before in the *TPM*'s code.

Justifying these predicates on the *Verifier*'s side is, however, much more challenging. The *Verifier* receives the zero-knowledge proof it verifies from an untrusted channel, which we assume to be under the control of the attacker. So the verified proof could indeed come from the *TPM*, in which case the predicates are very easy to justify, since

they were already checked on the *TPM*'s side. But the proof could also come from the attacker, who does not play by the rules of our type system, and does not need to assume any predicates before creating valid zero-knowledge proofs.

The type system enforces that only messages of type Un are given to the attacker, so if the proof was constructed by the attacker, then all its arguments ($x_f$, $x_{cert}$, $y_{vki}$, and $y_m$) would have type $\mathsf{Un}$. On the other hand, the *Verifier* has obtained the signature verification key of the issuer $\mathsf{vk}(k_i)$ from a trusted source, so $\mathsf{vk}(k_i)$ has type $\mathsf{VerKey}(T_{ki})$. If the verification of the zero-knowledge proof succeeds then the operational semantics guarantees that $y_{vki} = \mathsf{vk}(k_i)$, so we additionally know that $y_{vki}$ has type $\mathsf{VerKey}(T_{ki})$. We use an intersection type to express the fact that $y_{vki}$ has at the same time type $\mathsf{Un}$ and type $\mathsf{VerKey}(T_{ki})$, i.e., $y_{vki}$ has type $\mathsf{Un} \wedge \mathsf{VerKey}(T_{ki})$. So if the proof comes from the attacker, the type system initially has the following type information:

$$x_f : \mathsf{Un}, x_{cert} : \mathsf{Un}, y_{vki} : (\mathsf{Un} \wedge \mathsf{VerKey}(T_{ki})), y_m : \mathsf{Un}.$$

Additionally, if the zero-knowledge verification succeeds then the operational semantics guarantees that the formula of the proved statement $B_{sdaa} = \mathsf{check}(x_{cert}, y_{vki}) \rightsquigarrow x_f$ holds, no matter what the provenance of the proof is. Our type system uses the formula $B_{sdaa}$, together with the initial type information above to infer more precise type information for the arguments of the proof. In our simple example, knowing that $y_{vki}$ has type $\mathsf{VerKey}(T_{ki})$ and that $\mathsf{check}(x_{cert}, y_{vki}) \rightsquigarrow x_f$ holds is enough to infer that $x_f$ has type $T_{ki} = \{x_f : \mathsf{Private} \mid \mathit{OkTPM}(x_f)\}$. This is enough to ensure that the $\mathit{OkTPM}(x_f)$ predicate holds, but it does not help us directly in deriving the $\mathit{Send}(x_f, y_m)$ predicate. For this we use the following insight: since $x_f$ has type $T_{ki}$ and since $T_{ki}$ is a subtype of type $\mathsf{Private}$ we also have that $x_f$ has type $\mathsf{Private}$. But, we already know that $x_f$ also has type $\mathsf{Un}$, and the types $\mathsf{Private}$, the type of messages not known to the attacker, and $\mathsf{Un}$, the type of messages possibly known to the attacker, are disjoint. Since there exists no closed term for $x_f$ that satisfies these typing constraints, the proof cannot come from the attacker, so the proof must come from the *TPM*, and the two predicates were already checked on the *TPM*'s side, so they can indeed also be justified on the *Verifier*'s side.

Our type-checker performs this reasoning in an automatic and completely rigorous way (see §2.4.9). It can verify that this simple protocol is robustly safe in less than half a second on a normal laptop.

## 2.3. Spi-calculus with Zero-knowledge Proofs

We consider a variant of the spi-calculus [AG99] with constructors and destructors similar to the one implemented by ProVerif [AB05], and we extend it with zero-knowledge proofs. Following Fournet et al. [FGM07a, BBF+08], the calculus also includes special operators to assume and assert authorization logic formulas. This section presents the syntax and operational semantics of the calculus.

In the following we identify any phrase $\phi$ of syntax up to consistent renaming of bound names and variables. We say that $\phi$ is closed if it does not have any free variables. We write $\phi\{\phi'/x\}$ for the outcome of the capture-avoiding substitution of $\phi'$ for each free occurrence of $x$ in $\phi$. Our Coq formalization described in §2.5 is much more precise about binders, but for the sake of readability in the rest of this chapter we use the familiar named representation of binders.

### 2.3.1. Terms and Destructors

In our spi-calculus variant, cryptographic operations are either represented as symbolic *terms* (constructors) [DY83], or they operate on such terms (destructors).

**Terms**

| $K, L, M, N ::=$ | terms |
|---|---|
| $a, b, c, k$ | names |
| $x, y, z, v$ | variables |
| $(M, N)$ | pair |
| $\mathsf{unit}$ | unit |
| $\mathsf{ek}(K)$ | encryption key corresponding to decryption key $K$ |
| $\mathsf{enc}(M, K)$ | encrypt $M$ with public key $K$ |
| $\mathsf{vk}(K)$ | verification key corresponding to signing key $K$ |
| $\mathsf{sign}(M, K)$ | signature on $M$ with key $K$ |
| $\mathsf{hash}(M)$ | hash of message $M$ |
| $\mathsf{senc}(M, K)$ | encryption of $M$ with symmetric key $K$ |
| $\mathsf{zk}_S(\widetilde{N}; \widetilde{M})$ | zero-knowledge proof |

**Notation:** We write $\widetilde{M}$ for the sequence $M_1, \ldots, M_n$ when $n$ is clear from the context. We write $\langle\widetilde{M}\rangle$ for the encoded tuple $(M_1, (M_2, \ldots, (M_n, \mathsf{unit})\ldots))$.

Terms are built from variables and names by applying constructors (i.e., function symbols). The constructors we consider in this thesis include pairing, public-key encryption, digital signatures, symmetric encryption and hashing.[1]

**Destructors**

| $D ::=$ | destructors |
|---|---|
| $\mathsf{id}(M)$ | identity |
| $\mathsf{eq}(M, N)$ | equality on terms |
| $\mathsf{dec}(M, K)$ | decrypt $M$ using private key $K$ |
| $\mathsf{check}(M, K)$ | check signature $M$ using verification key $K$ |
| $\mathsf{sdec}(M, K)$ | decrypt $M$ using symmetric key $K$ |

[1] The complete DAA protocol in §2.7 additionally makes use of blind signatures. Our type-checker supports arbitrary constructors and destructors in a generic way (see §2.8).

| | |
|---|---|
| $\mathsf{public}_S(M)$ | obtain the public arguments of zero-knowledge proof $M$ |

Destructors are cryptographic operations that processes can apply to terms, such as decrypting or checking signatures. We also represent equality between terms as a destructor $\mathsf{eq}$.

**Reduction of destructors:** $D \Downarrow M$

$\mathsf{id}(M) \Downarrow M$
$\mathsf{eq}(M, M) \Downarrow M$
$\mathsf{dec}(\mathsf{enc}(M, \mathsf{ek}(K)), K) \Downarrow M$
$\mathsf{check}(\mathsf{sign}(M, K), \mathsf{vk}(K)) \Downarrow M$
$\mathsf{sdec}(\mathsf{senc}(M, K), K) \Downarrow M$
$\mathsf{public}_S(\mathsf{zk}_S(\widetilde{N};\ \widetilde{M})) \Downarrow \langle \widetilde{M} \rangle$

**Notation:** We write $D \not\Downarrow$ if there exists no $M$ so that $D \Downarrow M$, i.e., the destructor fails.

The semantics of destructors is specified by the destructor reduction relation $\Downarrow$: a destructor $D$ can either succeed and produce a term $M$ as result (which we denote as $D \Downarrow M$) or it can fail if none of the reduction rules apply (denoted as $D \not\Downarrow$). The $\mathsf{dec}$ destructor decrypts an encrypted message given the correct decryption key. The $\mathsf{check}$ destructor checks the validity of a signature using a verification key, and if this check succeeds the message without the signature is returned.

### 2.3.2. Representing Zero-knowledge Proofs

As first proposed by Backes et al. [BMU08], a non-interactive zero-knowledge proof of a statement $S$ is represented as a term of the form $\mathsf{zk}_S(N_1, \ldots, N_n;\ M_1, \ldots, M_m)$, where $N_1, \ldots, N_n$ and $M_1, \ldots, M_m$ are two lists of terms. The proof keeps the witnesses $N_1, \ldots, N_n$ secret, while the terms $M_1, \ldots, M_m$ are revealed (e.g., using the $\mathsf{public}_S$ destructor).

The statement $S$ conveyed by a zero-knowledge proof $\mathsf{zk}_S(N_1, \ldots, N_n;\ M_1, \ldots, M_m)$ has the form $\mathsf{witness}\ x_1, \ldots, x_n\ \mathsf{public}\ y_1, \ldots, y_m\ \mathsf{in}\ B$. The statement $S$ contains a basic formula $B$ and additionally declares (i.e., binds) the variables used in this formula: $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$. The variables $x_1, \ldots, x_n$ are placeholders for the secret witnesses $N_1, \ldots, N_n$, while $y_1, \ldots, y_m$ are placeholders for the public arguments $M_1, \ldots, M_m$. In order for $S$ to be well-formed we require that its basic formula $B$ uses no other variables than the declared ones, and contains no free names.

**Zero-knowledge statements**

| $S ::=$ | statement |
|---|---|
| $\mathsf{witness}\ \widetilde{x}\ \mathsf{public}\ \widetilde{y}\ \mathsf{in}\ B$ | scope of $\widetilde{x}$ and $\widetilde{y}$ is $B$ and $\widetilde{x}, \widetilde{y}$ pairwise distinct |

| $B ::=$ | basic formula |
|---|---|
| $D \rightsquigarrow M$ | destructor reduction |
| $B_1 \wedge B_2$ | conjunction |
| $B_1 \vee B_2$ | disjunction |

**Notation:** We encode the equality basic formula using the equality destructor $M = N \triangleq \mathsf{eq}(M, N) \rightsquigarrow M$, and the Boolean constants using equality: $\mathsf{true} \triangleq \mathsf{unit} = \mathsf{unit}$, $\mathsf{false} \triangleq \mathsf{unit} = (\mathsf{unit}, \mathsf{unit})$.

Basic formulas are positive Boolean formulas formed using a special binary predicate $\rightsquigarrow$, capturing the destructor reduction relation, as well as conjunctions and disjunctions of other basic formulas. This representation of statements allows us to express a wide class of zero-knowledge proofs, comprising for instance proof of signature verifications, decryptions, equalities, as well as Boolean combinations [Cra96].

For instance the zero-knowledge statement $S_{\text{pk}} = \mathsf{witness}\ x_k\ \mathsf{public}\ y_a, y_{pk}\ \mathsf{in}\ B_{\text{pk}}$ where $B_{\text{pk}} = \mathsf{dec}(\mathsf{enc}(y_a, y_{pk}), x_k) \rightsquigarrow y_a$ proves the knowledge of the secret decryption key $x_k$ corresponding to the public encryption key $y_{pk}$. Intuitively, the statement reads: "There exists a secret key $x_k$ such that the decryption of the ciphertext $\mathsf{enc}(y_a, y_{pk})$ with $x_k$ yields $y_a$". The values of $y_a$ and $y_{pk}$ are revealed by the proof while $x_k$ is kept secret. One valid zero-knowledge proof for this statement can be constructed as $\mathsf{zk}_{S_{\text{pk}}}(k;\ a, \mathsf{ek}(k))$ for two names $a$ and $k$.

Another interesting application is zero-knowledge proofs about certificate chains [BLMP10, MP11]. For instance, the statement

$$S_{\text{chain2}} = \mathsf{witness}\ x_{vk}, x_{cert1}, x_{cert2}\ \mathsf{public}\ y_{vk}, y_m\ \mathsf{in}\ B_{\text{chain2}}, \text{ where}$$
$$B_{\text{chain2}} = \mathsf{check}(x_{cert1}, y_{vk}) \rightsquigarrow x_{vk} \wedge \mathsf{check}(x_{cert2}, x_{vk}) \rightsquigarrow y_m$$

proves the knowledge of a valid certificate $x_{cert2}$ for message $y_m$ that can be verified with a hidden key $x_{vk}$, for which the prover possesses another valid certificate $x_{cert1}$ signed by a trusted party having verification key $y_{vk}$. The only information revealed by the proof is the payload message $y_m$ and the (public) verification key of the trusted party $y_{vk}$. The two certificates $x_{cert1}$ and $x_{cert2}$ and the verification key of the prover $x_{vk}$ are hidden by the proof, which guarantees the anonymity of the prover.

A zero-knowledge proof is valid if after substituting the placeholder variables with the actual terms in its basic formula we obtain a valid basic formula. Validity of basic formulas is straightforward to define based on the reduction relation for destructors and the usual interpretation of conjunction and disjunction.

**Semantics of basic formulas: $B$ valid**

| (Sem Red) | (Sem And) | (Sem Or 1) | (Sem Or 2) |
|---|---|---|---|
| $\dfrac{D \Downarrow M}{D \rightsquigarrow M \text{ valid}}$ | $\dfrac{B_1 \text{ valid} \quad B_2 \text{ valid}}{B_1 \wedge B_2 \text{ valid}}$ | $\dfrac{B_1 \text{ valid}}{B_1 \vee B_2 \text{ valid}}$ | $\dfrac{B_2 \text{ valid}}{B_1 \vee B_2 \text{ valid}}$ |

In the simple example above, in order to check whether $\mathsf{zk}_{S_{\mathrm{pk}}}(k;\, a, \mathsf{ek}(k))$ is a valid proof we substitute $B_{S_{\mathrm{pk}}}\{k/x_k\}\{a/y_a\}\{\mathsf{ek}(k)/y_{pk}\}$ and obtain $\mathsf{dec}(\mathsf{enc}(a, \mathsf{ek}(k)), k) \rightsquigarrow a$, which is a valid basic formula by (Sem Red) and the destructor reduction rule for $\mathsf{dec}$. We can check in a similar way that $\mathsf{zk}_{S_{\mathrm{chain2}}}(\mathsf{vk}(k_x), \mathsf{sign}(\mathsf{vk}(k_x), k_y), \mathsf{sign}(m, k_x);\, \mathsf{vk}(k_y), m)$ is a valid proof for $S_{\mathrm{chain2}}$ from the certificate chain example above. For this we substitute $B_{\mathrm{chain2}}\{\mathsf{vk}(k_x)/x_{vk}\}\{\mathsf{sign}(\mathsf{vk}(k_x), k_y)/x_{cert1}\}\{\mathsf{sign}(m, k_x)/x_{cert2}\}\{\mathsf{vk}(k_y)/y_{vk}\}\{m/y_m\}$ and obtain the following valid formula:

$$\mathsf{check}(\mathsf{sign}(\mathsf{vk}(k_x), k_y), \mathsf{vk}(k_y)) \rightsquigarrow \mathsf{vk}(k_x) \wedge \mathsf{check}(\mathsf{sign}(m, k_x), \mathsf{vk}(k_x)) \rightsquigarrow m.$$

### 2.3.3. Processes

The syntax of processes is mostly standard [AB05,FGM07a]. Replication can only appear before an input ($!\mathsf{in}(M, x).\, P$), which makes the calculus easier to implement [Bus11, BBH11]. Additional to the processes of Fournet et al. [FGM07a], we have an elimination construct for union types ($\mathsf{case}\ x = M\ \mathsf{in}\ P$) [Pie91], and a process to verify the validity of zero-knowledge proofs ($\mathsf{ver}_S(N, \widetilde{M}) \Downarrow x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$). We model the verification of zero-knowledge proofs as a process, and not as a destructor, since this simplifies the semantics of destructors by explicitly forbidding zero-knowledge statements that talk about the validity of other zero-knowledge proofs.

**Syntax of processes**

| $P, Q, R ::=$ | processes |
|---|---|
| $\mathsf{out}(N, M).\, P$ | output $M$ over channel $N$ then continue as $P$ |
| $\mathsf{in}(N, x).\, P$ | input $x$ from channel $N$ (scope of $x$ is $P$) |
| $!\mathsf{in}(N, x).\, P$ | replicated input (scope of $x$ is $P$) |
| $(\mathsf{new}\ a : T)\, P$ | restriction, name $a$ of type $T$ (scope of $a$ is $P$) |
| $P \mid Q$ | parallel composition |
| $\mathbf{0}$ | null process, does nothing |
| $\mathsf{if}\ D \Downarrow x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ | destructor evaluation (scope of $x$ is $P_1$) |
| $\mathsf{ver}_S(N, \widetilde{M}) \Downarrow x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ | zero-knowledge verification (scope of $x$ is $P_1$) |
| $\mathsf{let}\ (x, y) = M\ \mathsf{in}\ P$ | split pair $M$ (scope of $x$,$y$ is $P$ and $x \neq y$) |
| $\mathsf{case}\ x = M\ \mathsf{in}\ P$ | elimination of union types (scope of $x$ is $P$) |
| $\mathsf{assume}\ C$ | add formula $C$ to active assumes |
| $\mathsf{assert}\ C$ | expect formula $C$ to follow from active assumes |

**Notation:** $\mathsf{let}\ x = M\ \mathsf{in}\ P \triangleq \mathsf{if}\ \mathsf{id}(M) \Downarrow x\ \mathsf{then}\ P\ \mathsf{else}\ \mathbf{0}$
$\mathsf{if}\ M = N\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \triangleq \mathsf{if}\ \mathsf{eq}(M, N) \Downarrow x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$, where $x$ is fresh
$\mathsf{let}\ \langle x_1, \ldots, x_n\rangle = M\ \mathsf{in}\ P \triangleq$
$\quad \mathsf{let}\ (x_1, y_1) = M\ \mathsf{in}\ \ldots \mathsf{let}\ (x_n, y_n) = y_{n-1}\ \mathsf{in}\ \mathsf{if}\ y_n = \mathsf{unit}\ \mathsf{then}\ P\ \mathsf{else}\ \mathbf{0}$, for fresh $\widetilde{y}$
$(\mathsf{new}\ \widetilde{a} : \widetilde{T})\, P \triangleq (\mathsf{new}\ a_1 : T_1) \ldots (\mathsf{new}\ a_n : T_n)\, P$

**Convention:** We usually omit $\mathbf{0}$ continuation processes.

As proposed by Fournet et al. [FGM07a], the processes $\mathsf{assume}\ C$ and $\mathsf{assert}\ C$, where $C$ is a formula in the authorization logic, are used to express authorization policies, and do not have any computational significance. Assumptions are used to mark security-related events in processes (such as $\mathsf{assume}\ \mathit{Send}(f, m)$ in §2.2), and also to express global policies (such as $\mathit{Policy}_{sdaa}$ in §2.2). The scope of assumptions is global, i.e., once an assumption becomes active it affects all processes that run in parallel.

Assertions specify logical formulas that are expected to be entailed at run-time by the currently active assumptions (such as $\mathsf{assert}\ \mathit{Authenticate}(y_m)$ in §2.2). Our type system guarantees statically that in any execution of any well-typed protocol all asserts succeed (i.e., are entailed by the currently active assumptions) even in the presence of an arbitrary attacker (see §2.3.6 and §2.5).

### 2.3.4. Operational Semantics

The operational semantics of the calculus is defined by a structural equivalence relation ($P \equiv Q$) and an internal reduction relation ($P \to Q$). Structural equivalence captures rearrangements of parallel compositions and restrictions and is completely standard.

**Structural equivalence:** $P \equiv Q$

(Eq Refl) $P \equiv P$

(Eq Zero Id) $P \mid \mathbf{0} \equiv P$

(Eq Assoc) $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

(Eq Comm) $P \mid Q \equiv Q \mid P$

(Eq Symm) $\dfrac{Q \equiv P}{P \equiv Q}$

(Eq Trans) $\dfrac{P \equiv Q \quad Q \equiv R}{P \equiv R}$

(Eq Scope) $\dfrac{a \notin \mathit{fn}(P_1)}{P_1 \mid ((\mathsf{new}\ a : T)\, P_2) \equiv (\mathsf{new}\ a : T)\, (P_1 \mid P_2)}$

(Eq Ctxt Par) $\dfrac{P \equiv Q}{P \mid R \equiv Q \mid R}$

(Eq Ctxt New) $\dfrac{P \equiv Q}{(\mathsf{new}\ a : T)\, P \equiv (\mathsf{new}\ a : T)\, Q}$

Internal reduction defines the semantics of message passing communication, destructors, pair splits, cases, and, most interestingly, the verification of zero-knowledge proofs.

**Internal reduction:** $P \to Q$

(Red I/O) $(\mathsf{out}(a, M).\, P) \mid (\mathsf{in}(a, x).\, Q) \to P \mid Q\{M/x\}$

(Red !I/O) $(\mathsf{out}(a, M).\, P) \mid (!\mathsf{in}(a, x).\, Q) \to P \mid Q\{M/x\} \mid !\mathsf{in}(a, x).\, Q$

(Red Dtor Then) $\dfrac{D \Downarrow M}{\mathsf{if}\ D \Downarrow x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \to P_1\{M/x\}}$

(Red Dtor Else) $\dfrac{D \not\Downarrow}{\mathsf{if}\ D \Downarrow x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \to P_2}$

(Red Ver Then)
$$\frac{S = \mathsf{witness}\ \widetilde{x}\ \mathsf{public}\ \widetilde{y}\ \mathsf{in}\ B \quad B\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}\ \text{valid} \quad P_1' = P_1\{\langle M_{l+1}, .., M_m\rangle/z\}}{\mathsf{ver}_S(\mathsf{zk}_S(\widetilde{N};\ M_1, .., M_l, .., M_m), M_1, .., M_l) \Downarrow z\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \to P_1'}$$

(Red Ver Else No ZK)
$$\frac{N \text{ is not of the form } \mathsf{zk}_S(\widetilde{N};\ \widetilde{M})}{\mathsf{ver}_S(N, L_1, .., L_l) \Downarrow z\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \to P_2}$$

(Red Ver Else No Match)
$$\frac{\exists i \in \{1, \ldots, l\}.\ M_i \neq L_i}{\mathsf{ver}_S(\mathsf{zk}_S(\widetilde{N};\ M_1, .., M_l, .., M_m), L_1, .., L_l) \Downarrow z\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \to P_2}$$

(Red Ver Else Invalid)
$$\frac{S = \mathsf{witness}\ \widetilde{x}\ \mathsf{public}\ \widetilde{y}\ \mathsf{in}\ B \qquad B\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}\ \text{not valid}}{\mathsf{ver}_S(\mathsf{zk}_S(\widetilde{N};\ M_1, .., M_l, .., M_m), M_1, .., M_l) \Downarrow z\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \to P_2}$$

(Red Split)
$$\mathsf{let}\ (x_1, x_2) = (M_1, M_2)\ \mathsf{in}\ P \to P\{M_1/x_1\}\{M_2/x_2\}$$

(Red Case)
$$\mathsf{case}\ x = M\ \mathsf{in}\ P \to P\{M/x\}$$

(Red Ctxt Par)
$$\frac{P \to Q}{P \mid R \to Q \mid R}$$

(Red Ctxt New)
$$\frac{P \to Q}{(\mathsf{new}\ a : T)\, P \to (\mathsf{new}\ a : T)\, Q}$$

(Red Eq)
$$\frac{P \equiv P' \quad P' \to Q' \quad Q' \equiv Q}{P \to Q}$$

In order for a zero-knowledge verification $\mathsf{ver}_S(N, L_1, \ldots, L_l)$ to succeed for some $S = \mathsf{witness}\ x_1, \ldots, x_n\ \mathsf{public}\ y_1, \ldots, y_m\ \mathsf{in}\ B$ the following three conditions have to hold:

(i) $N$ has to be a zero-knowledge proof for the verified statement $S$, i.e., $N = \mathsf{zk}_S(N_1, .., N_n;\ M_1, .., M_m)$;

(ii) the remaining arguments of the verification $L_1, \ldots, L_l$ have to match the first $l$ public arguments of the proof, i.e., $M_1 = L_1$, $M_2 = L_2$, $\ldots$, $M_l = L_l$;

(iii) the basic formula obtained by substituting the placeholders in $B$ with their actual values has to be valid, i.e., $B\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}$ valid.

If these three conditions hold then verification succeeds and returns the other $m-l$ public arguments of the proof as a tuple: rule (Red Ver Then). Otherwise, verification fails and the else branch is executed: rules (Red Ver Else No ZK), (Red Ver Else No Match), and (Red Ver Else Invalid). Condition (ii) was not present in previous work [BMU08], but, as already illustrated by the simplified DAA-signing example from §2.2, in most cases it is the strong type of the matched terms $L_1, \ldots, L_l$ that allow us to give a strong type to the result of a zero-knowledge verification.

In previous work, Backes et al. [BMU08] define the operational semantics of zero-knowledge proof verification by an infinite equational theory, which needs to be compiled into a convergent rewriting system in order to be suitable for ProVerif [Bla01]. The semantics we give to the verification of zero-knowledge proofs is more direct since we use a type system to analyze the security protocols instead of ProVerif.

### 2.3.5. Authorization Logic

The formulas of the authorization logic are similar to the ones used by Bengtson et al. [BBF+08]. The atomic formulas in our logic are either unary uninterpreted predicate symbols applied to their argument ($p(M)$), or instances of an interpreted binary predicate representing the destructor reduction relation ($D \rightsquigarrow M$). On top of these atomic formulas we have the standard logical operations of first-order logic, with separate quantification over spi-calculus terms ($\forall x.\, C$ and $\exists x.\, C$; variables range over spi-calculus terms) and spi-calculus names ($\forall a.\, C$ and $\exists a.\, C$). We remark that the basic formulas of zero-knowledge proofs can be easily embedded in the authorization logic, so the positive Boolean logic of basic formulas can be seen as a fragment of the authorization logic. Finally, for technical reasons, we also have universal quantification over predicates ($\forall p.\, C$) [FGM07a]. Such second-order quantifiers are only used for obtaining a logical characterization of kinding that enjoys completeness (§2.4.5); they appear neither in authorization policies, nor in the first-order logic proof obligations discharged by our type-checker. Second-order universal quantifiers are only introduced by the logical characterization of kinding and only appear in positive positions, so in practice they can always be "extruded" after proper alpha-renaming, and removed once they reach the top-level since free predicates are implicitly universally quantified.

**Authorization logic**

| $C ::=$ | logical formula |
|---|---|
| $p(M)$ | predicate symbol (unary) |
| $D \rightsquigarrow M$ | destructor reduction (interpreted binary predicate) |
| $C_1 \wedge C_2$ | conjunction |
| $C_1 \vee C_2$ | disjunction |
| $\neg C$ | negation |
| $\forall x.\, C$ | universal quantification over terms ($x$ bound in $C$) |
| $\exists x.\, C$ | existential quantification over terms ($x$ bound in $C$) |
| $\forall a.\, C$ | universal quantification over names ($a$ bound in $C$) |
| $\exists a.\, C$ | existential quantification over names ($a$ bound in $C$) |
| $\forall p.\, C$ | universal quantification over predicates ($p$ bound in $C$) |

**Notations:** We write $M = N$ for $\mathsf{eq}(M, N) \rightsquigarrow M$, also $\mathsf{true} \triangleq \mathsf{unit} = \mathsf{unit}$, $\mathsf{false} \triangleq \neg\mathsf{true}$, $M \neq N \triangleq \neg(M = N)$, $C_1 \Rightarrow C_2 \triangleq \neg C_1 \vee C_2$, and $C_1 \Leftrightarrow C_2 \triangleq C_1 \Rightarrow C_2 \wedge C_2 \Rightarrow C_1$.
We encode n-ary predicates using tuples: $p(\widetilde{M}) \triangleq p(\langle \widetilde{M} \rangle)$

The entailment relation of the authorization logic $A \models C$, where $A$ is a list of formulas, is not completely fixed. The soundness of our type system relies on a set of properties that must be satisfied by the entailment relation. Most of the properties we require are completely standard, such as monotonicity, closure under substitution, cut, and introduction and elimination rules for the logical operations. Additionally, we have two requirements that are specific to our setting: (Red) and (Not Red) require that the

destructor reduction relation is faithfully reflected in the logic. Since equality in the logic is encoded using the equality destructor, these two requirements also fix the standard interpretation of equality. Also, by fixing the interpretation of the $\rightsquigarrow$ relation to match the destructor reduction relation we ensure that the semantics of the authorization logic agrees with the simple Boolean logic from §2.3.2 – i.e., we can prove as a lemma that if $B$ valid then $\emptyset \models B$. The implication in the other direction is not necessary in our proofs, but it does hold under the additional assumption that the authorization logic is consistent[2]. Finally, for proving the completeness of our logical characterization of kinding (§2.4.5) we require that the semantics of disjunction in the authorization logic matches the semantics of disjunction in the meta-logic (in our case Coq).

**Requirements on the entailment relation of the authorization logic:** $A \models C$

| | |
|---|---|
| (Multiset) | if $A_1, A_2 \models C$ then $A_2, A_1 \models C$ |
| (Axiom) | $C \models C$ |
| (Mon) | if $A \models C$ then $A, C' \models C$ |
| (Subst) | if $A \models C$ then $A\{M/x\} \models C\{M/x\}$ |
| (Subst Name) | if $A \models C$ then $A\{b/a\} \models C\{b/a\}$ |
| (Subst Pred) | if $A \models C$ then $A\{q/p\} \models C\{q/p\}$ |
| (Cut) | if $A \models C$ and $A, C \models C'$ then $A \models C'$ |
| (And Intro) | if $A \models C_1$ and $A \models C_2$ then $A \models C_1 \wedge C_2$ |
| (And Elim) | if $A \models C_1 \wedge C_2$ then $A \models C_1$ and $A \models C_2$ |
| (Or Intro) | if $A \models C_1$ or $A \models C_2$ then $A \models C_1 \vee C_2$ |
| (Or Elim) | if $A \models C_1 \vee C_2$ and $A, C_1 \models C$ and $A, C_2 \models C$ then $A \models C$ |
| (Or Elim Closed) | if $A \models C_1 \vee C_2$ and $\mathit{free}(C_1, C_2) = \emptyset$ then $A \models C_1$ or $A \models C_2$ |
| (Impl Intro) | if $A, C_1 \models C_2$ then $A \models C_1 \Rightarrow C_2$ |
| (Impl Elim) | if $A \models C_1 \Rightarrow C_2$ and $A \models C_1$ then $A \models C_2$ |
| (Forall Intro) | if $A \models C$ and $x \notin \mathit{fv}(A)$ then $A \models \forall x.\, C$ |
| (Forall Elim) | if $A \models \forall x.\, C$ then $A \models C\{M/x\}$ |
| (Forall Name Intro) | if $A \models C$ and $a \notin \mathit{fn}(A)$ then $A \models \forall a.\, C$ |
| (Forall Name Elim) | if $A \models \forall a.\, C$ then $A \models C\{b/a\}$ |
| (Forall Pred Intro) | if $A \models C$ and $p \notin \mathit{fp}(A)$ then $A \models \forall p.\, C$ |
| (Forall Pred Elim) | if $A \models \forall p.\, C$ then $A \models C\{q/p\}$ |
| (Exists Intro) | if $A \models C\{x/M\}$ then $A \models \exists x.\, C$ |
| (Exists Elim) | if $A \models \exists x.\, C$ and $A, C \models C'$ and $x \notin \mathit{fv}(A, C')$ then $A \models C'$ |
| (Exists Name Intro) | if $A \models C\{a/b\}$ then $A \models \exists a.\, C$ |
| (Exists Name Elim) | if $A \models \exists a.\, C$ and $A, C \models C'$ and $a \notin \mathit{fn}(A, C')$ then $A \models C'$ |
| (False Intro) | if $A \models \neg C$ and $A \models C$ then $A \models \mathsf{false}$ |
| (False Elim) | if $A \models \mathsf{false}$ then $A \models C$ |
| (Red) | if $D \Downarrow M$ then $\emptyset \models D \rightsquigarrow M$ |
| (Not Red) | if not $D \Downarrow M$ and $\mathit{fv}(D, M) = \emptyset$ then $\emptyset \models \neg(D \rightsquigarrow M)$ |

**Note:** The logics fulfilling these requirements can be classical or constructive.

[2] For proving the soundness of our type system we do not require that the authorization logic is consistent, a property one would surely expect for any practical purpose.

**Note:** This is not an inductive definition, but a set of *minimal* requirements on the authorization logic.

In our implementation we consider classical first-order logic with equality as the authorization logic and we use various automated theorem provers [WDF+09, Sch02, RV99] or SMT solvers [dMB08] to discharge the proof obligations generated by our type system. For the SMT solvers terms and destructors are represented as datatypes, while for the automated theorem provers they are encoded using explicit injectivity and distinctness axioms. The destructor reduction relation is encoded using axioms of the form: $\forall e.\, \forall k.\, \forall m.\, \mathsf{dec}(e, k) \rightsquigarrow m \Leftrightarrow e = \mathsf{enc}(m, \mathsf{ek}(k))$. Please note that the trivial encoding $\forall m.\, \forall k.\, \mathsf{dec}(\mathsf{enc}(m, \mathsf{ek}(k)), k) \rightsquigarrow m$, would not satisfy requirement (Not Red), since the trivial interpretation for $\rightsquigarrow$ that relates all destructors to all terms would fulfill such "positive" axioms, while invalidating (Not Red).

### 2.3.6. Safety and Robust Safety

Intuitively, a process is safe if in all executions all active assertions are entailed by the active assumptions. For this we define a recursive function $\overline{Q}$ that, given a process $Q$, extracts the assumptions currently active in $Q$ as a formula. For restrictions this function uses existential quantification over names in the authorization logic.

**Assumption extraction:** $\overline{Q}$

| | |
|---|---|
| $\overline{\mathsf{assume}\ C} = C$ | $\overline{A \mid B} = \overline{A} \wedge \overline{B}$ |
| $\overline{(\mathsf{new}\ a : T)\, A} = \exists a.\, \overline{A}$ | $\overline{P} = \mathsf{true}$, otherwise |

We use assumption extraction to give the formal definition of safety.

**Definition 2.1** (Safety)**.** A closed process $P$ is *safe* iff whenever $P \rightarrow^* P'$ and $P' \equiv (\mathsf{new}\ \widetilde{a} : \widetilde{T})\, (\mathsf{assert}\ C \mid Q)$ we have that $\models \exists \widetilde{a}.\, \overline{Q} \Rightarrow C$.

Intuitively, a process is robustly safe if it is safe when run in parallel with an arbitrary opponent [GJ03].

**Definition 2.2** (Opponent)**.** A closed process is an *opponent* iff it does not contain any assert, all restrictions occurring therein are annotated with Un[3], and the cardinalities of the created zero-knowledge proofs match the ones declared in the corresponding statements.

[3] Fournet et al. [FGM07a] call such a mono-typed process Un-typed. Following their and other previous work [GJ03, GJ04] we will set up the type system so that there is no difference between Un-typed and untyped opponents.

These three conditions on opponents are very mild, since the type annotations and asserts do not affect the behavior of processes[4]. Moreover, asserts are meant to be used for marking the expectations of the protocol designer, and if opponents were allowed to contain asserts then no protocol would be considered secure because the process assert false would also be an opponent. The third condition ensures that the attacker only creates well-formed terms, in which the cardinality constraints on the zero-knowledge constructors are respected.

**Definition 2.3** (Robust Safety)**.** A closed process $P$ is *robustly safe* iff $P \mid O$ is safe for every opponent $O$.

As we will later see, our type system guarantees that if a process is well-typed, then it is robustly safe. Please note that robust safety does not require opponents to be well-typed a priori, but instead the type system will be designed so that we can prove as a property of the type system that all opponents are indeed well-typed.

## 2.4. Type System for Zero-knowledge

This section presents our type system for analyzing protocols based on non-interactive zero-knowledge proofs. The type system builds upon previous work on authentication by typing [GJ03, GJ04], on statically enforcing authorization policies on distributed systems [FGM07a], and on refinement types for security [BBF+08]. The main technical novelties of the type system include: the usage of union and intersection types in a security context, the static reasoning about type disjointness, the sound and complete logical characterization of when a type is compromised, the very precise subtyping rules for asymmetric cryptography, the typing rule for the equality destructor that uses intersection types and type disjointness, the subtyping rule that makes trivially empty types subtypes of all the other types, and the very simple typing rule for destructor evaluations. We believe that these technical improvements over the existing type-based analyses for security protocols are useful in general, not only for analyzing protocols based on zero-knowledge proofs.

The remainder of this section is organized as follows: §2.4.1 presents the types in our system; §2.4.2 introduces typing environments and gives an overview of the judgments of our type system; §2.4.3 defines the formula entailment judgment; §2.4.4 discusses subtyping and kinding; §2.4.5 introduces our novel logical characterization of kinding; §2.4.6 presents our encoding of type Private, a type that is disjoint from Un, and then introduces our general technique for statically reasoning about type disjointness; §2.4.7 defines the term and destructor typing relation; §2.4.8 presents the typing relation for processes; finally §2.4.9 presents our solution for typing processes that verify zero-knowledge proofs.

[4] For any well-formed process we can obtain an observationally equivalent opponent simply by removing all asserts and changing the typing annotations to Un.

### 2.4.1. Types

The top type $\top$ is the supertype of all the other types, any well-typed term also has type $\top$ by subtyping. The intersection type $T \wedge U$ is the type of all terms that have both type $T$ and type $U$ [Pie97]. Conversely, the union type $T \vee U$ is the type of all terms that have type $T$ or type $U$, maybe both, and we do not necessarily know which one of them it is [Pie91]. We use union and intersection types as well as the top type for inferring very precise type information about the arguments of zero-knowledge proofs (see §2.4.9). These types are, however, also useful on their own. For instance a symmetric key that is used to sign terms of type $T$ as well as terms of type $U$ can be given type $\mathsf{SymKey}(T \vee U)$.

**Types**

| $T, U, V ::=$ | types |
|---|---|
| $\top$ | any well-typed term |
| $T \wedge U$ | term that has both type $T$ and type $U$ |
| $T \vee U$ | term that has type $T$ or type $U$ |
| $\{x : T \mid C\}$ | term of type $T$ that fulfills $C$ (scope of $x$ is $C$) |
| $\mathsf{Un}$ | term that is possibly known to the attacker |
| $\mathsf{Ch}(T)$ | channel conveying messages of type $T$ |
| $\mathsf{Pair}(x{:}T, U)$ | dependent pair type (scope of $x$ is $U$) |
| $\mathsf{SigKey}(T)$ | signing key that only signs terms of type $T$ |
| $\mathsf{VerKey}(T)$ | verification key for a signing key of type $\mathsf{SigKey}(T)$ |
| $\mathsf{Signed}(T)$ | signature done using a key of type $\mathsf{SigKey}(T)$ |
| $\mathsf{PubEnc}(T)$ | public-key encryption of term of type $T$ |
| $\mathsf{DecKey}(T)$ | key that only decrypts terms of type $\mathsf{PubEnc}(T)$ |
| $\mathsf{EncKey}(T)$ | encryption key for decryption key of type $\mathsf{DecKey}(T)$ |
| $\mathsf{Hash}(T)$ | hash of term of type $T$ |
| $\mathsf{SymKey}(T)$ | symmetric key that only encrypts terms of type $T$ |
| $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\ \exists \widetilde{x}.\, C)$ | zero-knowledge proof for statement $S$ (scope of $\widetilde{x}$ and $\widetilde{y}$ is $C$ and $\widetilde{x}, \widetilde{y}$ pairwise distinct) |

**Notation:**
We call a type $T$ generative iff $T \in \{\mathsf{Un}, \mathsf{Ch}(U), \mathsf{SigKey}(U), \mathsf{DecKey}(U), \mathsf{SymKey}(U)\}$

Let $\widetilde{x} : \widetilde{T}$ denote $x_1 : T_1, \ldots, x_n : T_n$, and $\exists \widetilde{x}.\, C$ denote $\exists x_1. \ldots \exists x_n.\, C$, for some $n$.

As proposed by Bengtson et al. [BBF+08] we use refinement types to associate logical formulas to terms: the refinement type $\{x : T \mid C\}$ can be given to a term $M$ if $M$ has type $T$ and if additionally the formula $C\{M/x\}$ is entailed by the typing environment. For instance the type $\{x_f : \mathsf{Private} \mid \mathit{OkTPM}(x_f)\}$ from §2.2 contains all terms of type $\mathsf{Private}$ for which the predicate $\mathit{OkTPM}$ holds. Pairs are given dependent types of the form $\mathsf{Pair}(x{:}T, U)$, where the type $U$ of the second component of the pair can depend on the term $x$ in the first component; for non-dependent pair types we omit the unused

variable and write $\mathsf{Pair}(T, U)$. Dependent pair types are used together with refinement types to more easily associate formulas to tuples encoded as nested pairs [FGM07a]. Channels carrying terms of type $T$ are given type $\mathsf{Ch}(T)$. More precisely, our type system enforces that only messages of type $T$ are sent over the channel, and it can therefore guarantee that the received messages are also of type $T$.

The untrusted type $\mathsf{Un}$ contains all terms possibly known to the attacker, i.e., our type system enforces that opponents only have access to data of type $\mathsf{Un}$. By a property called opponent typability [GJ04] (Lemma 2.33 in §2.5.8), any process that does not include asserts and whose names are all annotated with $\mathsf{Un}$ (i.e., any opponent as defined in §2.3.6) is well-typed in our system. This is possible because the subtyping relation (see §2.4.4) equates types like $\mathsf{Pair}(\mathsf{Un}, \mathsf{Un})$, $\mathsf{Ch}(\mathsf{Un})$, etc. to $\mathsf{Un}$. The main benefit of opponent typability is that robust safety, the main property enforced by the type system and defined in §2.3.6, does not depend in any way on the rules of the type system itself.

Additionally, we consider types for the different cryptographic primitives. For digital signatures, $\mathsf{SigKey}(T)$ denotes the type of signing keys that can only be used to sign terms of type $T$ (i.e., our type system enforces this restriction), $\mathsf{VerKey}(T)$ is the type of verification keys corresponding to signing keys of type $\mathsf{SigKey}(T)$, while $\mathsf{Signed}(T)$ is the type of signatures done with keys of type $\mathsf{SigKey}(T)$. For public-key encryption, $\mathsf{PubEnc}(T)$ is the type of encryptions of terms of type $T$, $\mathsf{DecKey}(T)$ is the type of keys that can only be used to decrypt terms of type $\mathsf{PubEnc}(T)$, and $\mathsf{EncKey}(T)$ is the type of public encryption keys corresponding to decryption keys of type $\mathsf{DecKey}(T)$. Finally, the type $\mathsf{SymKey}(T)$ is the type of symmetric keys that can only be used to encrypt terms of type $T$, while $\mathsf{Hash}(T)$ is the type of hashes of terms of type $T$. In the types above $T$ is often a refinement type conveying a logical formula. For instance, $\mathsf{SigKey}(\{x : \mathsf{Private} \mid \mathit{OkTPM}(x)\})$ is the type of keys that can only be used to sign private terms for which we know that the $\mathit{OkTPM}$ predicate holds.

The zero-knowledge proof type $\mathsf{ZKProof}_S(y_1 : T_1, \ldots, y_m : T_m; \exists x_1, \ldots, x_n.\, C)$ contains terms of the form $\mathsf{zk}_S(\widetilde{N}; \widetilde{M})$. This dependent type lists the types of the public arguments $y_1, \ldots, y_m$, and contains an authorization logic formula $C$ that is conveyed by the proof, where the witnesses kept secret by the proof $x_1, \ldots, x_n$ are existentially quantified. The type system preserves the invariant that the public arguments $\widetilde{M}$ can be given the types $\widetilde{T}$, and that the formula $C\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}$ is entailed by the typing environment.

Finally, we call a type $T$ generative if it is $\mathsf{Un}$, a channel type, a signing key type, a private decryption key type, or a symmetric key type. The type system enforces that only generative types can annotate name restrictions; see rule (Proc New) in §2.4.8. Moreover, in our current system the closed terms of all the generative types are necessarily names.

### 2.4.2. Typing Environments and Judgments

A typing environment $E$ is a list of name, variable and predicate bindings, and it can also contain logical formulas. The predicate bindings are needed for ensuring the well-formedness of authorization logic formulas involving second-order quantifiers, which are introduced by our logical characterization of kinding (see §2.4.5).

**Syntax of typing environments**

| | |
|---|---|
| $\mu ::=$ | environment entry |
| $a : T$ | name binding |
| $x : T$ | variable binding |
| $p$ | predicate binding |
| $\{C\}$ | authorization logic formula |
| $E ::= \mu_1, \ldots, \mu_n$ | typing environment is a list of environment entries |

We group the judgments of our type system into two: well-formedness judgments and typing judgments.

**Well-formedness judgments**

| | |
|---|---|
| $\vdash E\ ok$ | well-formed environment |
| $E \vdash C\ ok$ | well-formed formula |
| $E \vdash T\ ok$ | well-formed type |
| $E \vdash M\ ok$ | well-formed term |
| $E \vdash D\ ok$ | well-formed destructor |
| $E \vdash B\ ok$ | well-formed statement formula |
| $E \vdash S\ ok$ | well-formed statement |
| $E \vdash P\ ok$ | well-formed process |
| $E \vdash P\ opp$ | well-formed possibly-open opponent process |

An environment is well-formed if no name, variable or predicate is bound more than once, and if the types and formulas occurring in the environment entries are well-formed in the corresponding prefix of the environment. Additionally, a well-formed environment only associates generative types to the names. A syntactic phrase $\phi$ (formula, type, term, destructor, statement, or process) is well-formed in environment $E$ if all free names, variables and predicates in $\phi$ are bound in $E$ and $E$ is well-formed. Additionally, a zero-knowledge proof type $\mathsf{ZKProof}_S(y_1 : T_1, \ldots, y_{m'} : T_{m'}; \exists x_1, \ldots, x_{n'}.\, C)$ is well-formed, where $S = \mathsf{witness}\ x_1, \ldots, x_n\ \mathsf{public}\ y_1, \ldots, y_m\ \mathsf{in}\ B$, only if $n = n'$ and $m = m'$. Similar cardinality conditions are required for zero-knowledge terms: the term $\mathsf{zk}_S(N_1, \ldots, N_{m'}; M_1, \ldots, M_{n'})$ is well-formed, where $S = \mathsf{witness}\ x_1, \ldots, x_n\ \mathsf{public}\ y_1, \ldots, y_m\ \mathsf{in}\ B$, only if $n = n'$ and $m = m'$.[5]

[5] In previous presentations of this work these cardinality conditions were left implicit. Proving things formally required us to make them explicit, by adding them to the well-formedness judgments.

Finally, we generalize the notion of opponent to open processes, by defining an inductive judgment $E \vdash P\ opp$. The original definition from §2.3.6 is equivalent to $\emptyset \vdash P\ opp$. The more general judgment $E \vdash P\ opp$ is useful for proving properties such as Lemma 2.33 (Opponent Typability) by rule induction.

The typing judgments of our type system are listed below and explained in the following sections.

**Typing judgments**

| | |
|---|---|
| $E \vdash C$ | environment $E$ entails formula $C$ |
| $E \vdash T :: k$ | kinding, $k \in \{\mathsf{pub}, \mathsf{tnt}\}$ |
| $E \vdash T <: U$ | subtyping |
| $E \vdash T \between U \rightsquigarrow C$ | non-disjointness, if $T$ and $U$ intersect in $E$ then $E \vdash C$ |
| $E \vdash M : T$ | term typing |
| $E \vdash D : T$ | destructor typing |
| $E \vdash P$ | process typing |

### 2.4.3. Formula Entailment Judgment

The formula entailment judgment $E \vdash C$ enforces that $C$ is well-formed in the environment $E$ and additionally that the formulas extracted from $E$ entail $C$ in the authorization logic. Intuitively, our type system ensures that whenever $E \vdash C$ holds the formula $C$ is entailed at execution time from the the active assumes. This entailment judgment is used for instance when type-checking the process $\mathsf{assert}\ C$ using rule (Proc Assert): type-checking succeeds only if $E \vdash C$ holds, which ensures that the assert will succeed at execution time, as needed for safety.

**Entailed formula:** $E \vdash C$

(Entailed)
$$\frac{E \vdash C\ ok \quad forms(E) \models C}{E \vdash C}$$

**Definition:**
$forms_x(\{y : T \mid C\}) = forms_x(T), C\{x/y\}$
$forms_x(T_1 \wedge T_2) = forms_x(T_1), forms_x(T_2)$
$forms_x(T_1 \vee T_2) = [C_1 \vee C_2 \mid C_1 \in forms_x(T_1), C_2 \in forms_x(T_2)]$
$forms_x(\mathsf{ZKProof}_S(y_1{:}T_1, .., y_m{:}T_m;\ \exists \widetilde{x}.\, C)) = \exists y_1, \ldots, y_m.\, (\bigwedge_{i \in 1 \ldots m} forms_{y_i}(T_i) \wedge \exists \widetilde{x}.\, C)$
$forms_x(T) = \emptyset$, otherwise

$forms(E, x : T) = forms(E), forms_x(T)$
$forms(E, \{C\}) = forms(E), C$
$forms(E, \mu) = forms(E)$, if $\mu$ not of the form $x : T$ or $\{C\}$
$forms(\emptyset) = \emptyset$

The function $\mathit{forms}(E)$ inspects the entries in $E$ and gathers the top-level formulas. Formulas from entries of the form $\{C\}$ are directly copied. Entries of the form $x : T$ produce formulas using the auxiliary function $\mathit{forms}_x(T)$. For refinement types $\{y : T \mid C\}$ we recurse into the refined type $T$, and we additionally gather the formula $C$ after substituting $y$ with the actual binding $x$. For intersection types $T_1 \wedge T_2$ we just put together the formulas from $T_1$ and $T_2$; concatenation on the left side of a $\models$ is interpreted conjunctively. For union types $T_1 \vee T_2$ we take all the pair-wise disjunctions of the formulas extracted from $T_1$ and from $T_2$. Gathering formulas from intersection and union types ensures the type system has enough precision to deal with the types that are inferred for the arguments of zero-knowledge proofs (see §2.4.9). Finally, as a proof of concept, we show that existentially quantified formulas can be gathered even from zero-knowledge proof types. In practice, however, most zero-knowledge proofs are received from untrusted channels at type $\mathsf{Un}$, which allows no useful formula to be gathered.

### 2.4.4. Subtyping and Kinding

Our type system has a *subtyping* relation on types and allows a term of a subtype to be used in all contexts that require a term of a supertype. Subtyping increases the flexibility of the type system, since it allows more correct programs to be accepted as well-typed. For instance, this allows one to send a message of type $\{x : T \mid C\}$ over a channel of type $\mathsf{Ch}(T)$ and thus "forget" the constraint $C$, since $\{x : T \mid C\}$ is a subtype of $T$.

Following Gordon and Jeffrey [GJ03], subtyping plays two additional very important roles in our type system. First, subtyping allows us to compare types with type $\mathsf{Un}$: if $T$ is a subtype of $\mathsf{Un}$ then the terms in $T$ are allowed to flow to the opponent, and we call type $T$ public; on the other hand, if $\mathsf{Un}$ is a subtype of $T$ then every term coming from the opponent can be given type $T$, and we call type $T$ tainted. We define a separate *kinding* judgment $E \vdash T :: k$ where $k \in \{\mathsf{pub}, \mathsf{tnt}\}$ that captures this role and contributes to the subtyping relation via the (Sub Pub Tnt) rule. This rule makes every public type a subtype of every tainted type. Since by rule (Kind Un) type $\mathsf{Un}$ is itself both public and tainted, we immediately obtain that $E \vdash T :: \mathsf{pub}$ implies $E \vdash T <: \mathsf{Un}$, and that $E \vdash T :: \mathsf{tnt}$ implies $E \vdash \mathsf{Un} <: T$; the reverse direction of these two implications also holds but the proofs are more involved (see Lemma 2.16 and Lemma 2.17 in §2.5.2).

Second, subtyping equates type $\mathsf{Un}$ with each of the following types: $\mathsf{Ch}(\mathsf{Un})$, $\mathsf{Pair}(\mathsf{Un}, \mathsf{Un})$, $\mathsf{SigKey}(\mathsf{Un})$, $\mathsf{VerKey}(\mathsf{Un})$, $\mathsf{Signed}(\mathsf{Un})$, $\mathsf{PubEnc}(\mathsf{Un})$, $\mathsf{DecKey}(\mathsf{Un})$, $\mathsf{EncKey}(\mathsf{Un})$, $\mathsf{Hash}(\mathsf{Un})$, $\mathsf{SymKey}(\mathsf{Un})$, and $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{\mathsf{Un}}; \exists \widetilde{x}.\, \mathsf{true})$. This "universal type" property of $\mathsf{Un}$ allows us to prove that any opponent is well-typed, which is crucial for showing robust safety by typing.

The rules of the kinding judgment are listed below. As mentioned above type $\mathsf{Un}$ is both public and tainted. The top type $\top$ is tainted, since it is always safe to give any well-typed term type $\top$, including the terms coming from the attacker, which are well-typed at type $\mathsf{Un}$. Type $\top$ is however not public, since if that was the case any well-typed term

could be sent to the attacker, which would allow protocol participants to leak any secret. The rule (Kind Empty Pub) is new to our type system, and allows trivially empty types to be considered public, since this is harmless and actually useful. For instance, this rule makes the refinement type $\{x : T \mid \mathsf{false}\}$ public for any well-formed type $T$, and it allows us to regard type $\top$ as public in an inconsistent environment. The latter is important for proving an important property of our type system, which states that in an inconsistent environment $E$, for which $E \vdash \mathsf{false}$ holds, all types are both public and tainted, i.e., equivalent by subtyping to $\mathsf{Un}$. This is necessary for expressing the precise conditions under which a type is compromised as a logical formula (see §2.4.5), and it also allows us to prove that in an inconsistent environment any well-formed term has any well-formed type, by an argument similar to opponent typability (see Lemma 2.8 in §2.5.1).

An intersection type $T_1 \wedge T_2$ is public if $T_1$ is public or $T_2$ is public, and is tainted if both $T_1$ and $T_2$ are tainted. A term has type $T_1 \wedge T_2$ if and only if it has both type $T_1$ and type $T_2$. Intuitively, if we are given a term of type $T_1 \wedge T_2$ we can consider it at either type, so it is enough if one of them allows us to send it over the untrusted network. Conversely, if we receive a term from the untrusted network, in order to be able to give it type $T_1 \wedge T_2$ we need to give it both type $T_1$ and type $T_2$, which means that both $T_1$ and $T_2$ need to be tainted. The kinding rules for union types are exactly the dual of the ones for intersection types.

The refinement type $\{x : T \mid C\}$ is public if $T$ is public. The type $\{x : T \mid C\}$ is a subtype of $T$, so, intuitively, if it is safe to send terms of type $T$ to the attacker, then it is also safe to send such terms to the attacker when they additionally fulfill condition $C$. For considering $\{x : T \mid C\}$ tainted we need to be much more restrictive though: we require that $T$ is tainted and that $E, x : T \vdash C$ holds. The attacker is not required to assume any predicate, so if we want to give terms coming from the attacker a refinement type $\{x : T \mid C\}$, then $C$ needs to hold for every $x$ for which $\mathit{forms}_x(T)$ holds.

If we want to send a channel of type $\mathsf{Ch}(T)$ to the attacker, or give type $\mathsf{Ch}(T)$ to a channel received from the attacker, then $T$ needs to be both public and tainted, i.e., equivalent to $\mathsf{Un}$. Protocol participants can send terms of type $T$ over this channel, and if the attacker has access to the channel it can get these terms. Also, the code of the protocol expects to receive only terms of type $T$ from this channel, but if the attacker has access to the channel it can send any term it can construct over the channel. So the channel type $\mathsf{Ch}(T)$ is public or tainted, if $T$ is both public and tainted. In the same way, the type of symmetric keys $\mathsf{SymKey}(T)$ is public or tainted, if $T$ is both public and tainted. Symmetric encryption produces terms of type $\mathsf{Un}$, which can be safely sent over the untrusted network, and symmetric decryption takes terms of type $\mathsf{Un}$ as argument, which can come from the untrusted network, so a shared symmetric key allows one to establish a "private" channel over an untrusted one. It should thus be no surprise that there is no difference in terms of kinding between "private" channels and symmetric keys.

The kinding rules for asymmetric cryptography are much more subtle. The rules we consider are novel and much more precise than in previous work. For instance Fournet et al. [FGM07a] only consider signatures, their kinding rules do not always allow verification keys to be made public, and their kinding rules for signing keys are as restrictive as for symmetric keys. This prevents their type system from supporting common cryptographic patterns (e.g., for sign-then-encrypt their type system does not allow the verification key to be made public). In our type system, first of all, asymmetric encryption keys and verification keys can always be made public, since requiring them to be kept secret would not increase the security of the protocols while neutralizing the advantages asymmetric cryptography[6] has over symmetric cryptography (e.g., the ability to authenticate over an untrusted network without sharing any secret in advance).

Second, since encryption hides the payload term we allow public-key encryptions to be sent over the untrusted network, and since it is the successful signature verification that proves the authenticity of the payload term we allow terms that come from the untrusted network to be verified as signatures. So the type of public-key encryptions $\mathsf{PubEnc}(T)$ is always public, while the type of signatures $\mathsf{Signed}(T)$ is always tainted. On the other hand, signatures in our calculus contain the term that was signed, and verifying the signature with the public verification key returns this message in clear. This means that signing a term does not protect its secrecy, so a signature type $\mathsf{Signed}(T)$ is public only if type $T$ is public. Conversely, successfully decrypting a term does not necessarily authenticate it, since the attacker can also encrypt terms using public keys, so if the ciphertext comes from the attacker we cannot justify giving the result an untainted type. To prevent honest participants from decrypting messages from the network if the result of the decryption would be untainted, the decryption constructor only works on ciphertexts of type $\mathsf{PubEnc}(T)$ and we make sure that an encryption type $\mathsf{PubEnc}(T)$ is tainted, only when $T$ is tainted.

Third, our type system enforces that the verification keys and public encryption keys used by the protocols are authentic (not tainted), while the signing keys and decryption keys are both authentic and secret. However, these restrictions are necessary only when these asymmetric keys are used to protect the security of the system, and signing can only protect authenticity, while public-key encryption can only protect secrecy. So if a protocol is signing terms of a tainted type $T$ then the type of the signing key $\mathsf{SigKey}(T)$ and the type of the corresponding verification key $\mathsf{VerKey}(T)$ can be allowed to be both public and tainted, and thus equivalent to $\mathsf{Un}$. So intuitively, if there is no authenticity to protect for the terms being signed with a certain signing key, we do not need to enforce any restriction on the usage of this signing key or of the corresponding verification key. Conversely, if a protocol is using public-key encryption on terms having a public type $T$, we do not need to enforce any restriction on the corresponding encryption and decryption keys, so the types $\mathsf{EncKey}(T)$ and $\mathsf{DecKey}(T)$ are in this case both public and tainted.[7]

[6] Asymmetric cryptography is often called *public*-key cryptography.

[7] This last set of rules might seem unnecessarily permissive, but please note that the goal of this type system is to reject insecure protocols, not to enforce one particular way to design protocols. These rules could for instance be used to type-check protocols where certain information has to be kept

Finally, a zero-knowledge proof type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\ \exists \widetilde{x}.\, C)$ is public if and only if the types of all public arguments of the proof $\widetilde{T}$ are public. This is necessary, since an attacker that obtains a zero knowledge proof can extract the public arguments using the public destructor. Conversely, a type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\ \exists \widetilde{x}.\, C)$ is tainted if and only if the types of the public arguments of the proof $\widetilde{T}$ are tainted and additionally $E \vdash \forall \widetilde{x}.\, \forall \widetilde{y}.\, C$. If a zero-knowledge term $\mathsf{zk}_S(\widetilde{N};\ \widetilde{M})$ has type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\ \exists \widetilde{x}.\, C)$ then our type system guarantees that the terms $\widetilde{M}$ can be given the types $\widetilde{T}$ and the formula $C\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}$ is entailed by the current environment. So in order to allow giving a term received from the attacker-controlled network type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\ \exists \widetilde{x}.\, C)$ we need to ensure that the invariant above is not violated, i.e., asking that the types $T_1, \ldots, T_m$ are tainted allows us to give any term coming from the attacker type $T_i$, while asking that formula $C$ holds for all $\widetilde{x}$ and $\widetilde{y}$ allows us to instantiate $C$ for any terms the attacker might send.

**Kinding:** $E \vdash T :: k$ **for** $k \in \{\mathsf{pub}, \mathsf{tnt}\}$

$$\frac{\vdash E\ ok}{E \vdash \mathsf{Un} :: k}\ (\text{Kind Un}) \qquad \frac{\vdash E\ ok}{E \vdash \top :: \mathsf{tnt}}\ (\text{Kind Top Tnt}) \qquad \frac{E, x : T \vdash \mathsf{false}}{E \vdash T :: \mathsf{pub}}\ (\text{Kind Empty Pub})$$

$$\frac{E \vdash T :: \mathsf{pub} \quad E \vdash U\ ok}{E \vdash T \wedge U :: \mathsf{pub}}\ (\text{Kind And Pub 1}) \qquad \frac{E \vdash U :: \mathsf{pub} \quad E \vdash T\ ok}{E \vdash T \wedge U :: \mathsf{pub}}\ (\text{Kind And Pub 2}) \qquad \frac{E \vdash T :: \mathsf{tnt} \quad E \vdash U :: \mathsf{tnt}}{E \vdash T \wedge U :: \mathsf{tnt}}\ (\text{Kind And Tnt})$$

$$\frac{E \vdash T :: \mathsf{pub} \quad E \vdash U :: \mathsf{pub}}{E \vdash T \vee U :: \mathsf{pub}}\ (\text{Kind Or Pub}) \qquad \frac{E \vdash T :: \mathsf{tnt} \quad E \vdash U\ ok}{E \vdash T \vee U :: \mathsf{tnt}}\ (\text{Kind Or Tnt 1}) \qquad \frac{E \vdash U :: \mathsf{tnt} \quad E \vdash T\ ok}{E \vdash T \vee U :: \mathsf{tnt}}\ (\text{Kind Or Tnt 2})$$

$$\frac{E \vdash T :: \mathsf{pub} \quad E, x : T \vdash C\ ok}{E \vdash \{x : T \mid C\} :: \mathsf{pub}}\ (\text{Kind Refine Pub}) \qquad \frac{E \vdash T :: \mathsf{tnt} \quad E, x : T \vdash C}{E \vdash \{x : T \mid C\} :: \mathsf{tnt}}\ (\text{Kind Refine Tnt}) \qquad \frac{E \vdash T :: \mathsf{pub} \quad E \vdash T :: \mathsf{tnt}}{E \vdash \mathsf{Ch}(T) :: k}\ (\text{Kind Chan})$$

$$\frac{E \vdash T :: k \quad E, x : T \vdash U :: k}{E \vdash \mathsf{Pair}(x{:}T, U) :: k}\ (\text{Kind Pair}) \qquad \frac{E \vdash T :: \mathsf{pub} \quad E \vdash T :: \mathsf{tnt}}{E \vdash \mathsf{SymKey}(T) :: k}\ (\text{Kind SymKey}) \qquad \frac{E \vdash T :: \mathsf{pub}}{E \vdash \mathsf{Signed}(T) :: \mathsf{pub}}\ (\text{Kind Signed Pub})$$

$$\frac{E \vdash T\ ok}{E \vdash \mathsf{Signed}(T) :: \mathsf{tnt}}\ (\text{Kind Signed Tnt}) \qquad \frac{E \vdash T :: \mathsf{tnt}}{E \vdash \mathsf{SigKey}(T) :: k}\ (\text{Kind SigKey}) \qquad \frac{E \vdash T\ ok}{E \vdash \mathsf{VerKey}(T) :: \mathsf{pub}}\ (\text{Kind VerKey Pub}) \qquad \frac{E \vdash T :: \mathsf{tnt}}{E \vdash \mathsf{VerKey}(T) :: \mathsf{tnt}}\ (\text{Kind VerKey Tnt})$$

$$\frac{E \vdash T\ ok}{E \vdash \mathsf{PubEnc}(T) :: \mathsf{pub}}\ (\text{Kind PubEnc Pub}) \qquad \frac{E \vdash T :: \mathsf{tnt}}{E \vdash \mathsf{PubEnc}(T) :: \mathsf{tnt}}\ (\text{Kind PubEnc Tnt}) \qquad \frac{E \vdash T\ ok}{E \vdash \mathsf{EncKey}(T) :: \mathsf{pub}}\ (\text{Kind EncKey Pub})$$

secret for a period of time, but needs to be disclosed in a verifiable way after a certain declassification event occurs. If all the information in the protocol is disclosed at the end then a very simple way to achieve verifiability is to also publish the keys that were previously used to protect it.

$$\frac{E \vdash T :: \mathsf{pub}}{E \vdash \mathsf{EncKey}(T) :: \mathsf{tnt}}\ (\text{Kind EncKey Tnt}) \qquad \frac{E \vdash T :: \mathsf{pub}}{E \vdash \mathsf{DecKey}(T) :: k}\ (\text{Kind DecKey}) \qquad \frac{E \vdash T\ ok}{E \vdash \mathsf{Hash}(T) :: \mathsf{pub}}\ (\text{Kind Hash Pub}) \qquad \frac{E \vdash T :: \mathsf{tnt}}{E \vdash \mathsf{Hash}(T) :: \mathsf{tnt}}\ (\text{Kind Hash Tnt})$$

$$\frac{E \vdash \widetilde{T} :: \mathsf{pub} \quad E, \widetilde{y} : \widetilde{\top}, \widetilde{x} : \widetilde{\top} \vdash C\ ok}{E \vdash \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\ \widetilde{\exists x}.\, C) :: \mathsf{pub}}\ (\text{Kind ZK Pub}) \qquad \frac{E \vdash \widetilde{T} :: \mathsf{tnt} \quad E, \widetilde{y} : \widetilde{\top}, \widetilde{x} : \widetilde{\top} \vdash C}{E \vdash \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\ \widetilde{\exists x}.\, C) :: \mathsf{tnt}}\ (\text{Kind ZK Tnt})$$

We explain the most important rules of the subtyping judgment below.

**Subtyping:** $E \vdash T <: U$

$$\frac{E \vdash T :: \mathsf{pub} \quad E \vdash U :: \mathsf{tnt}}{E \vdash T <: U}\ (\text{Sub Pub Tnt}) \qquad \frac{E \vdash T\ ok}{E \vdash T <: \top}\ (\text{Sub Top}) \qquad \frac{E, x : T \vdash \mathsf{false} \quad E \vdash T'\ ok}{E \vdash T <: T'}\ (\text{Sub Empty})$$

$$\frac{E \vdash T <: T' \quad E \vdash U\ ok}{E \vdash T \wedge U <: T'}\ (\text{Sub And LB 1}) \qquad \frac{E \vdash U <: U' \quad E \vdash T\ ok}{E \vdash T \wedge U <: U'}\ (\text{Sub And LB 2}) \qquad \frac{E \vdash T <: T_1 \quad E \vdash T <: T_2}{E \vdash T <: T_1 \wedge T_2}\ (\text{Sub And Greatest})$$

$$\frac{E \vdash T_1 <: T \quad E \vdash T_2 <: T}{E \vdash T_1 \vee T_2 <: T}\ (\text{Sub Or Least}) \qquad \frac{E \vdash T' <: T \quad E \vdash U\ ok}{E \vdash T' <: T \vee U}\ (\text{Sub Or UB 1}) \qquad \frac{E \vdash U' <: U \quad E \vdash T\ ok}{E \vdash U' <: T \vee U}\ (\text{Sub Or UB 2})$$

$$\frac{E \vdash T <: T' \quad E, x : T \vdash C\ ok}{E \vdash \{x : T \mid C\} <: T'}\ (\text{Sub Refine Left}) \qquad \frac{E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}\ (\text{Sub Refine Right}) \qquad \frac{E \vdash T <:> U}{E \vdash \mathsf{Ch}(T) <: \mathsf{Ch}(U)}\ (\text{Sub Chan Inv})$$

$$\frac{E \vdash T_1 <: U_1 \quad E, x : T_1 \vdash T_2 <: U_2}{E \vdash \mathsf{Pair}(x{:}T_1, T_2) <: \mathsf{Pair}(x{:}U_1, U_2)}\ (\text{Sub Pair Cov}) \qquad \frac{E \vdash T <:> U}{E \vdash \mathsf{SymKey}(T) <: \mathsf{SymKey}(U)}\ (\text{Sub SymKey Inv})$$

$$\frac{E \vdash T <:> U}{E \vdash \mathsf{Signed}(T) <: \mathsf{Signed}(U)}\ (\text{Sub Signed Inv}) \qquad \frac{E \vdash T <:> U}{E \vdash \mathsf{SigKey}(T) <: \mathsf{SigKey}(U)}\ (\text{Sub SigKey Inv})$$

$$\frac{E \vdash T <: U}{E \vdash \mathsf{VerKey}(T) <: \mathsf{VerKey}(U)}\ (\text{Sub VerKey Cov}) \qquad \frac{E \vdash T <:> U}{E \vdash \mathsf{PubEnc}(T) <: \mathsf{PubEnc}(U)}\ (\text{Sub PubEnc Inv})$$

$$\frac{E \vdash U <: T}{E \vdash \mathsf{EncKey}(T) <: \mathsf{EncKey}(U)}\ (\text{Sub EncKey Con}) \qquad \frac{E \vdash T <:> U}{E \vdash \mathsf{DecKey}(T) <: \mathsf{DecKey}(U)}\ (\text{Sub DecKey Inv})$$

$$\frac{E \vdash T <:> U}{E \vdash \mathsf{Hash}(T) <: \mathsf{Hash}(U)}\ (\text{Sub Hash Inv}) \qquad \frac{E \vdash T_i <: U_i \quad \forall i \in 1 \ldots m \qquad E, \widetilde{y} : \widetilde{\top}, \widetilde{x} : \widetilde{\top}, \{C\} \vdash C'}{E \vdash \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\ \widetilde{\exists x}.\, C) <: \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{U};\ \widetilde{\exists x}.\, C')}\ (\text{Sub ZK})$$

**Notation:** We write $E \vdash T <:> U$ to denote $E \vdash T <: U$ and $E \vdash U <: T$.

As indicated above, the rule (Sub Pub Tnt) connects subtyping with kinding, by making every public type a subtype of every tainted one. The rule (Sub Top) makes type $\top$ the supertype of all other well-formed types. Similarly to (Kind Empty Pub), the rule (Sub Empty) allows trivially empty types to be considered a subtype of any other type. In particular this makes any refinement type $\{x : T \mid \mathsf{false}\}$ a subtype of all the other types (bottom). The rule (Sub Empty) needs however to be phrased in this more general way, which also takes union and intersection types into account, since otherwise the subtyping relation would not be transitive (e.g., since the intersection type $\bot \wedge U$ is a subtype of $\bot$ and $\bot$ is a subtype of an arbitrary $T$, then we need to ensure that $\bot \wedge U$ is also a subtype of $T$).

As far as subtyping is concerned, the intersection type $T_1 \wedge T_2$ is a[8] 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[9] 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 $T_1 \wedge T_2$. The union type $T_1 \vee T_2$ is a least upper bound of $T_1$ and $T_2$. The rules for subtyping union types are exactly the dual of the ones for intersection types.

The refinement type $\{x : T \mid C\}$ is a subtype of $T$; the reason rule (Sub Refine Left) is more general is again to "bake-in" transitivity. This rule allows us to discard logical formulas when they are not needed, for instance, a term of type $\{x : \mathsf{Un} \mid \mathsf{Ok}(x)\}$ can be sent on a channel of type $\mathsf{Ch}(\mathsf{Un})$. Conversely, the type $T$ is a subtype of $\{x : T \mid C\}$ if $\forall x.\, \mathit{forms}_x(T) \Rightarrow C$ is entailed in the current typing environment (Sub Refine Right), so by subtyping we can only add universally valid formulas. This allows us to prove as a property that whenever $T$ is a subtype of $U$ the formulas in $T$ imply the formulas in $U$ (see Lemma 2.6 in §2.5.1). Using (Sub Refine Left) and (Sub Refine Right) we can derive a rule that directly compares two refinement types [BBF+08]: the type $\{x : T_1 \mid C_1\}$ is a subtype of $\{x : T_2 \mid C_2\}$ if $T_1$ is a subtype of $T_2$ and if the formula $\forall x.\, \mathit{forms}_x(T_1) \Rightarrow C_1 \Rightarrow C_2$ is entailed by the current environment.

Channel types and symmetric key types are invariant, i.e., have most restrictive subtyping in which $\mathsf{Ch}(T)$ is a subtype of $\mathsf{Ch}(U)$ if $T$ is equivalent by subtyping to $U$.[10]If channel types were covariant, we could take a channel of type $\mathsf{Ch}(T)$, change its type by subtyping to $\mathsf{Ch}(\top)$ and write any term on it, which would clearly be unsafe since the receivers expect to obtain messages of type $T$ from such channels. Conversely, if channel types were contravariant, we could take a channel of type $\mathsf{Ch}(\top)$, change its type by subtyping to $\mathsf{Ch}(T)$, and then give type $T$ to all received terms, although the senders can send anything over this channel.

[8]Our subtyping relation is not anti-symmetric, so least and greatest elements are not necessarily unique.
[9]Reflexivity and transitivity are properties of our definition of subtyping, which make it a preorder.
[10] Such type constructors are sometimes called nonvariant or "rigid" [AC96, OSV10].

By similar arguments many of the types for asymmetric cryptography need to be made invariant. There are two exceptions though: the type of verification keys can be allowed to be covariant while the type of public keys can be contravariant. Giving a verification key of type $\mathsf{VerKey}(T)$ also type $\mathsf{VerKey}(U)$ by subsumption for some $U$ that is a supertype of $T$ is acceptable, since this only allows us to give type $U$ to the result of the $\mathsf{check}$ destructors that use this key, which is anyway already possible by subtyping even if these results are first given the stronger type $T$. Conversely, giving a public key of type $\mathsf{EncKey}(U)$ also type $\mathsf{EncKey}(T)$ by subsumption for $T$ a subtype of $U$ is also sound, since this allows us to encrypt terms of type $T$ with this key, which is again already possible by first giving the terms type $U$ by (Term Subsum) and only then encrypting them.

Finally, a zero-knowledge type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.\, C)$ is a subtype of another zero-knowledge type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{U}; \exists \widetilde{x}.\, C')$ if the types of the public arguments $\widetilde{T}$ are subtypes of $\widetilde{U}$, and if additionally the formula $\forall \widetilde{x}.\, \forall \widetilde{y}.\, C \Rightarrow C'$ is entailed by the current environment (Sub ZK). As for refinement types, this last condition ensures that by subtyping we can only weaken the formula in a zero-knowledge type.

### 2.4.5. Logical Characterization of Kinding

We capture any instance of the kinding judgment as a formula in the authorization logic. Such a logical characterization of kinding is helpful for reasoning about the disjointness of types (§2.4.6), and for encoding type information that is conditioned on a type not being compromised (§2.4.9).

Our logical characterization of kinding is fully precise (i.e., sound and complete as shown in §2.5.3): We define a function $\mathsf{fkind}(E, T, k)$ that returns an authorization logic formula that is valid if and only if type $T$ has kind $k$ in environment $E$. We define $\mathsf{fkind}$ by recursion over the size of types. We can easily define $\mathsf{fkind}(E, \mathsf{Un}, \mathsf{tnt})$ as $\mathsf{true}$, $\mathsf{fkind}(E, \mathsf{SigKey}(T), \mathsf{tnt})$ recursively as $\mathsf{fkind}(E, T, \mathsf{tnt})$, and $\mathsf{fkind}(E, \mathsf{Ch}(T), \mathsf{tnt})$ recursively as $\mathsf{fkind}(E, T, \mathsf{pub}) \wedge \mathsf{fkind}(E, T, \mathsf{tnt})$. Not all cases are equally simple though. First, the kinding judgment is not fully syntax-directed because of rules such as (Kind Empty Pub), (Kind And Pub 1), (Kind And Pub 2), etc., so $\mathsf{fkind}$ needs to also use logical disjunction. Second, the rules (Kind Empty Pub), (Kind Refine Tnt), and (Kind ZK Tnt) have premises that are instances of the formula entailment judgment. For this we need to encode the entailment relation of the authorization logic into the authorization logic itself. This is generally not possible in first-order logic, which motivates the presence of the second-order quantifiers over predicates in our authorization logic.

We encode $E \vdash C$ as a formula using the function $\mathsf{fentails}(E, C)$, which is defined as $\forall dom(E).\, \bigwedge forms(E) \Rightarrow C$, where the $\forall dom(E)$ part universally closes all binders in $E$, including all the predicates. This is necessary, since otherwise different formulas generated by $\mathsf{fentails}$ would interact with each other, while in the kinding judgment all occurrences of $E \vdash C$ are completely independent. In other words, while in the entailment relation all free variables, names and predicates are implicitly universally

quantified, the fkind function needs to make this universal quantification explicit so that the formula it returns can be used as a part of a bigger formula containing conjunctions and disjunctions.

We illustrate the definition of $\forall dom(E).\ C$ on the environment we use to type-check the simplified DAA example from §2.2.

$$\begin{aligned}&\forall dom(Authenticate, OkTPM, Send, k_i : \mathsf{SigKey}(T_{ki}), Policy_{sdaa}, x_z : \mathsf{Un}).\ C\\ &\quad = \forall Authenticate.\forall OkTPM.\forall Send.\forall k_i.\forall x_z.\ C\end{aligned}$$

Second-order quantifiers are only used for obtaining a logical characterization of kinding that enjoys completeness; they appear neither in authorization policies, nor in the first-order logic proof obligations discharged by our type-checker. The second-order quantifiers introduced by the logical characterization of kinding only appear in positive positions, so in practice they can always be "extruded" after proper alpha-renaming, and removed once they reach the top-level since free predicates are implicitly universally quantified.

Finally, the kinding rules (Kind Empty Pub), (Kind Pair), (Kind Refine Pub), and (Kind Refine Tnt) have premises in which the environment is extended with a fresh variable. In the definition of fkind below we generate this fresh variable explicitly, using the pick_not_in function.

**Logical characterization of kinding:** $\mathsf{fkind}(E, T, k)$

**Definition:**

$\forall dom(E, x : T).\ C \triangleq \forall dom(E).\ \forall x.\ C$ $\quad$ $\forall dom(E, \{C'\}).\ C \triangleq \forall dom(E).\ C$

$\forall dom(E, p).\ C \triangleq \forall dom(E).\ \forall p.\ C$ $\quad$ $\forall dom(\emptyset).\ C \triangleq C$

$\forall dom(E, a : T).\ C \triangleq \forall dom(E).\ \forall a.\ C$

**Definition:**

$\mathsf{fentails}(E, C) \triangleq \forall dom(E).\ \bigwedge forms(E) \Rightarrow C$

**Definition:**

$\mathsf{fkind}(E, V, \mathsf{pub}) \triangleq$
  let $x = \mathsf{pick\_not\_in}(dom(E))$ in
  $\mathsf{fentails}((E, x : V), \mathsf{false}) \vee$
  (match $V$ with
    $\mid \top \Rightarrow \mathsf{fentails}(E, \mathsf{false})$
    $\mid T_1 \wedge T_2 \Rightarrow \mathsf{fkind}(E, T_1, \mathsf{pub}) \vee \mathsf{fkind}(E, T_2, \mathsf{pub})$
    $\mid T_1 \vee T_2 \Rightarrow \mathsf{fkind}(E, T_1, \mathsf{pub}) \wedge \mathsf{fkind}(E, T_2, \mathsf{pub})$
    $\mid \{x : T \mid C\} \Rightarrow \mathsf{fkind}(E, T, \mathsf{pub})$
    $\mid \mathsf{Un} \Rightarrow \mathsf{true}$
    $\mid \mathsf{Ch}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{pub}) \wedge \mathsf{fkind}(E, T, \mathsf{tnt})$
    $\mid \mathsf{Pair}(x{:}T, U) \Rightarrow$
      let $y = \mathsf{pick\_not\_in}(dom(E))$ in
      $\mathsf{fkind}(E, T, \mathsf{pub}) \wedge \mathsf{fkind}((E, y : T), U\{y/x\}, \mathsf{pub})$

$$
\begin{array}{l}
\quad \mid \mathsf{Signed}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{pub}) \\
\quad \mid \mathsf{SigKey}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{tnt}) \\
\quad \mid \mathsf{VerKey}(T) \Rightarrow \mathsf{true} \\
\quad \mid \mathsf{PubEnc}(T) \Rightarrow \mathsf{true} \\
\quad \mid \mathsf{EncKey}(T) \Rightarrow \mathsf{true} \\
\quad \mid \mathsf{DecKey}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{pub}) \\
\quad \mid \mathsf{Hash}(T) \Rightarrow \mathsf{true} \\
\quad \mid \mathsf{SymKey}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{pub}) \wedge \mathsf{fkind}(E, T, \mathsf{tnt}) \\
\quad \mid \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\, \exists \widetilde{x}.\, C) \Rightarrow \bigwedge_{T \in \widetilde{T}} \mathsf{fkind}(E, T, \mathsf{pub}) \\
\quad \mathsf{end}) \\
\mathsf{with}\ \mathsf{fkind}(E, V, \mathsf{tnt}) \triangleq \\
\quad (\mathsf{match}\ V\ \mathsf{with} \\
\quad \mid \top \Rightarrow \mathsf{true} \\
\quad \mid T_1 \wedge T_2 \Rightarrow \mathsf{fkind}(E, T_1, \mathsf{tnt}) \wedge \mathsf{fkind}(E, T_2, \mathsf{tnt}) \\
\quad \mid T_1 \vee T_2 \Rightarrow \mathsf{fkind}(E, T_1, \mathsf{tnt}) \vee \mathsf{fkind}(E, T_2, \mathsf{tnt}) \\
\quad \mid \{x : T \mid C\} \Rightarrow \\
\qquad \mathsf{let}\ y = \mathsf{pick\_not\_in}(dom(E))\ \mathsf{in} \\
\qquad \mathsf{fkind}(E, T, \mathsf{tnt}) \wedge \mathsf{fentails}((E, y : T), C\{y/x\}) \\
\quad \mid \mathsf{Un} \Rightarrow \mathsf{true} \\
\quad \mid \mathsf{Ch}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{pub}) \wedge \mathsf{fkind}(E, T, \mathsf{tnt}) \\
\quad \mid \mathsf{Pair}(x{:}T, U) \Rightarrow \\
\qquad \mathsf{let}\ y = \mathsf{pick\_not\_in}(dom(E))\ \mathsf{in} \\
\qquad \mathsf{fkind}(E, T, \mathsf{tnt}) \wedge \mathsf{fkind}((E, y : T), U\{y/x\}, \mathsf{tnt}) \\
\quad \mid \mathsf{Signed}(T) \Rightarrow \mathsf{true} \\
\quad \mid \mathsf{SigKey}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{tnt}) \\
\quad \mid \mathsf{VerKey}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{tnt}) \\
\quad \mid \mathsf{PubEnc}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{tnt}) \\
\quad \mid \mathsf{EncKey}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{pub}) \\
\quad \mid \mathsf{DecKey}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{pub}) \\
\quad \mid \mathsf{Hash}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{tnt}) \\
\quad \mid \mathsf{SymKey}(T) \Rightarrow \mathsf{fkind}(E, T, \mathsf{pub}) \wedge \mathsf{fkind}(E, T, \mathsf{tnt}) \\
\quad \mid \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\, \exists \widetilde{x}.\, C) \Rightarrow \bigwedge_{T \in \widetilde{T}} \mathsf{fkind}(E, T, \mathsf{tnt}) \wedge \mathsf{fentails}(E, \forall \widetilde{x}.\ \forall \widetilde{y}.\ C) \\
\quad \mathsf{end})
\end{array}
$$

We illustrate the logical characterization of kinding, by deriving the formula stating that the channel type $\mathsf{Ch}(\{x : \mathsf{Un} \mid C\})$ is tainted in environment $E$. We calculate as follows by applying the definition above:

$$
\begin{array}{l}
\mathsf{fkind}(E, \mathsf{Ch}(\{x : \mathsf{Un} \mid C\}), \mathsf{tnt}) \\
= \mathsf{fkind}(E, \{x : \mathsf{Un} \mid C\}, \mathsf{pub}) \wedge \mathsf{fkind}(E, \{x : \mathsf{Un} \mid C\}, \mathsf{tnt})
\end{array}
$$

$\mathsf{fkind}(E, \{x : \mathsf{Un} \mid C\}, \mathsf{pub})$
$= \mathsf{fentails}((E, z : \{x : \mathsf{Un} \mid C\}), \mathsf{false}) \vee \mathsf{fkind}(E, \mathsf{Un}, \mathsf{pub})$
$= (\forall dom(E).\forall z.\ \bigwedge forms(E, z : \{x : \mathsf{Un} \mid C\}) \Rightarrow \mathsf{false}) \vee \mathsf{true}$
$= (\forall dom(E).\forall z.\ \bigwedge forms(E) \wedge C\{z/x\} \Rightarrow \mathsf{false}) \vee \mathsf{true}$

$\mathsf{fkind}(E, \{x : \mathsf{Un} \mid C\}, \mathsf{tnt})$
$= \mathsf{fkind}(E, \mathsf{Un}, \mathsf{tnt}) \wedge \forall dom(E).\forall z.\ \bigwedge forms(E, z : \mathsf{Un}) \Rightarrow C\{z/x\}$
$= \mathsf{true} \wedge \forall dom(E).\forall z.\ \bigwedge forms(E) \Rightarrow C\{z/x\}$

The result of $\mathsf{fkind}(E, \{x : \mathsf{Un} \mid C\}, \mathsf{pub})$ is logically equivalent to $\mathsf{true}$ (by (Or Intro), (Red), and the encoding of $\mathsf{true}$), so the result of $\mathsf{fkind}(E, \mathsf{Ch}(\{x : \mathsf{Un} \mid C\}), \mathsf{tnt})$ is logically equivalent to the formula $\forall dom(E).\ \bigwedge forms(E) \Rightarrow \forall z.\, C\{z/x\}$.

### 2.4.6. Type Private and Non-disjointness of Types

Type Private, the type of messages not known to the attacker, is not primitive in our type system. It could easily be added, but it can also be easily encoded, so we chose to encode it. One thing to note, though, is that in our type system we chose not to have any unconditionally secure type. So in an inconsistent environment, which entails false, all types are both public and tainted, so all types are equivalent by subtyping, even Private and Un. We therefore encode a more general type $\mathsf{PrivateUnless}(C)$: the terms in this type are not known to the attacker, unless the formula $C$ is entailed by the environment. We then obtain type Private as $\mathsf{PrivateUnless}(\mathsf{false})$.

**Encoding of OK-types and Private types**

| | |
|---|---|
| $\{C\} \triangleq \{x : \mathsf{Un} \mid C\}$ | for some $x \notin fv(C)$ |
| $\mathsf{PrivateUnless}(C) \triangleq \mathsf{Ch}(\{C\})$ | |
| $\mathsf{Private} \triangleq \mathsf{PrivateUnless}(\mathsf{false})$ | |

We encode type $\mathsf{PrivateUnless}(C)$ as the channel type $\mathsf{Ch}(\{C\})$. Since channel types are generative this allows us to create terms of type $\mathsf{PrivateUnless}(C)$ using restrictions (new). The OK-type [FGM07a] $\{C\}$ is always public, but is tainted only in an environment in which $C$ holds. Since the only kinding rules that apply for channel types are (Kind Chan) and (Kind Empty Pub), we can easily derive that the channel type $\mathsf{Ch}(\{C\})$ is public or tainted only in an environment in which $C$ is entailed. In particular, type Private is only public or tainted in an inconsistent environment.

Knowing that type Private is neither public (i.e., not a subtype of Un) nor tainted (i.e., not a supertype of Un) unless the environment is inconsistent is, however, not enough for the kind of reasoning we want to be able to do for determining with certainty that a zero-knowledge proof cannot come from the attacker. If we can infer that one of the arguments of a zero-knowledge proof has both type Private and type Un we want to be able to deduce that the environment is inconsistent. Intuitively, in a consistent environment type Private and type Un are disjoint, i.e., they do not share any closed

term.[11] More generally, type $\mathsf{PrivateUnless}(C)$ and type $\mathsf{Un}$ do not share any closed terms, unless $C$ is entailed by the environment.

This disjointness property is quite complicated to show, but at a very high-level the reasoning can be summarized as follows: In a consistent environment any closed term of type $\mathsf{Private}$ has to be a name $a$ that is bound in the typing environment to a channel type $\mathsf{Ch}(T)$, for some type $T$ that is equivalent by subtyping[12] to the OK-type $\{\mathsf{false}\}$. Since the name $a$ can also be given type $\mathsf{Un}$ then it must be the case that type $\mathsf{Ch}(T)$ is public, which means that type $T$ is public and tainted, and therefore $\{\mathsf{false}\}$ is public and tainted. The type $\{\mathsf{false}\}$ is however only tainted in an inconsistent environment, which contradicts our original assumption. While we have manually done such disjointness proofs in Coq, the proofs involve non-trivial inductive arguments and use many of the properties we have proved about the type system, and we cannot expect an automated type-checker to do such complicated meta-reasoning about our typing judgments.

In order to reason about type (non-)disjointness in our type system in a purely syntactic manner, we introduce a new inductively-defined typing judgment $E \vdash\ T \between U \rightsquigarrow C$ which guarantees that if the types $T$ and $U$ intersect in $E$ then the formula $C$ is entailed by $E$. Defining this judgment is challenging in our setting because kinding makes many types overlap. For instance, in our type system the types $\mathsf{Ch}(\mathsf{Un})$ and $\mathsf{Pair}(\mathsf{Un},\mathsf{Un})$ are equivalent, so it is not enough to look only at the top-level type constructor to decide if two types can overlap. Moreover, in an inconsistent environment all types overlap, since all types are equivalent by subtyping.

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

(ND Gen)

$$\frac{T \text{ and } U \text{ are generative and have different top-level type constructors}}{E \vdash\ T \between U \rightsquigarrow (\mathsf{fkind}(E,T,\mathsf{pub}) \wedge \mathsf{fkind}(E,U,\mathsf{tnt})) \vee (\mathsf{fkind}(E,T,\mathsf{tnt}) \wedge \mathsf{fkind}(E,U,\mathsf{pub}))}$$

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

(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)
$$\frac{E \vdash T_1\ ok \quad E \vdash T_2\ ok \quad x \notin fv(T_1,T_2)}{E \vdash\ T_1 \between T_2 \rightsquigarrow \exists x.\ \bigwedge forms_x(T_1 \wedge T_2)}$$

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

(ND Pair)
$$\frac{E \vdash\ T_1 \between U_1 \rightsquigarrow C_1 \quad E \vdash\ T_2 \between U_2 \rightsquigarrow C_2}{E \vdash\ \mathsf{Pair}(x{:}T_1,T_2) \between \mathsf{Pair}(y{:}U_1,U_2) \rightsquigarrow C_1 \wedge C_2}$$

[11] The fact that a type $T$ is neither public (i.e., not a subtype of $\mathsf{Un}$) nor tainted (i.e., not a supertype of $\mathsf{Un}$) is a necessary but not a sufficient condition for $T$ being disjoint from $\mathsf{Un}$. For example, type $T_0 = \mathsf{Pair}(\top, \{x : \mathsf{Un} \mid P(x)\})$ is neither public nor tainted in the environment $\{P(\mathsf{unit})\}$, still in this environment the term $(\mathsf{unit},\mathsf{unit})$ has both type $T_0$ and type $\mathsf{Un}$.

[12] Channel types are invariant, but our subtyping relation is not anti-symmetric.

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

Fortunately, we can use the logical characterization of kinding to capture the effect of kinding on type disjointness. The most important rule of the non-disjointness judgment (ND Gen) states that if two generative types with different top-level type constructors overlap then one of them is public and the other is tainted. This rule is very general and can be instantiated with different generative types. For instance it allows us to derive syntactically that Private and Un only overlap in an inconsistent environment (see derived rule (ND Private Un) below); that if $\mathsf{Ch}(T)$ or $\mathsf{SymKey}(T)$ overlap with Un then $T$ is both public and tainted (derived rules (ND Channel Un) and (ND SymKey Un)); and that if $\mathsf{SigKey}(T)$ overlaps with Un then $T$ is tainted (derived rule (ND SigKey Un)).

**Non-disjointness rules derived from (ND Gen) and (ND Entails)**

(ND Private Un)
$$\frac{E \vdash C\ ok}{E \vdash\ \mathsf{Private} \between \mathsf{Un} \rightsquigarrow \mathsf{false}}$$

(ND PrivateUnless Un)
$$\frac{E \vdash C\ ok}{E \vdash\ \mathsf{PrivateUnless}(C) \between \mathsf{Un} \rightsquigarrow C}$$

(ND Channel Un)
$$\frac{E \vdash T\ ok}{E \vdash\ \mathsf{Ch}(T) \between \mathsf{Un} \rightsquigarrow \mathsf{fkind}(E, T, \mathsf{tnt}) \wedge \mathsf{fkind}(E, T, \mathsf{pub})}$$

(ND SymKey Un)
$$\frac{E \vdash T\ ok}{E \vdash\ \mathsf{SymKey}(T) \between \mathsf{Un} \rightsquigarrow \mathsf{fkind}(E, T, \mathsf{tnt}) \wedge \mathsf{fkind}(E, T, \mathsf{pub})}$$

(ND DecKey Un)
$$\frac{E \vdash T\ ok}{E \vdash\ \mathsf{DecKey}(T) \between \mathsf{Un} \rightsquigarrow \mathsf{fkind}(E, T, \mathsf{pub})}$$

(ND SigKey Un)
$$\frac{E \vdash T\ ok}{E \vdash\ \mathsf{SigKey}(T) \between \mathsf{Un} \rightsquigarrow \mathsf{fkind}(E, T, \mathsf{tnt})}$$

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 states that bottom 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). The remaining rules ((ND Or), (ND And), and (ND Pair)) lift the non-disjointness judgment to union and intersection types as well as non-dependent pair types.

**More derived non-disjointness rules**

$$\frac{E \vdash T_1\ ok \quad E \vdash T_2\ ok}{E \vdash\ T_1 \between T_2 \rightsquigarrow \bigwedge forms(E)}\ \text{(ND Forms Env)} \qquad \frac{E \vdash T_1\ ok \quad E \vdash T_2\ ok \quad E, x : T_1 \vdash \mathsf{false}}{E \vdash\ T_1 \between T_2 \rightsquigarrow \mathsf{false}}\ \text{(ND Forms Empty)}$$

$$\frac{E \vdash\ T_1 \between T_2 \rightsquigarrow C \quad E, x : T_1 \vdash C_1\ ok}{E \vdash\ \{x : T_1 \mid C_1\} \between T_2 \rightsquigarrow C}\ \text{(ND Refine)} \qquad \frac{E \vdash\ T_1 \between T_2 \rightsquigarrow C \quad E \vdash \{x : T_1 \mid C_1\}\ ok \quad E \vdash \{x : T_2 \mid C_2\}\ ok}{E \vdash\ \{x : T_1 \mid C_1\} \between \{x : T_2 \mid C_2\} \rightsquigarrow C \wedge \exists x.\, C_1 \wedge C_2}\ \text{(ND Refine Exists)}$$

## 2.4.7. Typing Terms and Destructors

The rules of the term typing judgment $E \vdash M : T$ are listed below. The rules (Term Var) and (Term Name) look up the type of variables and names in the typing environment. The subsumption rule (Term Subsum) allows a term having type $T$ to be given any supertype of $T$. Rule (Term And) gives a term an intersection type $T \wedge U$ provided that it can be typed to both type $T$ and type $U$. Rule (Term Refine) gives a term $M$ a refinement type $\{x : T \mid C\}$ provided $M$ has type $T$ and the formula $C\{M/x\}$ is entailed in the typing environment[13]. Rule (Term Pair) gives pairs dependent types $\mathsf{Pair}(x{:}T, U)$, where the type of the second component in the pair $U$ refers to the term in the first component via the variable $x$.

**Typing terms:** $E \vdash M : T$

$$\frac{\vdash E\ ok}{E \vdash \mathsf{unit} : \mathsf{Un}}\ \text{(Term Unit)} \qquad \frac{\vdash E\ ok \quad (x : T) \in E}{E \vdash x : T}\ \text{(Term Var)} \qquad \frac{\vdash E\ ok \quad (a : T) \in E}{E \vdash a : T}\ \text{(Term Name)} \qquad \frac{E \vdash M : T \quad E \vdash T <: T'}{E \vdash M : T'}\ \text{(Term Subsum)}$$

$$\frac{E \vdash M : T \quad E \vdash M : U}{E \vdash M : T \wedge U}\ \text{(Term And)} \qquad \frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x : T \mid C\}}\ \text{(Term Refine)} \qquad \frac{E \vdash M_1 : T_1 \quad E \vdash M_2 : T_2\{M_1/x\}}{E \vdash (M_1, M_2) : \mathsf{Pair}(x{:}T_1, T_2)}\ \text{(Term Pair)}$$

[13] Rule (Term Refine) cannot be encoded with (Term Subsum) and (Sub Refine Right), since this would require proving $C$ for all values of $x$, while (Term Refine) only requires proving $C$ for the term $M$.

$$\text{(Term Enc Key)}\ \frac{E \vdash M : \mathsf{DecKey}(T)}{E \vdash \mathsf{ek}(M) : \mathsf{EncKey}(T)} \qquad \text{(Term Pub Enc)}\ \frac{E \vdash M : T \quad E \vdash K : \mathsf{EncKey}(T)}{E \vdash \mathsf{enc}(M, K) : \mathsf{PubEnc}(T)} \qquad \text{(Term Ver Key)}\ \frac{E \vdash M : \mathsf{SigKey}(T)}{E \vdash \mathsf{vk}(M) : \mathsf{VerKey}(T)}$$

$$\text{(Term Sign)}\ \frac{E \vdash M : T \quad E \vdash K : \mathsf{SigKey}(T)}{E \vdash \mathsf{sign}(M, K) : \mathsf{Signed}(T)} \qquad \text{(Term Hash)}\ \frac{E \vdash M : T}{E \vdash \mathsf{hash}(M) : \mathsf{Hash}(T)}$$

$$\text{(Term Sym Enc)}\ \frac{E \vdash M : T \quad E \vdash K : \mathsf{SymKey}(T)}{E \vdash \mathsf{senc}(M, K) : \mathsf{Un}} \qquad \text{(Term ZK Un)}\ \frac{E \vdash \widetilde{N} : \widetilde{\mathsf{Un}} \quad E \vdash \widetilde{M} : \widetilde{\mathsf{Un}}}{E \vdash \mathsf{zk}_S(\widetilde{N};\, \widetilde{M}) : \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{\mathsf{Un}};\, \exists \widetilde{x}.\, \mathsf{true})}$$

$$\text{(Term ZK)}\ \frac{T_S = \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\, \exists \widetilde{x}.\, C) \quad E \vdash \widetilde{N} : \widetilde{\top} \quad E \vdash \widetilde{M} : \widetilde{T} \quad E \vdash C\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}}{E \vdash \mathsf{zk}_S(\widetilde{N};\, \widetilde{M}) : T_S}$$

The typing rules for the constructors representing the basic cryptographic primitives are all standard [FGM07a]. More interestingly, for typing zero-knowledge terms we have two different rules: rule (Term ZK) is used for typing honest protocol participants, while rule (Term ZK Un) is used for typing the opponent to ensure that we are not putting undue restrictions on it.

Typing most of the basic cryptographic primitives relies on the type of some key, which the user has to annotate explicitly. But zero-knowledge proofs do not depend on keys in general. This is a problem since the successful verification of a zero-knowledge proof should propagate logical formulas in the typing environment of the verifier, and it is not clear what formulas to consider. For instance, when typing a verification process $\mathsf{if}\ \mathsf{ver}_S(z) \Downarrow \langle y \rangle\ \mathsf{then}\ P$ for some statement $S = \mathsf{witness}\ \widetilde{x}\ \mathsf{public}\ \widetilde{y}\ \mathsf{in}\ B$, we can safely assume that the basic formula $\exists \widetilde{x}.B$ holds for the continuation process $P$, since this is guaranteed by the operational semantics of the ver destructor (see §2.3.2). However, such a basic formula does not suffice for typing many of the examples we have tried, since it does not mention any user-defined logical predicate.

In order to solve this problem, we assume that each zero-knowledge statement $S = \mathsf{witness}\ \widetilde{x}\ \mathsf{public}\ \widetilde{y}\ \mathsf{in}\ B$ is annotated by the user with a zero-knowledge proof type $T_S = \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\, \exists \widetilde{x}.\, C)$. When typing a zero-knowledge proof created by a honest protocol participant $\mathsf{zk}_S(\widetilde{N};\, \widetilde{M})$ using rule (Term ZK) we ensure that the public arguments of the proof $\widetilde{M}$ have the types specified by the user $\widetilde{T}$, and that the logical formula $C$ specified by the user and instantiated with the actual arguments of the proof ($\widetilde{x}$ with $\widetilde{N}$ and $\widetilde{y}$ with $\widetilde{M}$) is entailed by the typing environment. Only if these two conditions are satisfied we give the zero-knowledge term type $T_S$. Note that if the formula $C$ contains the basic formula $B$ (i.e., if $C$ is of the form $B \wedge C_P$, for some promise $C_P$) we can ensure that only valid zero-knowledge proofs are created by the honest participants. However, since $C$ can be chosen arbitrarily by the user this is not necessarily true in general. Moreover, rule (Term ZK Un) allows the opponent to generate possibly invalid

zero-knowledge proofs from terms it knows (i.e., any term of type $\mathsf{Un}$) for any statement. In this case, however, even if the statement was annotated by the user with type $T_S$, this type is ignored, and the proof is given type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{\mathsf{Un}}; \exists \widetilde{x}.\, \mathsf{true})$, which is equivalent by subtyping to $\mathsf{Un}$.

So the fact that a term can be given a zero-knowledge proof type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.\, C)$ for some statement $S = \mathsf{witness}\ \widetilde{x}\ \mathsf{public}\ \widetilde{y}\ \mathsf{in}\ B$, only guarantees that the formula $C$ in the type holds for the arguments of the proof, while this need not be the case for the basic formula $B$ in the statement. To obtain the basic formula $B$ the zero-knowledge proof has to be successfully verified. Most of the time zero-knowledge proofs are communicated over untrusted channels, so before the receiver verifies the proof we can only give the received proofs type $\mathsf{Un}$, which is equivalent by subtyping to $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{\mathsf{Un}}; \exists \widetilde{x}.\, \mathsf{true})$. This type conveys no useful logical formula.

Most of the typing rules for destructors are simple[14]. The typing rule for equality is very strong: the return type makes use of an intersection type and of the non-disjointness judgment. The equality destructor $\mathsf{eq}(M_1, M_2)$ only succeeds when $M_1 = M_2$, and returns the common value of $M_1$ and $M_2$ as a result. If statically $M_1$ had type $T_1$ and $M_2$ had type $T_2$ then their common value will have both type $T_1$ and type $T_2$, so also type $T_1 \wedge T_2$. Moreover, since this closed term is an inhabitant of both $T_1$ and $T_2$ the two types overlap, so we use the non-disjointness judgment to derive an additional formula that we use to refine the type of the result. This increases the precision of the type system when type-checking conditionals encoded using the equality destructor.

**Typing destructors:** $E \vdash D : T$

(Dtor Id)
$$\frac{E \vdash M : T}{E \vdash \mathsf{id}(M) : T}$$

(Dtor Eq)
$$\frac{E \vdash M_1 : T_1 \quad E \vdash M_2 : T_2 \quad E \vdash T_1 \between T_2 \rightsquigarrow C \quad x \notin dom(E)}{E \vdash \mathsf{eq}(M_1, M_2) : \{x : T_1 \wedge T_2 \mid C\}}$$

(Dtor Dec)
$$\frac{E \vdash M : \mathsf{PubEnc}(T) \quad E \vdash K : \mathsf{DecKey}(T)}{E \vdash \mathsf{dec}(M, K) : T}$$

(Dtor Check)
$$\frac{E \vdash M : \mathsf{Signed}(T) \quad E \vdash K : \mathsf{VerKey}(T)}{E \vdash \mathsf{check}(M, K) : T}$$

(Dtor Sym Dec)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash K : \mathsf{SymKey}(T)}{E \vdash \mathsf{sdec}(M, K) : T}$$

(Dtor Public)
$$\frac{E \vdash M : \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.\, C)}{E \vdash \mathsf{public}_S(M) : \langle \widetilde{y} : \widetilde{T} \rangle \{\exists \widetilde{x}.\, C\}}$$

**Notation:**
Let $\langle \widetilde{x} : \widetilde{T} \rangle \{C\}$ denote the refined tuple type $\mathsf{Pair}(x_1{:}T_1, \ldots \mathsf{Pair}(x_n{:}T_n, \{C\}))$ where we require that the variables $\widetilde{x}$ are not bound in $\widetilde{T}$ (they can be bound in $C$ though).

Let $\langle \widetilde{T} \rangle$ denote the tuple type $\langle \widetilde{x} : \widetilde{T} \rangle \{\mathsf{true}\}$ for some fresh $\widetilde{x}$.

[14] Proving them consistent is still very complicated because of subtyping and kinding; see §2.5.5.

The public destructor returns a tuple containing the public arguments of a zero-knowledge proof, without verifying whether the proof is valid. Rule (Dtor Public) requires that the input to public has a zero-knowledge proof type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}. C)$, and if this is indeed the case it gives the result the dependent tuple type $\langle \widetilde{y} : \widetilde{T} \rangle \{\exists \widetilde{x}. C\}$. This apparently strong return type might be surprising, since after all the public destructor does not verify whether the proof is valid. However, it is the zero-knowledge type of the input that justifies this return type. If the proof comes from a public channel, as is usually the case, then it can only be given type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{\mathsf{Un}}; \exists \widetilde{x}. \mathsf{true})$, which makes the result of public have type $\langle \widetilde{\mathsf{Un}} \rangle$, so not so strong after all. Only when the proof is somehow authenticated (e.g., it comes from an authentic channel) can the receiver give the result a strong type, but this is justified by the fact that $C$ was already checked on the side of the sender. More formally, our type system preserves the invariant that whenever a zero-knowledge proof $\mathsf{zk}_Z(\widetilde{N}; \widetilde{M})$ can be given type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}. C)$, the formula $C\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}$ is entailed by the typing environment. Since the $\mathsf{public}_S$ destructor returns the tuple $\langle \widetilde{M} \rangle$ when given such a proof, we can easily justify giving this tuple type $\langle \widetilde{y} : \widetilde{T} \rangle \{\exists \widetilde{x}. C\}$.

### 2.4.8. Typing Processes

The rules of the process typing judgment $E \vdash P$ are listed below. Rule (Proc Out) enforces that only terms of type $T$ can be sent over a channel of type $\mathsf{Ch}(T)$, while rules (Proc In) and (Proc In Repl) guarantee that the terms received from such a channel have type $T$. Rule (Proc New) enforces that the names bound by restriction processes are given generative types. Rule (Proc Assume) is very permissive, any well-formed formula can be assumed. On the other hand, rule (Proc Assert) checks that the asserted formula is entailed by the current typing environment and therefore also entailed at execution time from the active assumes. It is up to the user of the type system to make sure that the assumes and asserts really reflect her intentions, and special attention should be given to them when auditing the protocol model. Assumes in particular are trusted operations, and a bad assume could make the environment inconsistent and thus allow any protocol to be robustly safe. Rule (Proc Par) for parallel compositions extracts the active assumes from one process and adds them to the typing environment when typing the other process, ensuring active assumes have global scope. This is necessary for proving subject reduction, because in the operational semantics active assumes are not added to some explicit formula log, but instead persist in parallel with the rest of the system.

**Typing processes:** $E \vdash P$

(Proc Out)
$$\frac{E \vdash N : \mathsf{Ch}(T) \quad E \vdash M : T \quad E \vdash P}{E \vdash \mathsf{out}(N, M).\, P}$$

(Proc In)
$$\frac{E \vdash N : \mathsf{Ch}(T) \quad E, x : T \vdash P}{E \vdash \mathsf{in}(N, x).\, P}$$

(Proc In Repl)
$$\frac{E \vdash \mathsf{in}(N, x).\, P}{E \vdash !\mathsf{in}(N, x).\, P}$$

$$\frac{E \vdash C\ ok}{E \vdash \mathsf{assume}\ C}\ (\text{Proc Assume}) \qquad \frac{E \vdash C}{E \vdash \mathsf{assert}\ C}\ (\text{Proc Assert}) \qquad \frac{E, \{\overline{Q}\} \vdash P \quad E, \{\overline{P}\} \vdash Q}{E \vdash P \mid Q}\ (\text{Proc Par}) \qquad \frac{\vdash E\ ok}{E \vdash \mathbf{0}}\ (\text{Proc Zero})$$

$$\frac{E \vdash M : T_1 \vee T_2 \quad E, x : T_1, \{x = M\} \vdash P \quad E, x : T_2, \{x = M\} \vdash P}{E \vdash \mathsf{case}\ x = M\ \mathsf{in}\ P}\ (\text{Proc Case})$$

$$\frac{E \vdash M : \mathsf{Pair}(x{:}T, U) \quad E, x : T, y : U, \{(x, y) = M\} \vdash P}{E \vdash \mathsf{let}\ (x, y) = M\ \mathsf{in}\ P}\ (\text{Proc Split}) \qquad \frac{T\ \text{generative} \quad E, a : T \vdash P}{E \vdash (\mathsf{new}\ a : T)\, P}\ (\text{Proc New})$$

$$\frac{E \vdash D : T \quad E, x : T, \{D \rightsquigarrow x\} \vdash P_1 \quad E, \{\neg \exists y.\, D \rightsquigarrow y\} \vdash P_2}{E \vdash \mathsf{if}\ D \Downarrow x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2}\ (\text{Proc Dtor})$$

$$\frac{\begin{array}{c} T_S = \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T};\ \exists \widetilde{x}.\, C) \qquad S = \mathsf{witness}\ x_1, \ldots, x_n\ \mathsf{public}\ y_1, \ldots, y_m\ \mathsf{in}\ B \\ E \vdash N : \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{U};\ \exists \widetilde{x}.\, C_0) \qquad \forall i \in [1, l].\ E \vdash L_i : T_i \\ E_0 = [x_k : \mathsf{Un} \mid k \in [1, n]], [y_i : T_i \wedge U_i \wedge \mathsf{Un} \mid i \in [1, l]], [y_j : U_j \wedge \mathsf{Un} \mid j \in [l+1, m]] \\ E \vdash [|B|]_{E_0, C_0} \rightsquigarrow E_1, C_1 \qquad E, E_1, \{C_1\} \vdash C \qquad \forall j \in [l+1, m].\ E, E_1, \{C_1\} \vdash y_j : T_j \\ E, z : \langle y_{l+1} : T_{l+1}, \ldots, y_m : T_m \rangle \{\exists \widetilde{x}.\ C\{L_i / y_i\}_{i \in [1,l]}\} \vdash P_1 \qquad E \vdash P_2 \end{array}}{E \vdash \mathsf{ver}_S(N, L_1, \ldots, L_l) \Downarrow z\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2}\ (\text{Proc Ver})$$

$$\frac{E \vdash N : \mathsf{Un} \quad \forall i \in [1, l].\ E \vdash L_i : \mathsf{Un} \quad E, z : \mathsf{Un} \vdash P_1 \quad E \vdash P_2}{E \vdash \mathsf{ver}_S(N, \widetilde{L}) \Downarrow z\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2}\ (\text{Proc Ver Un})$$

The case process explicitly marks the places where union types are eliminated, which makes type-checking simpler and more tractable. Rule (Proc Case) requires that the term $M$ passed to case has a union type $T_1 \vee T_2$, binds $M$ to a variable $x$, and types the continuation process twice, once under the assumption that $x$ has type $T_1$ and a second time assuming that $x$ has type $T_2$. Additionally we record the fact that $x$ and $M$ are equal by adding this formula to the environment used to type the continuation process. Formulas are also added in rules (Proc Split) and (Proc Dtor), and this increases the precision of the type system. In rule (Proc Dtor) we add a formula stating that the destructor reduced successfully and returned $x$ when typing the then branch, and a negative formula stating that destructor reduction failed when typing the else branch. This is particularly interesting for the equality destructor, which already has a very strong typing rule (Dtor Eq), which uses the non-disjointness judgment.

As for creating zero-knowledge proofs, there are two different rules for typing zero-knowledge verification processes: rule (Proc Ver) for typing honest verifiers and rule (Proc Ver Un) for typing the verifications done by the attacker. Rule (Proc Ver Un) is simple, the zero-knowledge proof to be verified as well as the matched arguments are all expected to have type Un, and the returned tuple of arguments is given type Un in the

continuation process. Rule (Proc Ver) is much more complicated, and is explained in §2.4.9 below.

### 2.4.9. Type-checking Zero-knowledge Verification

This section is dedicated to type-checking honest verifiers of zero-knowledge proofs, which is very challenging and constitutes the main technical contribution of this whole chapter. If the proof received by a honest verifier was created by a honest prover, then we can justify giving the returned arguments of the proof the strong type annotated by the user. This is sound because honest provers are type-checked using rule (Term ZK), which ensures that the arguments used to create the proof have the strong type. However, opponents can also create valid zero-knowledge proofs and send them over untrusted channels. Opponents (and compromised participants) are typed using the much more permissive (Term ZK Un) rule, which does not place any restriction on what arguments the opponent can use to create the proof. Rule (Term ZK Un) only requires that the arguments can be given type $\mathsf{Un}$, but this is not a restriction on the opponent, because our type system enforces anyway that opponents only learn terms of type $\mathsf{Un}$[15].

When verifying a zero-knowledge proof received from the untrusted network (which is under the control of the opponent) a honest participant does not know a priori whether the creator of the proof is honest or not. So the type system has to check whether the strong type annotated by the user is still justified even if the proof was created by the opponent. Our type system starts from the zero-knowledge statement being verified and from the types of those public arguments of the proof that the verifier has obtained from a reliable source, and uses intersection, union, and refinement types to infer very precise type information about the other arguments of the zero-knowledge proof. For certain protocols the types inferred this way are already strong enough to justify the user-annotated zero-knowledge proof type [BGHM09, BLMP10]. In other protocols [BCC04, LHH+07] an additional insight is needed: this whole inference process also succeeds if the type system can somehow deduce that the proof was necessarily constructed by an honest prover. This happens when the type inferred for one of the secret witnesses of the proof is disjoint from the type of messages possibly known to the attacker.

Since we support security despite compromise this is even more complicated: all the reasoning above has to be conditioned by certain protocol participants being indeed honest (not compromised). We use union types together with refinement types that contain formulas obtained using our logical characterization of kinding to express type information that is conditioned by a participant not being compromised. Such conditional types are inferred automatically when processing the zero-knowledge statement. The remainder of this section explains the whole inference process in more detail.

[15] As opponent typability (Lemma 2.33 in §2.5.8) shows, this is only a restriction on the protocol, not on the opponent.

We start by explaining the (Proc Ver) typing rule. Suppose that we want to type-check the process $\mathsf{ver}_S(N, L_1, \ldots, L_l) \Downarrow z$ $\mathsf{then}$ $P_1$ $\mathsf{else}$ $P_2$, which checks whether $N$ is a valid proof for the statement $S = \mathsf{witness}\ x_1, \ldots, x_n\ \mathsf{public}\ y_1, \ldots, y_m\ \mathsf{in}\ B$. We require $N$ to have some zero-knowledge type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{U}; \exists \widetilde{x}. C_0)$; this type is usually much weaker than the type $T_S = \mathsf{ZKProof}_S(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}. C)$ annotated by the user. In the very common case when $N$ is received from a public channel the only type the verifier can give to $N$ a priori is $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{\mathsf{Un}}; \exists \widetilde{x}. \mathsf{true})$, which is equivalent to $\mathsf{Un}$. We also require that the matched arguments $L_1, \ldots, L_l$ have the strong types $T_1, \ldots, T_l$ defined by $T_S$. Most often it is the strong types of these matched arguments that allow our type system to infer strong types for the other arguments, even when the proof was created by the opponent. If $N$ was a zero-knowledge proof $\mathsf{zk}_S(\widetilde{N}; \widetilde{M})$ created by a honest prover, then the strong types of the public arguments $\widetilde{T}$ and the formula $C\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}$ were already checked on the prover's side.

On the other hand, if $N$ was created by the opponent, justifying the strong type $T_S$ in case the zero-knowledge verification succeeds is much more complicated. In this case the type system starts from the basic formula $B$ of the zero-knowledge statement $S$, whose validity is guaranteed by the operational semantics of zero-knowledge verification, and from an environment $E_0$ containing the type information the type system already has about the arguments of the proof, and tries to infer more information about the arguments. The initial environment $E_0$ is constructed as follows: First, because we are dealing with the case in which the proof was created by the opponent, and opponents only learn terms of type $\mathsf{Un}$, we can assume that all arguments of the proof have type $\mathsf{Un}$. Second, since the received proof $N$ has the (usually weak) zero-knowledge type $\mathsf{ZKProof}_S(\widetilde{y} : \widetilde{U}; \exists \widetilde{x}. C_0)$ we can additionally assume that the public arguments of the proof $\widetilde{y}$ also have types $\widetilde{U}$, and that the formula $C_0$ holds. Finally, since verification succeeds only if the arguments of the proof $y_1, \ldots, y_l$ match the terms provided by the verifier $L_1, \ldots, L_l$, we can assume that the matched arguments $y_1, \ldots, y_l$ also have the strong types $T_1, \ldots, T_l$ of $L_1, \ldots, L_l$. We use intersection types to capture the fact that the public arguments have simultaneously more than one type:

$$E_0 = [x_k : \mathsf{Un} \mid k \in [1, n]], [y_i : T_i \wedge U_i \wedge \mathsf{Un} \mid i \in [1, l]], [y_j : U_j \wedge \mathsf{Un} \mid j \in [l + 1, m]].$$

For instance in the simplified DAA example from §2.2, the type system initially has the following type information:

$$E_0^{sdaa} = x_f : \mathsf{Un}, x_{cert} : \mathsf{Un}, y_{vki} : (\mathsf{VerKey}(T_{ki}) \wedge \mathsf{Un} \wedge \mathsf{Un}), y_m : (\mathsf{Un} \wedge \mathsf{Un}).$$

Rule (Proc Ver) uses the basic formula $B$ of the zero-knowledge statement and our novel statement-based inference judgment $E \vdash [|B|]_{E_0,C_0} \rightsquigarrow E_1, C_1$ to infer a stronger set of types $E_1$ for the arguments of the zero-knowledge proof, as well as a stronger formula $C_1$. The new information in $E_1$ and $C_1$ is then used to check whether the strong formula $C$ from the zero-knowledge type annotated by the user is entailed, and whether the returned public arguments $y_{l+1}, \ldots, y_m$ have the strong types $T_{l+1}, \ldots, T_m$ given by the

user, in case the proof were to come from the attacker. Only if all these checks succeed is the verification process accepted by our type system.

The statement-based inference judgment $E \vdash [|B|]_{E_0,C_0} \rightsquigarrow E_1, C_1$ is a 6-ary relation inductively defined by the rules below. Intuitively, the arguments $E$, $B$, $E_0$, and $C_0$ are inputs, while $E_1$ and $C_1$ are outputs: given $B$, the basic formula of the statement we are verifying, and an initial typing environment $E, E_0, \{C_0\}$, the relation infers a stronger typing environment $E, E_1, \{C_1\}$.

**Statement-based inference:** $E \vdash [|B|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, C_{new}$

(Sinfer Stmt)

$$\frac{E \vdash [|B|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, C_{new}}{E \vdash [|B|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, (C_{new} \wedge B)}$$

(Sinfer Ident)

$$\frac{E \vdash E' \; spec \quad E, E' \vdash C' \; ok}{E \vdash [|B|]_{E',C'} \rightsquigarrow E', C'}$$

(Sinfer Red)

$$\frac{E, E_{old}, \{C_{old}\} \vdash D : T \quad E \vdash T \; ok \quad E \vdash (E_{old}, C_{old})[v_N : T] \rightsquigarrow E_{new}, C_{new}}{E \vdash [|D \rightsquigarrow v_N|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, C_{new}}$$

(Sinfer Check)

$$\frac{\begin{array}{c} E, E_{old}, \{C_{old}\} \vdash v_K : \mathsf{VerKey}(T) \quad E \vdash T \; ok \\ E \vdash (E_{old}, C_{old})[v_N : \{\!|\, \mathsf{fkind}(E, T, \mathsf{tnt}) \,|\!\} \vee T] \rightsquigarrow E_{new}, C_{new} \end{array}}{E \vdash [|\mathsf{check}(v_M, v_K) \rightsquigarrow v_N|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, C_{new}}$$

(Sinfer Dec)

$$\frac{\begin{array}{c} E, E_{old}, \{C_{old}\} \vdash v_M : \mathsf{PubEnc}(T) \quad E \vdash T \; ok \\ E \vdash (E_{old}, C_{old})[v_N : \{\!|\, \mathsf{fkind}(E, T, \mathsf{tnt}) \,|\!\} \vee T] \rightsquigarrow E_{new}, C_{new} \end{array}}{E \vdash [|\mathsf{dec}(v_M, v_K) \rightsquigarrow v_N|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, C_{new}}$$

(Sinfer Enc)

$$\frac{\begin{array}{c} E, E_{old}, \{C_{old}\} \vdash v_M : \mathsf{PubEnc}(T) \quad E \vdash T \; ok \\ E \vdash (E_{old}, C_{old})[v_N : \{\!|\, \mathsf{fkind}(E, T, \mathsf{tnt}) \,|\!\} \vee T] \rightsquigarrow E', C' \\ E \vdash (E', C')[v_K : \{\!|\, \mathsf{fkind}(E, T, \mathsf{tnt}) \,|\!\} \vee \mathsf{EncKey}(T)] \rightsquigarrow E_{new}, C_{new} \end{array}}{E \vdash [|v_M = \mathsf{enc}(v_N, v_K)|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, C_{new}}$$

(Sinfer Pair)

$$\frac{\begin{array}{c} E, E_{old}, \{C_{old}\} \vdash v_M : \mathsf{Pair}(z{:}T_1, T_2) \quad E \vdash T_1 \; ok \quad E \vdash T_2 \; ok \\ E \vdash (E_{old}, C_{old})[v_1 : T_1, v_2 : T_2] \rightsquigarrow E_{new}, C_{new} \end{array}}{E \vdash [|v_M = (v_1, v_2)|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, C_{new}}$$

(Sinfer Hash)

$$\frac{\begin{array}{c} E, E_{old}, \{C_{old}\} \vdash v_M : \mathsf{Hash}(T) \quad E \vdash T \; ok \\ E \vdash (E_{old}, C_{old})[v_N : \{\!|\, \mathsf{fkind}(E, T, \mathsf{tnt}) \,|\!\} \vee T] \rightsquigarrow E', C' \end{array}}{E \vdash [|v_M = \mathsf{hash}(v_M)|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, C_{new}}$$

(Sinfer And)

$$\frac{E \vdash [|B_1|]_{E_{old},C_{old}} \rightsquigarrow E_1, C_1 \quad E \vdash [|B_2|]_{E_1,C_1} \rightsquigarrow E_{21}, C_{21} \quad E \vdash [|B_2|]_{E_2,C_2} \rightsquigarrow E_{12}, C_{12} \quad E_{12} \wedge E_{21} \rightsquigarrow E_{new}}{E \vdash [|B_1 \wedge B_2|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, (C_{12} \wedge C_{21})}$$

(Sinfer Or)

$$\frac{E \vdash [|B_1|]_{E_{old},C_{old}} \rightsquigarrow E_1, C_1 \quad E \vdash [|B_2|]_{E_{old},C_{old}} \rightsquigarrow E_2, C_2 \quad E_1 \vee E_2 \rightsquigarrow E_{new}}{E \vdash [|B_1 \vee B_2|]_{E_{old},C_{old}} \rightsquigarrow E_{new}, (C_1 \vee C_2)}$$

**Notation:** Let $\{\!|\, C \,|\!\}$ denote the refinement type $\{x : \top \mid C\}$, where $x \notin fv(C)$.

Rule (Sinfer Stmt) conjoins the basic formula $B$ of the statement to the returned formula, since by the operational semantics (see §2.3.4) zero-knowledge verification succeeds only if $B$ holds. Rule (Sinfer Ident) provides a trivial base case of our inductively defined relation, it simply allows to copy the input environment and formula, provided that they are well-formed. For the environment we require a stronger notion of well-formedness $E \vdash E'$ *spec*, which holds if and only if $E'$ only contains variable bindings, and $E'$ does not have any dependencies to itself, and the compound environment $E, E'$ is well-formed.

**Special well-formed environment extension:** $E \vdash E'$ *spec*

(Wfes Nil)

$$\frac{\vdash E \; ok}{E \vdash \emptyset \; spec}$$

(Wfes Cons)

$$\frac{E \vdash E' \; spec \quad E \vdash T \; ok \quad x \notin dom(E, E')}{E \vdash E', x : T \; spec}$$

Rule (Sinfer Red) deals with arbitrary logical atoms, which in our case are destructor reductions. If the destructor $D$ has type $T_{new}$ in the original environment, and if $D$ reduces to the term denoted by the variable $v_N$, then by destructor consistency (Lemma 2.29 in §2.5) we know that the resulting term will also have type $T_{new}$. We update the environment to record this new information using the strong environment update judgment $E \vdash (E_{old}, C_{old})[v_N : T_{new}] \rightsquigarrow E_{new}, C_{new}$. This updates the type of variable $v_N$ to the intersection between the old type of $v_N$, say $T_{old}$, and the new type $T_{new}$. The argument of the zero-knowledge proof corresponding to variable $v_N$, has thus both type $T_{old}$ and type $T_{new}$, so the types $T_{old}$ and $T_{new}$ cannot be disjoint. We therefore use the non-disjointedness judgment $E \vdash T_{old} \between T_{new} \rightsquigarrow C_{nd}$ to infer an additional formula $C_{nd}$, which we conjoin to the resulting formula, as well as to the type of all variables. If the types $T_{old}$ and $T_{new}$ are disjoint then $C_{nd}$ is $\mathsf{false}$, and the intersection type $V \wedge \{x : \top \mid \mathsf{false}\}$ is empty for any $V$. By rule (Sub Empty) this intersection type is a subtype of any other type, including the strong type annotated by the user for this variable.

**Strong environment update:** $E \vdash (E_{old}, C_{old})[x : T_{new}] \rightsquigarrow E_{new}, C_{new}$

(Strong Update)
$$\frac{E \vdash T_{old} \mathbin{⊙\!\!⃝} T_{new} \rightsquigarrow C_{nd} \qquad \mathsf{supd}(E_{old}, x, T_{old}, T_{new}, C_{nd}) \rightsquigarrow\!\!\!\!\rightarrow E_{new}}{E \vdash (E_{old}, C_{old})[x : T_{new}] \rightsquigarrow\!\!\!\!\rightarrow E_{new}, (C_{old} \wedge C_{nd})}$$

(Supd Empty)
$$\mathsf{supd}(\emptyset, x, T_{old}, T_{new}, C_{nd}) \rightsquigarrow\!\!\!\!\rightarrow \emptyset$$

(Supd Cons Neq)
$$\frac{x \neq y \qquad \mathsf{supd}(E_1, x, T_{old}, T_{new}, C_{nd}) \rightsquigarrow\!\!\!\!\rightarrow E}{\mathsf{supd}((E_1, y : T_1), x, T_{old}, T_{new}, C_{nd}) \rightsquigarrow\!\!\!\!\rightarrow (E, y : (T_1 \wedge \{\!|\, C_{nd} \,|\!\}))}$$

(Supd Cons Eq)
$$\frac{\mathsf{supd}(E_1, x, T_{old}, T_{new}, C_{nd}) \rightsquigarrow\!\!\!\!\rightarrow E}{\mathsf{supd}((E_1, x : T_{old}), x, T_{old}, T_{new}, C_{nd}) \rightsquigarrow\!\!\!\!\rightarrow (E, x : (T_{old} \wedge T_{new} \wedge \{\!|\, C_{nd} \,|\!\}))}$$

While rule (Sinfer Red) applies to any destructor and allows to transfer the type of the destructor to its result, for the asymmetric cryptography destructors check and dec we provide specialized rules that can infer the type of the result only from the type of one of the arguments of the destructor, provided that the result type is not tainted. Rule (Sinfer Check) only requires that the verification key in a check can be given type $\mathsf{VerKey}(T)$ for an untainted type $T$ in order to give the result of the check type $T$. The condition that $T$ is untainted is crucial for the soundness of this rule, however we cannot add it directly as a premise of the rule, since a negative premise of the form $E \nvdash T :: \mathsf{tnt}$ would break the weakening property of the type system, which requires that judgments are stable under environment extensions[16]. Instead we use the logical characterization of kinding and update the type of the result variable with the union type $\{\!|\, \mathsf{fkind}(E, T, \mathsf{tnt}) \,|\!\} \vee T$. If $T$ is not tainted, then this union type is equivalent to $\{\!|\, \mathsf{false} \,|\!\} \vee T$, which is in turn equivalent to just $T$. If $T$ is tainted, this union type is equivalent to $\top \vee T$, which is in turn equivalent to $\top$, and updating a variable to type $\top$ does not change anything because our strong environment update constructs the intersection between the old and the new type of the updated variable. This idea avoids negative statements and works well in the setting of security despite compromise, where a type being untainted is usually conditioned by certain participants not being compromised. Rule (Sinfer Dec) is very similar, if we know that the encryption passed to the dec destructor has type $\mathsf{PubEnc}(T)$ for some untainted $T$, then the decrypted term can be given type $T$. Similarly, rule (Sinfer Enc) allows us to infer type information about the message inside an encryption from the type of the whole encryption.

We illustrate rule (Sinfer Check) by returning to the simplified DAA protocol from §2.2 and examining in full detail how the statement-based inference judgment works on this example. The typing environment at the point of the verification is $E_{sdaa}$, while the initial typing information about the witnesses of the zero-knowledge proof is $E_0^{sdaa}$.

$E_{sdaa} = \mathit{Authenticate}, \mathit{OkTPM}, \mathit{Send}, k_i : \mathsf{SigKey}(T_{ki}), \mathit{Policy}_{sdaa}, x_z : \mathsf{Un}$

[16] The negation of judgments is clearly not stable under environment extensions.

$E_0^{sdaa} = x_f : \mathsf{Un}, x_{cert} : \mathsf{Un}, y_{vki} : (\mathsf{VerKey}(T_{ki}) \wedge \mathsf{Un} \wedge \mathsf{Un}), y_m : (\mathsf{Un} \wedge \mathsf{Un})$

The instance of the statement-based inference judgment looks as follows:

$$\frac{E_{sdaa}, E_0^{sdaa}, \{\mathsf{true}\} \vdash y_{vki} : \mathsf{VerKey}(T_{ki}) \quad E_{sdaa} \vdash T_{ki}\ ok \quad E_{sdaa} \vdash (E_0^{sdaa}, \mathsf{true})[x_f : \{\!|\, \mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt}) \,|\!\} \vee T_{ki}] \rightsquigarrow E_1^{sdaa}, C_1^{sdaa}}{E_{sdaa} \vdash [|\mathsf{check}(x_{cert}, y_{vki}) \rightsquigarrow x_f|]_{E_0^{sdaa}, \mathsf{true}} \rightsquigarrow E_1^{sdaa}, C_1^{sdaa}}$$

where the resulting environment $E_1^{sdaa}$ and formula $C_1^{sdaa}$ still need to be determined by looking at the following strong environment update derivation.

$$\frac{E_{sdaa} \vdash\ \mathsf{Un} \between \{\!|\, \mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt}) \,|\!\} \vee T_{ki} \rightsquigarrow \mathsf{false} \quad \mathsf{supd}(E_0^{sdaa}, x_f, \mathsf{Un}, \{\!|\, \mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt}) \,|\!\} \vee T_{ki}, \mathsf{false}) \rightsquigarrow E_1^{sdaa}}{E_{sdaa} \vdash (E_0^{sdaa}, \mathsf{true})[x_f : \{\!|\, \mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt}) \,|\!\} \vee T_{ki}] \rightsquigarrow E_1^{sdaa}, (\mathsf{true} \wedge \mathsf{false})}$$

We list the non-disjointness derivation below; when reading the derivation backwards we first apply (ND Entails), (ND Sym) and (ND Or), then on the left branch we use the derived rule (ND Forms Empty), while on the right branch we use the derived rules (ND Refine) and (ND Private Un).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots \quad E_{sdaa}, x : \{\!|\, \mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt}) \,|\!\} \vdash \mathsf{false}}{E_{sdaa} \vdash\ \{\!|\, \mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt}) \,|\!\} \between \mathsf{Un} \rightsquigarrow \mathsf{false}} \quad \dfrac{E_{sdaa} \vdash\ \mathsf{Private} \between \mathsf{Un} \rightsquigarrow \mathsf{false} \quad \dots}{E_{sdaa} \vdash\ T_{ki} \between \mathsf{Un} \rightsquigarrow \mathsf{false}}}{E_{sdaa} \vdash\ \{\!|\, \mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt}) \,|\!\} \vee T_{ki} \between \mathsf{Un} \rightsquigarrow \mathsf{false} \vee \mathsf{false}}}{E_{sdaa} \vdash\ \mathsf{Un} \between \{\!|\, \mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt}) \,|\!\} \vee T_{ki} \rightsquigarrow \mathsf{false} \vee \mathsf{false}}}{E_{sdaa} \vdash\ \mathsf{Un} \between \{\!|\, \mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt}) \,|\!\} \vee T_{ki} \rightsquigarrow \mathsf{false}}$$

For applying (ND Forms Empty) in the derivation above we compute the value of $\mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt})$ using the logical characterization of kinding from §2.4.5.

$\mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt})$
$= \mathsf{fkind}(E_{sdaa}, \{x_f : \mathsf{Private} \mid OkTPM(x_f)\}, \mathsf{tnt})$
$= \mathsf{fkind}(E_{sdaa}, \mathsf{Private}, \mathsf{tnt}) \wedge \mathsf{fentails}(E_{sdaa}, x'_f : \mathsf{Private}, OkTPM(x'_f))$
$\mathsf{fkind}(E_{sdaa}, \mathsf{Private}, \mathsf{tnt})$
$= \mathsf{fkind}(E_{sdaa}, \mathsf{Ch}(\{x : \mathsf{Un} \mid \mathsf{false}\}), \mathsf{tnt})$
$= \mathsf{fkind}(E_{sdaa}, \{x : \mathsf{Un} \mid \mathsf{false}\}, \mathsf{pub}) \wedge \mathsf{fkind}(E_{sdaa}, \{x : \mathsf{Un} \mid \mathsf{false}\}, \mathsf{tnt})$
$= \mathsf{fkind}(E_{sdaa}, \{x : \mathsf{Un} \mid \mathsf{false}\}, \mathsf{pub}) \wedge \mathsf{fkind}(E_{sdaa}, \mathsf{Un}, \mathsf{tnt}) \wedge \mathsf{fentails}((E_{sdaa}, z : \mathsf{Un}), \mathsf{false})$
$\mathsf{fentails}((E_{sdaa}, z : \mathsf{Un}), \mathsf{false})$
$= \forall Authenticate, OkTPM, Send, k_i, x_z, z.\, Policy_{sdaa} \Rightarrow \mathsf{false}$

The $\mathsf{fentails}((E_{sdaa}, z : \mathsf{Un}), \mathsf{false})$ conjunct allows us to derive $\mathsf{false}$ in an environment in which $Policy_{sdaa}$ holds, and $E_{sdaa}$ does contain $Policy_{sdaa}$. This together with the fact that $\mathsf{Private}$ and $\mathsf{Un}$ are disjoint allows us to obtain that $C_1^{sdaa} = \mathsf{true} \wedge \mathsf{false}$. Furthermore, by the definition of the auxiliary $\mathsf{supd}$ relation we obtain that

$$\begin{aligned} E_1^{sdaa} = {} & x_f : \mathsf{Un} \wedge (\{\!|\, \mathsf{fkind}(E_{sdaa}, T_{ki}, \mathsf{tnt}) \,|\!\} \vee T_{ki}) \wedge \{\!|\, \mathsf{false} \,|\!\}, \\ & x_{cert} : \mathsf{Un} \wedge \{\!|\, \mathsf{false} \,|\!\}, \\ & y_{vki} : (\mathsf{VerKey}(T_{ki}) \wedge \mathsf{Un} \wedge \mathsf{Un}) \wedge \{\!|\, \mathsf{false} \,|\!\}, \\ & y_m : (\mathsf{Un} \wedge \mathsf{Un}) \wedge \{\!|\, \mathsf{false} \,|\!\}. \end{aligned}$$

The environment $E_{sdaa}, E_1^{sdaa}, \{C_1^{sdaa}\}$ is inconsistent, corresponding to the intuition that in this protocol successfully verified zero-knowledge proofs cannot really come from the opponent. The checks in the (Proc Ver) typing rule

$$E_{sdaa}, E_1^{sdaa}, \{C_1^{sdaa}\} \vdash \mathit{Send}(x_f, y_m) \wedge \mathit{OkTPM}(x_f)$$
$$E_{sdaa}, E_1^{sdaa}, \{C_1^{sdaa}\} \vdash y_m : \mathsf{Un}$$

will therefore both succeed, which allows the type system to successfully type-check the verification process in the simplified DAA protocol.

The remaining two rules in the statement-based inference judgment (Sinfer And) and (Sinfer Or) deal with logical conjunctions and disjunctions in the zero-knowledge statement by reflecting them as conjunctions and disjunctions between types. We define two auxiliary judgments that combine two typing environments conjunctively or disjunctively.

**Combining environments:** $E_1 \otimes E_2 \rightsquigarrow E$ **for** $\otimes \in \{\wedge, \vee\}$

(Combine Empty)
$$\frac{}{\emptyset \otimes \emptyset \rightsquigarrow \emptyset}$$

(Combine Cons)
$$\frac{E_1 \otimes E_2 \rightsquigarrow E}{(E_1, x : T_1) \otimes (E_2, x : T_2) \rightsquigarrow E, x : (T_1 \otimes T_2)}$$

Since the entries in typing environments are themselves treated conjunctively, only $E_1 \wedge E_2 \rightsquigarrow E$ is fully precise, while $E_1 \vee E_2 \rightsquigarrow E$ is just a sound approximation that loses some precision. For instance given $E_1 = x : T_1, y : U_1$ and $E_2 = x : T_2, y : U_2$, if we apply $E_1 \vee E_2 \rightsquigarrow E$ we obtain the environment $E = x : T_1 \vee T_2, y : U_1 \vee U_2$, in which the correlation between $T_1$ and $U_1$ is lost (provided that all involved types are disjoint, if $x$ has type $T_1$ then it must be the case that $y$ has type $U_1$, while in this case the result environment only gives us $U_1 \vee U_2$ for $y$). This loss of precision in the disjunctive case is partially offset by the way we have defined the strong environment update judgment, which adds the non-disjointness information to all the entries in the environment. This should explain why in the simplified DAA example above there were four $\{\!|\,\mathsf{false}\,|\!\}$ conjuncts in $E_1^{sdaa}$.

While in rule (Sinfer Or) we simply infer typing information separately for each disjunct and then just combine it, in rule (Sinfer And) we first use the initial typing information to infer new information from each conjunct, we then use this inferred information as input when performing the whole inference process again on the other conjunct, and only in the end we combine the results. Propagating typing information between conjuncts is necessary because some of our strong inference rules like (Sinfer Red), (Sinfer Check) and (Sinfer Dec) rely on certain typing information being already present in the environment in order to fire, and some of that information is often inferred from the other conjuncts. Propagating this information in both directions ensures that the order in which the conjuncts are processed does not essentially affect the precision of the final outcome. This saves the user of our type system from having to manually arrange the conjuncts in

a way that respects all dependencies (at the cost of generating bigger types that express all possible dependencies).

We illustrate how rule (Sinfer And) works on the certificate chain of length two example from §2.3.2. The initial type information is captured by the environment

$$E_0^{\text{chain2}} = x_{vk} : \mathsf{Un}, x_{cert1} : \mathsf{Un}, x_{cert2} : \mathsf{Un}, y_{vk} : T_{vk} \wedge \mathsf{Un} \wedge \mathsf{Un}, y_m : \mathsf{Un} \wedge \mathsf{Un},$$

where $T_m = \{x : \mathsf{Un} \mid \mathit{Good}(x)\}$, $T_{vk2} = \mathsf{VerKey}(T_m)$, and $T_{vk} = \mathsf{VerKey}(T_{vk2})$. The instance of (Sinfer And) looks as follows.

$$\frac{\begin{array}{c} E \vdash [|\mathsf{check}(x_{cert2}, x_{vk}) \rightsquigarrow y_m|]_{E_0^{\text{chain2}},\mathsf{true}} \rightsquigarrow E_2, C_2 \\ E \vdash [|\mathsf{check}(x_{cert1}, y_{vk}) \rightsquigarrow x_{vk}|]_{E_2,C_2} \rightsquigarrow E_{12}, C_{12} \\ E \vdash [|\mathsf{check}(x_{cert1}, y_{vk}) \rightsquigarrow x_{vk}|]_{E_0^{\text{chain2}},\mathsf{true}} \rightsquigarrow E_1, C_1 \\ E \vdash [|\mathsf{check}(x_{cert2}, x_{vk}) \rightsquigarrow y_m|]_{E_1,C_1} \rightsquigarrow E_{21}, C_{21} \quad E_{12} \wedge E_{21} \rightsquigarrow E_1^{\text{chain2}} \end{array}}{E \vdash [|\mathsf{check}(x_{cert1}, y_{vk}) \rightsquigarrow x_{vk} \wedge \mathsf{check}(x_{cert2}, x_{vk}) \rightsquigarrow y_m|]_{E_0^{\text{chain2}},\mathsf{true}} \rightsquigarrow E_1^{\text{chain2}}, (C_{12} \wedge C_{21})}$$

We start by computing $E_2, C_2$ and $E_1, C_1$. Since the current type of $x_{vk}$ is just $\mathsf{Un}$ we cannot apply rule (Sinfer Check) when inferring $E_2, C_2$, so we default to rule (Sinfer Ident), which makes $E_2 = E_0^{\text{chain2}}$ and $C_2 = \mathsf{true}$. On the other hand, for inferring $E_1, C_1$ we can apply rule (Sinfer Check) as follows.

$$\frac{\begin{array}{c} E, E_0^{\text{chain2}}, \{\mathsf{true}\} \vdash y_{vk} : \mathsf{VerKey}(T_{vk2}) \quad E \vdash T_{vk2}\ ok \\ E \vdash (E_0^{\text{chain2}}, \mathsf{true})[x_{vk} : \{\!|\, \mathsf{fkind}(E, T_{vk2}, \mathsf{tnt}) \,|\!\} \vee T_{vk2}] \rightsquigarrow E_1, C_1 \end{array}}{E \vdash [|\mathsf{check}(x_{cert1}, y_{vk}) \rightsquigarrow x_{vk}|]_{E_0^{\text{chain2}},\mathsf{true}} \rightsquigarrow E_1, C_1}$$

The result of the strong environment update is obtained by rule (Strong Update).

$$\frac{\begin{array}{c} E \vdash \mathsf{Un} \between \{\!|\, \mathsf{fkind}(E, T_{vk2}, \mathsf{tnt}) \,|\!\} \vee T_{vk2} \rightsquigarrow C_{nd} \\ \mathsf{supd}(E_0^{\text{chain2}}, x_{vk}, \mathsf{Un}, \{\!|\, \mathsf{fkind}(E, T_{vk2}, \mathsf{tnt}) \,|\!\} \vee T_{vk2}, C_{nd}) \rightsquigarrow E_1 \end{array}}{E \vdash (E_0^{\text{chain2}}, \mathsf{true})[x_{vk} : T_{new}] \rightsquigarrow E_1, (\mathsf{true} \wedge C_{nd})}$$

Since verification keys are always public, the types $\mathsf{Un}$ and $T_{vk2}$ are not disjoint, so we obtain that $C_{nd} = \mathsf{true}$ by rule (ND True), and hence that $C_1 = \mathsf{true} \wedge \mathsf{true}$. In $E_1$ the only significant change compared to $E_0^{\text{chain2}}$ is for variable $x_{vk}$.

$$\begin{array}{l} E_1 = x_{vk} : \mathsf{Un} \wedge (\{\!|\, \mathsf{fkind}(E, T_{vk2}, \mathsf{tnt}) \,|\!\} \vee T_{vk2}) \wedge \{\!|\, \mathsf{true} \,|\!\}, \\ \qquad x_{cert1} : \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\}, x_{cert2} : \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\}, \\ \qquad y_{vk} : T_{vk} \wedge \mathsf{Un} \wedge \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\}, \\ \qquad y_m : \mathsf{Un} \wedge \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\} \end{array}$$

Since $T_{vk2}$ is not tainted the new type of $x_{vk}$ is equivalent to $\mathsf{Un} \wedge T_{vk2}$. This is crucial for applying rule (Sinfer Check) when inferring $E_{21}, C_{21}$.

$$\frac{\begin{array}{c} E, E_1, \{\mathsf{true} \wedge \mathsf{true}\} \vdash x_{vk} : \mathsf{VerKey}(T_m) \quad E \vdash T_m\ ok \\ E \vdash (E_1, \mathsf{true} \wedge \mathsf{true})[y_m : \{\!|\, \mathsf{fkind}(E, T_m, \mathsf{tnt}) \,|\!\} \vee T_m] \rightsquigarrow E_{21}, C_{21} \end{array}}{E \vdash [|\mathsf{check}(x_{cert2}, x_{vk}) \rightsquigarrow y_m|]_{E_1,\mathsf{true}\wedge\mathsf{true}} \rightsquigarrow E_{21}, C_{21}}$$

The result of the strong environment update is the following.

$$\frac{\begin{array}{c} E \vdash \mathsf{Un} \circledcirc \{\!|\, \mathsf{fkind}(E, T_m, \mathsf{tnt}) \,|\!\} \vee T_m \rightsquigarrow C'_{nd} \\ \mathsf{supd}(E_1, x, \mathsf{Un}, \{\!|\, \mathsf{fkind}(E, T_m, \mathsf{tnt}) \,|\!\} \vee T_m, C'_{nd}) \rightsquigarrow E_{21} \end{array}}{E \vdash (E_1, \mathsf{true} \wedge \mathsf{true})[x : \{\!|\, \mathsf{fkind}(E, T_m, \mathsf{tnt}) \,|\!\} \vee T_m] \rightsquigarrow E_{21}, (\mathsf{true} \wedge \mathsf{true} \wedge C'_{nd})}$$

Since the type $T_m$ is also public, it is not disjoint from $\mathsf{Un}$, so we apply rule (ND True) and obtain that $C'_{nd} = \mathsf{true}$, and thus $C_{21} = \mathsf{true} \wedge \mathsf{true} \wedge \mathsf{true}$. The typing environment $E_{21}$ gives a stronger type to $y_m$ and is equivalent to $E_1$ on the other variables.

$$\begin{aligned} E_{21} = {} & x_{vk} : \mathsf{Un} \wedge (\{\!|\, \mathsf{fkind}(E, T_{vk2}, \mathsf{tnt}) \,|\!\} \vee T_{vk2}) \wedge \{\!|\, \mathsf{true} \,|\!\} \wedge \{\!|\, \mathsf{true} \,|\!\}, \\ & x_{cert1} : \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\} \wedge \{\!|\, \mathsf{true} \,|\!\}, x_{cert2} : \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\} \wedge \{\!|\, \mathsf{true} \,|\!\}, \\ & y_{vk} : T_{vk} \wedge \mathsf{Un} \wedge \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\} \wedge \{\!|\, \mathsf{true} \,|\!\}, \\ & y_m : \mathsf{Un} \wedge \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\} \wedge (\{\!|\, \mathsf{fkind}(E, T_m, \mathsf{tnt}) \,|\!\} \vee T_m) \wedge \{\!|\, \mathsf{true} \,|\!\} \end{aligned}$$

Since $y_m$ already has a strong enough type, we simply apply (Sinfer Ident) for obtaining $E_{12} = E_0^{\text{chain2}}$ and $C_{12} = \mathsf{true}$. Combining $E_{12}$ and $E_{21}$ into the final $E_1^{\text{chain2}}$ does not change anything significant compared to what we already had in $E_{21}$.

$$\begin{aligned} E_1^{\text{chain2}} = {} & x_{vk} : \mathsf{Un} \wedge \mathsf{Un} \wedge (\{\!|\, \mathsf{fkind}(E, T_{vk2}, \mathsf{tnt}) \,|\!\} \vee T_{vk2}) \wedge \{\!|\, \mathsf{true} \,|\!\} \wedge \{\!|\, \mathsf{true} \,|\!\}, \\ & x_{cert1} : \mathsf{Un} \wedge \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\} \wedge \{\!|\, \mathsf{true} \,|\!\}, x_{cert2} : \mathsf{Un} \wedge \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\} \wedge \{\!|\, \mathsf{true} \,|\!\}, \\ & y_{vk} : T_{vk} \wedge \mathsf{Un} \wedge \mathsf{Un} \wedge T_{vk} \wedge \mathsf{Un} \wedge \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\} \wedge \{\!|\, \mathsf{true} \,|\!\}, \\ & y_m : \mathsf{Un} \wedge \mathsf{Un} \wedge \mathsf{Un} \wedge \mathsf{Un} \wedge \{\!|\, \mathsf{true} \,|\!\} \wedge (\{\!|\, \mathsf{fkind}(E, T_m, \mathsf{tnt}) \,|\!\} \vee T_m) \wedge \{\!|\, \mathsf{true} \,|\!\} \end{aligned}$$

Most importantly, since $T_m$ is not tainted the final type of $y_m$ is equivalent to $\mathsf{Un} \wedge T_m$, which allows us to transfer the predicate $\mathit{Good}(y_m)$ to the environment of the verifier. Unlike in the simplified DAA example, the final environment $E_1^{\text{chain2}}$ is consistent, so even if the proof were to come from the opponent, the signature checks in the zero-knowledge statement ensure that even if this is the case the predicate $\mathit{Good}(y_m)$ holds.

## 2.5. Machine-checked Robust Safety Proof

We have formalized our spi-calculus variant and our type system in the Coq proof assistant, and have proved formally that the type system enforces robust safety[17]. We remark that although this mechanized robust safety proof is still partial (the proofs of some helper lemmas are not assert-free), the formalization of this type system is important, since the complexity of the system makes the proof non-trivial, tedious, and error-prone. Indeed, this work has allowed us to discover several relatively small problems in the proofs of prior type systems with refinement types [FGM07a,BBF+08], as well as our own previous manual proofs [BHM08c, BGHM09], and to propose and evaluate fixes for the affected definitions and proofs.

[17] Full disclosure: On 21st of December 2011 we have discovered a flaw affecting the weakening property of this type system. We hope to fix this problem in future work.

Our formalization uses a locally nameless representation of binders [Gor93, ACP+08]: free variables and free spi-calculus names are represented in a named way, while bound variables and bound spi-calculus names are represented using de Bruijn indices [dB72]. As advertised by Aydemir et al. [ACP+08], in our formalization 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. While the locally nameless representation of binders and the cofinite quantification avoid the difficulties associated with alpha-renaming and are very important for the machine-checked proofs, for the sake of readability, in this section we will continue using the more standard named representation of binders.

Our Coq formalization[18] totals more than 16.9kLOC,[19] out of which more than 2.9kLOC are just definitions (our type system has more than 250 rules when also counting the well-formedness judgements). We have used Ott [SNO+10] to generate a large part of these definitions from a 1.7kLOC long Ott specification. We have used LNgen [AW10] to generate an additional 37.9kLOC of infrastructure lemmas, which proved invaluable when working with the locally nameless representation.

In the remainder of this section we present the high-level structure of the proofs and briefly discuss the main theorems and lemmas. Figure 2.1 on page 62 lists the names of the most important files in our Coq formalization, and displays the dependencies between the proofs (transitive dependencies are omitted).

### 2.5.1. Basic Properties

For the sake of brevity, we use $E \vdash \mathcal{J}$ to denote any typing judgment (i.e., $\mathcal{J} \in \{C, T :: k, T <: U, T \circledcirc U \rightsquigarrow C, M : T, D : T, P\}$).

The standard Weakening lemma states that all the judgments in our system are closed under well-formed extension of the typing environment.

**Lemma 2.4** (Weakening)**.** *If* $E, E' \vdash \mathcal{J}$ *and* $\vdash E, \mu, E'$ *ok then* $E, \mu, E' \vdash \mathcal{J}$.

The Tainted Entailed lemma proves that the formulas extracted out of tainted types are entailed by the current typing environment. For instance if the refinement type $\{x : \mathsf{Un} \mid C\}$ is tainted, then it has to be the case that the formula $\forall x.\, C$ is entailed. This property is used for proving Lemma 2.6 (Logical Subtyping), Lemma 2.10 (Tainted Bound), Lemma 2.14 (Pub Down), Lemma 2.15 (Tnt Up), Lemma 2.24 (Non-disjoint Tainted) and Lemma 2.25 (Non-disjoint).

**Lemma 2.5** (Tainted Entailed)**.** *If* $E \vdash T :: \mathsf{tnt}$ *and* $x \notin dom(E)$ *then* $E \vdash \forall x.\, forms_x(T)$.

[18] Available at `http://www.infsec.cs.uni-saarland.de/projects/zk-typechecker/`

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

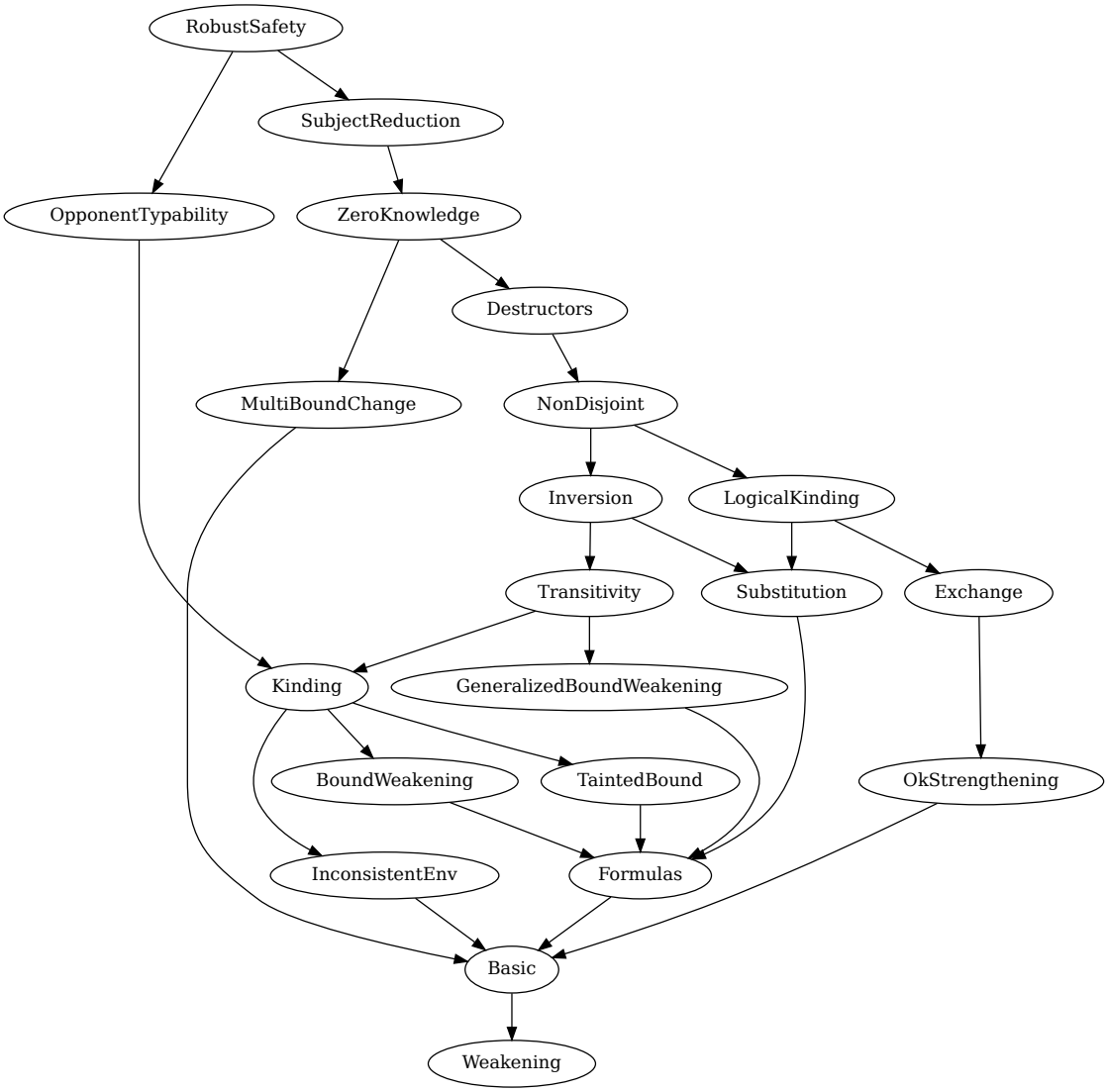



Figure 2.1.: Dependency graph of the most important results in our Coq formalization (transitive dependencies are omitted)

The Logical Subtyping lemma shows that if $T_1$ is a subtype of $T_2$ then the formulas extracted out of $T_2$ are implied by the formulas from $T_1$. For instance if $\{x : \mathsf{Un} \mid C_1\}$ is a subtype of $\{x : \mathsf{Un} \mid C_2\}$ then it must be the case that the formula $\forall x.\, C_1 \Rightarrow C_2$ is entailed by the environment. This result is used for proving Lemma 2.7 (Formulas) and Lemma 2.9 (Bound Weakening).

**Lemma 2.6** (Logical Subtyping)**.**
*If* $E \vdash T_1 <: T_2$ *and* $x \notin dom(E)$ *then* $E, x : T_1 \vdash forms_x(T_2)$.

The Formulas lemma states that the formulas extracted from any type $T$ hold when instantiated with any term having type $T$. This is useful for proving Lemma 2.12 (Substitution) for the entailment judgment, as well as for showing the soundness of rules (ND Gen) and (ND Forms And Type) in Lemma 2.25 (Non-disjoint).

**Lemma 2.7** (Formulas)**.** *If* $E \vdash M : T$ *and* $y \notin dom(E)$ *then* $E \vdash (forms_y(T))\{M/y\}$.

An important property of our type system is that typing becomes trivial if the environment becomes inconsistent. In an inconsistent environment all well-formed types are both public and tainted, so by (Sub Pub Tnt) all well-formed types are equivalent, any well-formed term or destructor has any well-formed type, and any well-formed process is also well-typed.

**Lemma 2.8** (Inconsistent Environment)**.** *If* $E \vdash \mathsf{false}$ *and all components of* $\mathcal{J}$ *are well formed in* $E$ *then* $E \vdash \mathcal{J}$.

The Bound Weakening lemma allows to replace a variable binding $x : T$, with another one $x : T'$ in any judgment, provided that $T'$ is a subtype of $T$. Note, that the direction of subtyping is reversed with respect to (Term Subsum) since the typing environment stores premises, and making the premises stronger weakens the judgment as a whole.

**Lemma 2.9** (Bound Weakening)**.**
*If* $E \vdash T' <: T$ *and* $E, x : T, E' \vdash \mathcal{J}$ *then* $E, x : T', E' \vdash \mathcal{J}$.

In case the type $U$ in a variable binding $x : U$ is tainted, we can replace it with any other type, not just subtypes of $U$ like in Lemma 2.9 (Bound Weakening). By Lemma 2.5 (Tainted Entailed) we know that the formulas extracted from $U$ are entailed by the preceding environment entries, so we use the (Cut) and (Mon) properties of the authorization logic to obtain this result.

**Lemma 2.10** (Tainted Bound)**.**
*If* $E, x : U, E' \vdash T :: k$ *and* $E \vdash U :: \mathsf{tnt}$ *and* $E \vdash U'$ *ok then* $E, x : U', E' \vdash T :: k$.

Similarly, we can remove environment entries $\{C\}$ if the formula $C$ is entailed by the preceding environment entries.

**Lemma 2.11** (OK Strengthening)**.** *If* $E, \{C\}, E' \vdash \mathcal{J}$ *and* $E \vdash C$ *then* $E, E' \vdash \mathcal{J}$.

The Substitution lemma does a type-preserving substitution of a variable not only in the right-hand-side of a judgment, but also in the succeeding environment entries (this is necessary because of type dependencies).

**Lemma 2.12** (Substitution)**.**
*If* $E, x : T, E' \vdash \mathcal{J}$ *and* $E \vdash N : T$ *then* $E, (E'\{N/x\}) \vdash (\mathcal{J}\{N/x\})$.

Finally, the Exchange lemma allows us to swap independent environment bindings.

**Lemma 2.13** (Exchange)**.** *If* $E, \mu_1, \mu_2, E' \vdash \mathcal{J}$ *and* $\vdash E, \mu_2, \mu_1$ *ok then* $E, \mu_2, \mu_1, E' \vdash \mathcal{J}$.

### 2.5.2. Transitivity of Subtyping

Transitivity of subtyping depends on the following relatively standard lemmas [BBF+08].

**Lemma 2.14** (Pub Down)**.** *If* $E \vdash T <: T'$ *and* $E \vdash T' :: \mathsf{pub}$ *then* $E \vdash T :: \mathsf{pub}$.

**Lemma 2.15** (Tnt Up)**.** *If* $E \vdash T <: T'$ *and* $E \vdash T :: \mathsf{tnt}$ *then* $E \vdash T' :: \mathsf{tnt}$.

**Lemma 2.16** (Public)**.** $E \vdash T :: \mathsf{pub}$ *if and only if* $E \vdash T <: \mathsf{Un}$.

**Lemma 2.17** (Tainted)**.** $E \vdash T :: \mathsf{tnt}$ *if and only if* $E \vdash \mathsf{Un} <: T$.

The transitivity proof requires generalizing the induction hypothesis to handle contravariant type constructors and dependent types. For the latter we extend the subtyping relation to environments using the following judgment:

**Subtyping Environments:** $E' <: E$

$$\frac{}{\emptyset <: \emptyset} \qquad \frac{E <: E'}{E, \mu <: E', \mu} \qquad \frac{E <: E' \quad E \vdash T <: T'}{E, x : T <: E', x : T'}$$

**Lemma 2.18** (Transitivity Generalized)**.**
*For all* $E$, $T_1$ *and* $T_2$ *so that* $E \vdash T_1 <: T_2$ *we have that:*

1. *for all* $T_3$ *and* $E'$ *so that* $E' <: E$ *and* $E' \vdash T_2 <: T_3$ *we have that* $E' \vdash T_1 <: T_3$*;*
2. *and for all* $T_3$ *and* $E'$ *so that* $E' <: E$ *and* $E' \vdash T_3 <: T_1$ *we have that* $E' \vdash T_3 <: T_2$.

**Corollary 2.19** (Transitivity of Subtyping)**.**
*If* $E \vdash T_1 <: T_2$ *and* $E \vdash T_2 <: T_3$ *then* $E \vdash T_1 <: T_3$.

### 2.5.3. Logical Characterization of Kinding

We have proved that our logical characterization of kinding (§2.4.5) is sound and complete with respect to our inductive kinding judgement (§2.4.4). As an intermediate step we show that our logical characterization of entailment is sound and complete. These proofs crucially rely on the use of second-order quantifiers in the definition of $\mathsf{fentails}(E, C)$ to ensure that each entailment check is independent from the others.

**Lemma 2.20** (Soundness of fentails)**.** *If* $E \vdash C$ *then* $\models \mathsf{fentails}(E, C)$.

**Lemma 2.21** (Completeness of fentails)**.** *If* $E \vdash C$ *ok and* $\models \mathsf{fentails}(E, C)$ *then* $E \vdash C$.

**Lemma 2.22** (Soundness of fkind)**.** *If* $E \vdash T :: k$ *then* $\models \mathsf{fkind}(E, T, k)$.

**Lemma 2.23** (Completeness of fkind)**.** *If* $E \vdash T$ *ok and* $\models \mathsf{fkind}(E, T, k)$ *then* $E \vdash T :: k$.

### 2.5.4. Non-disjointness of Types

We start by proving that the formula obtained for two tainted types by the non-disjointness judgment is entailed by the typing environment.

**Lemma 2.24** (Non-disjoint Tainted)**.**
*If* $E \vdash T \between U \rightsquigarrow C_{nd}$ *and* $E \vdash T :: \mathsf{tnt}$ *and* $E \vdash U :: \mathsf{tnt}$ *then* $E \vdash C_{nd}$.

More importantly, the Non-disjoint lemma provides the characteristic property of the non-disjointness judgement from §2.4.6. It states that if the non-disjointness judgement says that types $T_1$ and $T_2$ overlap in environment $E$ implies formula $C_{nd}$ ($E \vdash T_1 \between T_2 \rightsquigarrow C_{nd}$) and we can find a closed term $M$ that inhabits both $T_1$ and $T_2$, then the formula $C_{nd}$ is indeed entailed by $E$.

**Lemma 2.25** (Non-disjoint)**.**
*If* $E \vdash T_1 \between T_2 \rightsquigarrow C_{nd}$ *and* $E \vdash M : T_1$ *and* $E \vdash M : T_2$ *and* $fv(M) = \emptyset$ *then* $E \vdash C_{nd}$.

### 2.5.5. Destructor Consistency

One crucial step for the subject-reduction proof is proving type preservation for the destructor reduction relation. This relies on a very large number of syntactic inversion lemmas. Proving consistency is particularly challenging for asymmetric cryptography (the check and dec destructors) because of the permissive kinding and subtyping rules for the involved types.

**Lemma 2.26** (Check consistent)**.**
*For all closed terms* $N$ *and* $K$ *such that* $E \vdash \mathsf{sign}(N, K) : \mathsf{Signed}(T)$ *and* $E \vdash \mathsf{vk}(K) : \mathsf{VerKey}(T)$ *we have that* $E \vdash N : T$.

**Lemma 2.27** (Dec consistent)**.**
*For all closed terms $N$ and $K$ such that $E \vdash \mathsf{enc}(N, \mathsf{ek}(K)) : \mathsf{PubEnc}(T)$ and $E \vdash K : \mathsf{DecKey}(T)$ we have that $E \vdash N : T$.*

**Lemma 2.28** (SDec consistent)**.**
*For all closed terms $N$ and $K$ such that $E \vdash \mathsf{senc}(N, K) : U$ and $E \vdash K : \mathsf{SymKey}(T)$ we have that $E \vdash N : T$.*

**Lemma 2.29** (Destructor Consistency)**.**
*For all closed destructors $D$ such that $D \Downarrow N$ and $E \vdash D : T$ we have $E \vdash N : T$.*

### 2.5.6. Zero-knowledge

The correctness proof of the statement-based inference judgement from §2.4.9 requires a fairly sophisticated inductive invariant. The judgement $E \vdash [|B|]_{E_{old}, C_{old}} \rightsquigarrow E_{new}, C_{new}$ takes $B$, the basic formula of the statement we are verifying, and an initial typing environment $E, E_{old}, \{C_{old}\}$, and infers a new typing environment $E, E_{new}, \{C_{new}\}$. We require that the basic formula $B$ is well-formed in environment $E, E_{old}$, but only references variables in $E_{old}$, i.e., it only references the arguments of the zero-knowledge proof we are verifying. We also require that $E_{old}$ is a special well-formed environment extension of $E$ (judgement $E \vdash E_{old}$ *spec* in §2.4.9), which implies that $E_{old}$ only contains variable bindings, that $E_{old}$ does not have any dependencies to itself, and that the compound environment $E, E_{old}$ is well-formed. We also require that all the arguments of the zero-knowledge proof $\widetilde{M}$ have type $\mathsf{Un}$[20] and additionally have the types listed in $E_{old}$. The latter requirement is formalized by the $E \vdash \widetilde{M} : E'$ judgement below.

**Closed terms $\widetilde{M}$ have the types in $E'$: $E \vdash \widetilde{M} : E'$**

(TE Empty)
$$\frac{}{E \vdash \emptyset : \emptyset}$$

(TE Cons)
$$\frac{E \vdash \widetilde{M} : E' \quad E \vdash M : T \quad \emptyset \vdash M\ ok}{E \vdash (\widetilde{M}, M) : (E', x : T)}$$

Finally, we require that the formula $C_{old}$ and the basic formula $B$[21] both hold for arguments $\widetilde{M}$. If all these requirement hold, Lemma 2.30 (Statement-based Inference Correct) proves that the inferred environment $E_{new}$ has exactly the same bindings as $E_{old}$ (formalized by the $E_1 \simeq E_2$ judgement below), that the arguments $\widetilde{M}$ also have the new types in $E_{new}$, and that the formula $C_{new}$ holds for these arguments.

[20] This assumption is justified since the statement-based inference judgement is only used to deal with the case when the zero-knowledge proof was created by an opponent.

[21] The basic formula $B$ of the zero-knowledge statement can be assumed to hold since statement-based inference is only used to deal with the case when the zero-knowledge verification succeeds.

**Environments with the same variable bindings:** $E_1 \simeq E_2$

| (EE Empty) | (EE Cons) |
|---|---|
| | $\dfrac{E_1 \simeq E_2}{E_1, x : T_1 \simeq E_2, x : T_2}$ |
| $\dfrac{}{\emptyset \simeq \emptyset}$ | |

**Lemma 2.30** (Statement-based Inference Correct)**.** *If* $E \vdash [|B|]_{E_{old}, C_{old}} \rightsquigarrow E_{new}, C_{new}$ *and* $E, E_{old} \vdash B$ *ok and* $fv(B) \subseteq dom(E_{old})$ *and* $E \vdash E_{old}$ *spec and* $E \vdash \widetilde{M} : \widetilde{\mathsf{Un}}$ *and* $E \vdash \widetilde{M} : E_{old}$ *and* $E \vdash C_{old}\{\widetilde{M}/E_{old}\}$ *and* $B\{\widetilde{M}/E_{old}\}$ *valid then* $E_{old} \simeq E_{new}$ *and* $E \vdash \widetilde{M} : E_{new}$ *and* $E \vdash C_{new}\{\widetilde{M}/E_{old}\}$.

### 2.5.7. Subject-reduction

Type preservation constitutes an important step towards proving Theorem 2.34 (Safety). We show that both structural equivalence and reduction preserve typing.

**Lemma 2.31** (Structural Equivalence Preserves Typing)**.**
*If* $E \vdash P$ *and* $P \equiv P'$ *then* $E \vdash P'$

**Theorem 2.32** (Reduction Preserves Typing)**.**
*If* $fv(P) = \emptyset$ *and* $E \vdash P$ *and* $P \to P'$ *then* $E \vdash P'$

### 2.5.8. Robust Safety

We call $E$ an opponent environment if all names and variables in $E$ are associated type $\mathsf{Un}$. For the Opponent Typability lemma we require that the process $P$ is a well-formed possibly-open opponent (i.e., a well-formed process without any $\mathsf{assert}$ and for which all restrictions are annotated with $\mathsf{Un}$) in an opponent environment.

**Lemma 2.33** (Opponent Typability)**.**
*If* $E$ *is an opponent environment and* $E \vdash P$ *opp then* $E \vdash P$

The proof of the Safety theorem uses Theorem 2.32 (Reduction Preserves Typing) and Lemma 2.31 (Structural Equivalence Preserves Typing).

**Theorem 2.34** (Safety)**.**
*If* $E$ *is of the form* $a_1 : T_1, \ldots, a_n : T_n$*, and* $E \vdash P$ *then* $P$ *is safe.*

Finally, the Robust Safety theorem follows immediately from Lemma 2.33 (Opponent Typability) and Theorem 2.34 (Safety).

**Theorem 2.35** (Robust Safety)**.**
*If* $E$ *is of the form* $a_1 : \mathsf{Un}, \ldots, a_n : \mathsf{Un}$*, and* $E \vdash P$ *then* $P$ *is robustly safe.*

## 2.6. Case Study: Achieving Security Despite Compromise Using Zero-knowledge

An important challenge when designing and analyzing cryptographic protocols is the enforcement of security properties in the presence of compromised participants. In the setting of logic-based authorization policies, the notion of "security despite compromise" [FGM07a] captures the intuition that *an invalid authorization decision by an uncompromised participant should only arise if participants on which the decision logically depends are compromised.* The impact of participant compromise should be thus apparent from the authorization policy, without having to study the details of the protocol.

Zero-knowledge proofs are a natural candidate for strengthening protocols so that they achieve security despite compromise, since they allow the participants to prove that they correctly generated the messages they send, without revealing any secret data. In another work [Gro09, BGHM09], we have introduced a general technique for strengthening cryptographic protocols in order to satisfy authorization policies despite participant compromise. We automatically transform the original cryptographic protocols by adding non-interactive zero-knowledge proofs, so that each participant proves that the messages sent to the other participants are generated in accordance to the protocol. The zero-knowledge proofs are forwarded to ensure the correct behavior of all participants involved in the protocol, without revealing any secret data. Moreover, the transformation automatically derives type annotations for the strengthened protocol from the type annotations of the original protocol.

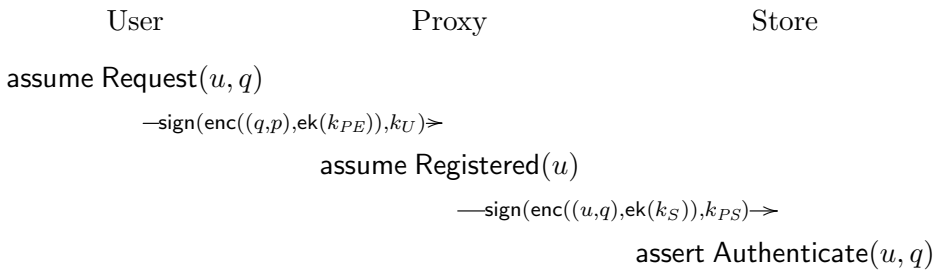
We use our type-checker for zero-knowledge to validate the protocols produced by this transformation. This use case has motivated some of the important design decisions in our type system. Most importantly, as explained in §1.1 and §2.4.9, our technique for type-checking zero-knowledge crucially relies on honest provers being type-checked, and on honest verifiers being able to infer that a proof comes from an honest prover by deducing that one of the witnesses of the proof has a type whose values are not known to the attacker. In the setting of security despite compromise though, all this reasoning has to be conditioned by certain participants being indeed honest.

In order to address this challenge our type system has no unconditionally secure types. Instead, we use logical formulas that precisely characterize when a type is compromised (see §2.4.5). We use refinement types that contain such logical formulas together with union types to express type information that is conditioned by a participant not being compromised. Such conditional types are inferred automatically by the statement-based inference judgment from §2.4.9.

In the remainder of this section we illustrate the transformation from [BGHM09] by means of a simple example, and we explain how our type system handles the original protocol and, more interestingly, the protocol generated by the transformation.

### 2.6.1. Illustrative Example

As a running example, we consider a simple protocol involving a user, a proxy, and an online store. This is inspired by a protocol proposed by Fournet et al. [FGM07a]. The main difference is that we use asymmetric cryptography in the first message, while the original protocol uses symmetric encryption.




In this protocol, the user $u$ sends a query $q$ and a password $p$ to the proxy. This data is first encrypted with the public key $\mathsf{ek}(k_{PE})$ of the proxy and then signed with $u$'s signing key $k_U$. The proxy verifies the signature and decrypts the message, checks that the password is correct, and sends the user's name and the query to the online store. This data is first encrypted with the public key $\mathsf{ek}(k_S)$ of the store and then signed with the signing key $k_{PS}$ of the proxy.

The protocol is decorated with two assumptions and one assertion: the assumption $\mathsf{Request}(u, q)$ states that the user $u$ is willing to send a query $q$, the assumption $\mathsf{Registered}(u)$ states that the user $u$ is registered in the system, and the assertion $\mathsf{Authenticate}(u, q)$ states that the store authenticates the query $q$ sent by user $u$.

The goal of this protocol is that the online store authenticates the query $q$ as coming from $u$ only if $u$ has indeed sent query $q$ and $u$ is registered in the system. This is formulated as the following authorization policy:

$$\forall u, q.\ \mathsf{Request}(u, q) \wedge \mathsf{Registered}(u) \Rightarrow \mathsf{Authenticate}(u, q) \tag{2.1}$$

We want $\mathsf{Authenticate}(u, q)$ to be entailed in all executions of the protocol that reach the assert. Since the only way to obtain this predicate is by using policy (2.1), which only applies if the assertions $\mathsf{Request}(u, q)$ and $\mathsf{Registered}(u)$ have been previously executed, this policy enforces that the store authenticates $q$ only if a registered user requested $q$.

#### Typing the Original Protocol (Uncompromised Setting)

We illustrate our type system on this protocol. Since the query $q$ the user sends to the proxy is not secret, but authentic, we give it type $\{x_q : \mathsf{Un} \mid \mathsf{Request}(u, x_q)\}$. The

password $p$ is of course secret and is given type $\mathsf{Private}$. The payload sent by the user, the pair $(q, p)$, can therefore be typed to $T_1 = \mathsf{Pair}(\{x_q : \mathsf{Un} \mid \mathsf{Request}(u, x_q)\}, \mathsf{Private})$. The public key of the proxy $\mathsf{ek}(k_{PE})$ is used to encrypt messages of type $T_1$ so we give it type $\mathsf{EncKey}(T_1)$. Similarly, the signing key of the user $k_U$ is used to sign the term $\mathsf{enc}((q, p), \mathsf{ek}(k_{PE}))$, so we give it the type $\mathsf{SigKey}(\mathsf{PubEnc}(T_1))$, while the corresponding verification key $\mathsf{vk}(k_U)$ has type $\mathsf{VerKey}(\mathsf{PubEnc}(T_1))$. Once the proxy verifies the signature using $\mathsf{vk}(k_U)$, decrypts the result using $k_{PE}$, and splits the pair into $q$ and $p$ it can be sure not only that $q$ is of type $\mathsf{Un}$ and $p$ is of type $\mathsf{Private}$, but also that $\mathsf{Request}(u, q)$ holds, i.e., the user has indeed issued a request.

In a very similar way, the signing key of the proxy $k_{PS}$ is given type $\mathsf{SigKey}(\mathsf{PubEnc}(T_2))$, where $T_2$ is the dependent pair type $\langle x_u : \mathsf{Un}, x_q : \mathsf{Un}\rangle\{\mathsf{Request}(x_u, x_q) \wedge \mathsf{Registered}(x_u)\}$, which conveys the conjunction of two logical predicates. If the store successfully checks the signature using $\mathsf{vk}(k_{PS})$ the resulting message will have type $\mathsf{PubEnc}(T_2)$. Since $k_S$ has type $\mathsf{DecKey}(T_2)$ it can be used to decrypt this message and obtain the user name $u$ and the query $q$, for which $\mathsf{Request}(u, q) \wedge \mathsf{Registered}(u)$ holds. By the authorization policy given above, this logically implies $\mathsf{Authenticate}(u, q)$. The authentication request is thus justified by the policy, so if all participants are honest the original protocol is secure (i.e., robustly safe with respect to authorization policy (2.1)).

### 2.6.2. Compromising Participants

We now investigate what happens if some of the participants are compromised. We model the compromise of a participant $v$ by $(a)$ revealing all her secrets to the attacker; $(b)$ removing the code of $v$, since it is controlled by the attacker; and $(c)$ introducing the assumption $\mathsf{Compromised}(v)$. Since the attacker can impersonate $v$ and send messages on her behalf without assuming any predicate, we make the convention that for each assumption $F$ in the code of $v$ we have a rule of the form $\mathsf{Compromised}(v) \Rightarrow \forall\widetilde{x}.F$ in the authorization policy. In our example we have two such additional rules:

$$\mathsf{Compromised}(\mathit{proxy}) \Rightarrow \forall u.\mathsf{Registered}(u). \tag{2.2}$$

$$\mathsf{Compromised}(\mathit{user}) \Rightarrow \forall q.\mathsf{Request}(u, q) \tag{2.3}$$

With these additional rules the protocol is robustly safe even when the user is compromised, since the only way for the attacker to interact with the honest proxy is to follow the protocol and, by impersonating the user, to authenticate a query with a valid password. This is, however, harmless since the attacker is just following the protocol. The protocol is vacuously safe if the store is compromised, since no assertion has to be justified; moreover, it is safe if both the proxy and the user are compromised, since in this case the two hypotheses of (2.1) are always entailed.

Therefore the only interesting case is when the proxy is compromised and the other participants are not. In this case, we introduce the assumption $\mathsf{Compromised}(\mathit{proxy})$, which by (2.2) implies that $\forall u.\mathsf{Registered}(u)$. Still, the compromised proxy might send

a message to the store without having received any query from the user, which would lead to an $\mathsf{Authenticate}(u, q)$ assertion that is not logically entailed by the preceding assumptions. Notice that the only way to infer $\mathsf{Authenticate}(u, q)$ is using (2.1), and this requires that both $\mathsf{Request}(u, q)$ and $\mathsf{Registered}(u)$ hold. However, since the user did not issue a request, the $\mathsf{Request}(u, q)$ predicate is not entailed in the system.

As suggested in [FGM07a], we could document the attack by weakening the authorization policy. This could be achieved by introducing a new rule stating that if the proxy is compromised, then $\forall u, q.\ \mathsf{Request}(u, q)$ holds. We take a different approach and, instead of weakening the authorization policy and accepting the attack, we propose a general methodology to strengthen any protocol so that such attacks are prevented [BGHM09]. In §2.6.3 we strengthen our example protocol and use our type system for zero-knowledge to analyze the strengthened version.

### Adjusting Types for Compromise Scenarios

As explained above, when a participant $v$ is compromised all its secrets are revealed to the attacker and the predicate $\mathsf{Compromised}(v)$ is added to the environment. However, we need to make the types of $p$'s secrets public, in order to be able to reveal them to the attacker. For instance, in our running example, when compromising the proxy the type of the decryption key $k_{PE}$ needs to be made public. However, once we replace the type annotation of this key from $\mathsf{DecKey}(T_1)$ to $\mathsf{Un}$, other types need to be changed as well. The type of the signing key of the user $k_U$ is used to sign an encryption done with $\mathsf{ek}(k_{PE})$, so one could change the type of $k_U$ from $\mathsf{SigKey}(\mathsf{PubEnc}(T_1))$ to $\mathsf{SigKey}(\mathsf{PubEnc}(\mathsf{Un}))$, which is actually equivalent to $\mathsf{Un}$. This type would be, however, weaker than necessary. The fact that the store is compromised does not affect the fact that the user assumes $\mathsf{Request}(u, q)$, so we can give $k_U$ type $\mathsf{SigKey}(\mathsf{PubEnc}(T_1'))$, where $T_1' = \mathsf{Pair}(\{x_q : \mathsf{Un} \mid \mathsf{Request}(u, x_q)\}, \mathsf{Un})$. Similar changes need to be done manually for the other type annotations, resulting in a specification that differs from the original uncompromised one only with respect to the type annotations.

However, having two different specifications that need to be kept in sync would be error prone. As proposed by Fournet et al. [FGM07a], we use only one set of type annotations for both the uncompromised and the compromised scenarios, containing types that are secure only under the condition that certain participants are uncompromised.

### Typing the Original Protocol (Compromised Setting)

We illustrate this on our running example. The type of the payload sent by the user, which used to be $T_1 = \mathsf{Pair}(\{x_q : \mathsf{Un} \mid \mathsf{Request}(u, x_q)\}, \mathsf{Private})$, is now changed to $T_1^* = \mathsf{Pair}(\{x_q : \mathsf{Un} \mid \mathsf{Request}(u, x_q)\}, \mathsf{PrivateUnless}(P))$. In the uncompromised setting $\neg\mathsf{Compromised}(\mathit{proxy})$ is entailed in the system, type $\mathsf{PrivateUnless}(P)$ is equivalent to $\mathsf{Private}$, and $T_1^*$ is equivalent to $T_1$. However, if the proxy is compromised then the

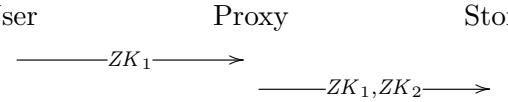
predicate $\mathsf{Compromised}(\mathit{proxy})$ is entailed, $\mathsf{PrivateUnless}(P)$ is equivalent to $\mathsf{Un}$ and $T_1^*$ is equivalent to $T_1'$. Using this we can give $k_U$ type $\mathsf{SigKey}(\mathsf{PubEnc}(T_1^*))$ and $k_{PE}$ type $\mathsf{DecKey}(T_1^*)$.

In the uncompromised setting, the payload sent by the proxy has type $T_2 = \langle x_u : \mathsf{Un}, x_q : \mathsf{Un}\rangle\{\mathsf{Request}(x_u, x_q) \wedge \mathsf{Registered}(x_u)\}$. However, once the proxy is compromised, the attacker can replace this payload with a message of his choice, so the type of this payload becomes $\mathsf{Un}$. In order to be able to handle both scenarios we give this payload the union type $T_2^* = \{x : T_2 \mid \neg\mathsf{Compromised}(\mathit{proxy})\} \vee \{x : \mathsf{Un} \mid \mathsf{Compromised}(\mathit{proxy})\}$. The types of $k_{PS}$ and $k_S$ are updated accordingly.

With these changed annotations in place we can still successfully type-check the example protocol in the case all participants are honest, but in addition we can also try to check the protocol in case some of the participants are compromised. If only the proxy is compromised type-checking will, however, fail, since the store is going to obtain a payload of type $T_2^*$. Since the proxy is compromised, $T_2^*$ is equivalent to $\mathsf{Un}$, and provides no guarantees that could justify the authentication of the request. This is not surprising since, as explained above, the original protocol is not secure if the proxy is compromised.

### 2.6.3. Strengthened Protocol

The central idea of our technique [Gro09,BGHM09] is to replace each message exchanged in the protocol with a non-interactive zero-knowledge proof showing that the message has been correctly generated. Additionally, zero-knowledge proofs are forwarded by each participant in order to allow the others to independently check that all the participants have followed the protocol. For instance, the protocol considered before is transformed as follows:




$$S_1 \triangleq \mathsf{witness}\ \alpha_1, \alpha_2\ \mathsf{public}\ \beta_1, \beta_2, \beta_3, \beta_4\ \mathsf{in}\ \beta_4 = \mathsf{enc}((\alpha_1, \alpha_2), \beta_2) \wedge \mathsf{check}(\beta_3, \beta_1) \rightsquigarrow \beta_4$$

$$ZK_1 \triangleq \mathsf{zk}_{S_1}\big(\overbrace{q, p}^{\alpha_1,\alpha_2}; \overbrace{\mathsf{vk}(k_U)}^{\beta_1}, \overbrace{\mathsf{ek}(k_{PE})}^{\beta_2}, \overbrace{\mathsf{sign}\big(\mathsf{enc}\big((q,p), \mathsf{ek}(k_{PE})\big), k_U\big)}^{\beta_3}, \overbrace{\mathsf{enc}\big((q,p), \mathsf{ek}(k_{PE})\big)}^{\beta_4}\big)$$

$$S_2 \triangleq \mathsf{witness}\ \alpha_1, \alpha_2, \alpha_3\ \mathsf{public}\ \beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6, \beta_7, \beta_8, \beta_9\ \mathsf{in}\ \mathsf{check}(\beta_5, \beta_4) \rightsquigarrow \beta_9$$
$$\wedge \mathsf{dec}(\beta_9, \alpha_3) \rightsquigarrow (\alpha_1, \alpha_2) \wedge \beta_3 = \mathsf{ek}(\alpha_3) \wedge \beta_7 = \mathsf{enc}((\beta_8, \alpha_1), \beta_2) \wedge \mathsf{check}(\beta_6, \beta_1) \rightsquigarrow \beta_7$$

$$ZK_2 \triangleq \mathsf{zk}_{S_2}\big(\overbrace{q, p, k_{PE}}^{\alpha_1,\alpha_2,\alpha_3}; \overbrace{\mathsf{vk}(k_{PS})}^{\beta_1}, \overbrace{\mathsf{ek}(k_S)}^{\beta_2}, \overbrace{\mathsf{ek}(k_{PE})}^{\beta_3}, \overbrace{\mathsf{vk}(k_U)}^{\beta_4}, \overbrace{\mathsf{sign}\big(\mathsf{enc}\big((q,p), \mathsf{ek}(k_{PE})\big), k_U\big)}^{\beta_5},$$
$$\overbrace{\mathsf{sign}\big(\mathsf{enc}\big((u,q), \mathsf{ek}(k_S)\big), k_{PS}\big)}^{\beta_6}, \overbrace{\mathsf{enc}\big((u,q), \mathsf{ek}(k_S)\big)}^{\beta_7}, \overbrace{u}^{\beta_8}, \overbrace{\mathsf{enc}\big((q,p), \mathsf{ek}(k_{PE})\big)}^{\beta_9}\big)$$

The first zero-knowledge proof states that the message $\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{ek}(k_{PE})\big),k_U\big)$ sent by the user complies with the protocol specification: the verification of this message with the user's verification key succeeds ($\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4$) and the result is the encryption of the query and the password with the proxy's encryption key ($\beta_4=\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)$). We model proofs of knowledge, so the user proves to know the secret query $\alpha_1$ and the secret password $\alpha_2$, not just that they exist.

The public arguments of the proof were all public in the original protocol. The query and the password are witnesses since they were encrypted in the original protocol and could be secrets[22]. Furthermore, notice that the statement $S_1$ simply describes the operations performed by the user, except for the signature generation which is replaced by the signature verification (this is necessary to preserve the secrecy of the signing key). In general, the statement of the generated zero-knowledge proof is computed as the conjunction of the individual operations performed to produce the output message.

The second zero-knowledge proof states that the message $\beta_5$ received from the user complies with the protocol, i.e., it is the signature ($\mathsf{check}(\beta_5,\beta_4)\rightsquigarrow\beta_9$)) of an encryption of two secret terms $\alpha_1$ and $\alpha_2$ ($\mathsf{dec}(\beta_9,\alpha_3)\rightsquigarrow(\alpha_1,\alpha_2)$). The zero-knowledge proof additionally ensures that the message $\beta_6$ sent by the proxy is the signature ($\mathsf{check}(\beta_6,\beta_1)\rightsquigarrow\beta_7$) of an encryption of the user's name and the query $\alpha_1$ received from the user ($\beta_7=\mathsf{enc}((\beta_8,\alpha_1),\beta_2)$). The statement $S_2$ guarantees that the query $\alpha_1$ signed by the user is the same as the one signed by the proxy. Also notice that the proof does not reveal the secret password $\alpha_2$ received from the user.

The resulting protocol is secure despite compromise, since a compromised proxy can no longer cheat the store by pretending to have received a query from the user. The query will be authenticated only if the store can verify the two zero-knowledge proofs sent by the proxy, and the semantics of these proofs ensures that the proxy is able to generate a valid proof only it has previously received the query from the user.

### Typing the Strengthened Protocol (Compromised Setting)

We use our type system to show that the automatically strengthened protocol above is robustly safe with respect to its original authorization policy (2.1, 2.2, and 2.3). The zero-knowledge proof $ZK_1$ sent by the user to the proxy in the strengthened protocol, which was defined above as:

$$\mathsf{zk}_{S_1}\big(\overbrace{q,p}^{\alpha_1,\alpha_2};\overbrace{\mathsf{vk}(k_U)}^{\beta_1},\overbrace{\mathsf{ek}(k_{PE})}^{\beta_2},\overbrace{\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{ek}(k_{PE})\big),k_U\big)}^{\beta_3},\overbrace{\mathsf{enc}\big((q,p),\mathsf{ek}(k_{PE})\big)}^{\beta_4}\big)$$

[22] We need to ensure that no secret messages are leaked by the transformation, at least in case all participants are honest.

where $S_1 = \mathsf{witness}\ \alpha_1, \alpha_2\ \mathsf{public}\ \beta_1, .., \beta_4\ \mathsf{in}\ \beta_4 = \mathsf{enc}((\alpha_1, \alpha_2), \beta_2) \wedge \mathsf{check}(\beta_3, \beta_1) \rightsquigarrow \beta_4$, is given type:

$$\mathsf{ZKProof}_{S_1} \left( \begin{array}{lll} \beta_1{:}\mathsf{VerKey}(\mathsf{PubEnc}(T_1^*)), & \beta_2{:}\mathsf{EncKey}(T_1^*), & \\ \beta_3{:}\mathsf{Signed}(\mathsf{PubEnc}(T_1^*)), & \beta_4{:}\mathsf{PubEnc}(T_1^*); & \exists \alpha_1, \alpha_2.C_1 \end{array} \right)$$

where $C_1 = (\beta_4 = \mathsf{enc}((\alpha_1, \alpha_2), \beta_2) \wedge \mathsf{check}(\beta_3, \beta_1) \rightsquigarrow \beta_4 \wedge \mathsf{Request}(u, \alpha_2))$. The first two conjuncts in $C_1$ directly correspond to the statement $S_1$. It is always safe to include the proved statement in the formula being conveyed by the zero-knowledge type (rule (Sinfer Stmt)), since the verification of the proof succeeds only if the statement is valid.

However, very often conveying the statement alone does not suffice to type-check the examples we have tried, since the statement only talks about terms and does not mention any logical predicate. The predicates are dependent on the particular protocol and policy, and are automatically inferred by the transformation. For instance, in our running example the original message from the user to the proxy was conveying the predicate $\mathsf{Request}(u, q)$, so this predicate is added by the transformation to the formula $C_1$. Our type-checker verifies that these additional predicates are indeed justified by the statement and by the types of the public components checked for equality by the verifier of the proof.

We illustrate this by type-checking the store in the strengthened protocol in case the proxy is compromised. We start with $ZK_1$, the zero-knowledge proof created by the user, intended to be forwarded by the (actually compromised) proxy and then verified by the store. The first two public messages in $ZK_1$, $\mathsf{vk}(k_U)$ and $\mathsf{ek}(k_{PE})$, are checked for equality against the values the store already has. If the verification of $ZK_1$ succeeds, the store knows that $\beta_1$ and $\beta_2$ have indeed type $\mathsf{VerKey}(\mathsf{PubEnc}(T_1^*))$ and $\mathsf{EncKey}(T_1^*)$, respectively. However, since the proof is received from an untrusted source, it could have been generated by the attacker, so the other public arguments, $\beta_3$ and $\beta_4$, and the witnesses $\alpha_1$ and $\alpha_2$ are initially given type $\mathsf{Un}$. Using this initial type information and the fact that the statement's formula $\beta_4 = \mathsf{enc}((\alpha_1, \alpha_2), \beta_2) \wedge \mathsf{check}(\beta_3, \beta_1) \rightsquigarrow \beta_4$ holds, the type-checker tries to infer additional information.

Since $\beta_1$ has type $\mathsf{VerKey}(\mathsf{PubEnc}(T_1^*))$ and $\mathsf{check}(\beta_3, \beta_1) \rightsquigarrow \beta_4$ holds, we use rule (Sinfer Check) to infer that $\beta_4$ has type $\{\!|\, \mathsf{fkind}(E, \mathsf{PubEnc}(T_1^*), \mathsf{tnt}) \,|\!\} \vee \mathsf{PubEnc}(T_1^*)$, i.e., $\beta_4$ has type $\mathsf{PubEnc}(T_1^*)$ under the condition that the type $\mathsf{PubEnc}(T_1^*)$ is not tainted. If this type was tainted then the type $\mathsf{VerKey}(\mathsf{PubEnc}(T_1^*))$ would be equivalent to $\mathsf{Un}$. However, this is not the case since the user is not compromised. So the new type inferred for $\beta_4$ is equivalent to $\bot \vee \mathsf{PubEnc}(T_1^*)$ and therefore to $\mathsf{PubEnc}(T_1^*)$. Since $\beta_4$ also has type $\mathsf{Un}$ from before, the most precise type we can give to it is the intersection type $\mathsf{PubEnc}(T_1^*) \wedge \mathsf{Un}$. Since $\mathsf{PubEnc}(T_1^*)$ is public this happens to be equivalent to just $\mathsf{PubEnc}(T_1^*)$. Since $\beta_4$ has type $\mathsf{PubEnc}(T_1^*)$ and $\beta_4 = \mathsf{enc}((\alpha_1, \alpha_2), \beta_2)$ we can infer by (Sinfer Enc) that $(\alpha_1, \alpha_2)$ has type $\{\!|\, \mathsf{fkind}(E, T_1^*, \mathsf{tnt}) \,|\!\} \vee T_1^*$. Since the user is not compromised $\mathsf{fkind}(E, T_1^*, \mathsf{tnt})$ is false so $(\alpha_1, \alpha_2)$ has type $T_1^*$. This implies that the predicate $\mathsf{Request}(u, \alpha_2)$ holds, and thus justifies the type annotation automatically generated by the transformation.

The proof $ZK_2$ is easier to type-check since its type just contains the formula of $S_2$, but no additional predicates. This means that its successful verification only conveys certain relations between terms. These relations are, however, critical for linking the different messages. Most importantly, they ensure that the query received in $ZK_2$ is the same as variable $\alpha_2$ in $ZK_1$ for which $\mathsf{Request}(u, \alpha_2)$ holds by the verification of $ZK_1$, as explained above. Since the proxy is compromised the predicate $\mathsf{Registered}(u)$ holds. So in the strengthened protocol the authentication decision of the store is indeed justified by the authorization policy, even if the proxy is compromised.

## 2.7. Case Study: Direct Anonymous Attestation (DAA)

To exemplify the applicability of our type system to real-world protocols, we have modeled and analyzed the authenticity properties of the Direct Anonymous Attestation protocol (DAA) [BCC04]. DAA is a cryptographic protocol that enables the remote authentication of a hardware module called the Trusted Platform Module (TPM), while preserving the anonymity of the user owning the module. Such TPMs are included in many personal computers and servers. More precisely, 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 relies heavily on zero-knowledge proofs to achieve this kind of anonymous authentication.

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 from an entity called the issuer. The DAA-signing protocol enables a TPM to authenticate a message and to prove the verifier to own a valid certificate without revealing the TPM's identity. The protocol ensures that even the issuer cannot link the TPM to its subsequently produced DAA-signatures.

Every TPM has a unique identifier $id$ as well as a key-pair $(k_{id}, \mathsf{ek}(k_{id}))$ called *endorsement key* (EK). The issuer is assumed to know the public component $\mathsf{ek}(k_{id})$ of each EK. The protocol further assumes the existence a publicly known string $bsn_I$ called the basename of the issuer. Every TPM has a secret seed $daaseed$ that allows it to derive secret values $f := \mathsf{shash}(\langle \mathsf{shash}(\langle daaseed, \mathsf{hash}(lpk_I)\rangle), cnt, n_0\rangle)$, where $lpk_I$ is the long-term public key of the issuer, $cnt$ is a counter, and $n_0$ is the integer 0. Each such f-value represents a virtual identity with respect to which the TPM can execute the join protocol and the DAA-signing protocol. Secret hashes $\mathsf{shash}(M)$ are given type $\mathsf{SHash}(T)$, which is neither public nor tainted, unless $T$ is compromised (see Appendix A).

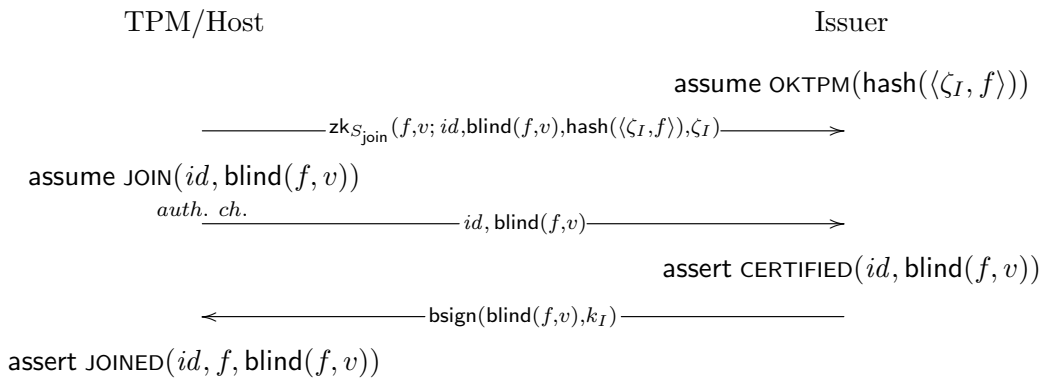
In order to prevent the issuer from learning f-values, DAA relies on blind signatures [Cha83]. The idea is that the TPM sends the disguised (i.e., blinded) f-value $\mathsf{blind}(f, r)$, where $r$ is a random blinding factor, to the issuer, which then produces the blind signature $\mathsf{bsign}(\mathsf{blind}(f, r), k_I)$. The TPM can later unblind the signature obtaining a signature

$\mathsf{usign}(f, k_I)$ of the f-value, which can be publicly verified. The unblinding of blind signatures is done by the unblind destructor, while the verification of the unblinded signature is done by the bcheck destructor. The type $\mathsf{Blinder}(T)$ describes a blinding factor for messages of type $T$, $\mathsf{Blinded}(T)$ describes blinded messages of type $T$, $\mathsf{BSigKey}(T, z.\, C)$ and $\mathsf{BVerKey}(T, z.\, C)$ describe blind signing and verification keys for messages of type $\{z : \mathsf{Blinded}(T) \mid C\}$, and $\mathsf{USigned}(T)$ describes unblinded signatures of messages of type $T$. Our formalization and proofs include blind signatures and secret hashes. For more details we refer the interested reader to Appendix A.

Table 2.1 reports the process for the DAA system. For the sake of readability we use if $D \Downarrow \langle x_1, \ldots, x_n \rangle$ then $P$ else $Q$ to denote the process if $D \Downarrow z$ then let $\langle x_1, \ldots, x_n \rangle = z$ in $P$ else $Q$, where $z$ is a fresh variable.

### 2.7.1. The Join Protocol

In the join protocol, the TPM requests a certificate for one of its f-values $f$ from the issuer. The join protocol has the following overall shape:




The TPM sends to the issuer the blinded f-value $\mathsf{blind}(f, v)$, for some random blinding factor $v$. The TPM is also required to send the hash value $\mathsf{hash}\big(\langle \zeta_I, f \rangle\big)$ along with its request, where $\zeta_I$ is a value derived from the issuer's basename $bsn_I$. This message is used in a rogue-tagging procedure allowing the issuer to recognize corrupted TPMs. All these messages are transmitted together with a zero-knowledge proof, which guarantees that the f-value $f$ is hashed together with $\zeta_I$ in $\mathsf{hash}\big(\langle \zeta_I, f \rangle\big)$. The statement of this zero-knowledge proof is modeled as follows:

$$S_{\mathsf{join}} := \mathsf{witness}\ x_f, x_v\ \mathsf{public}\ y_{id}, y_U, y_N, y_\zeta\ \mathsf{in}\ y_U = \mathsf{blind}(x_f, x_v) \wedge y_N = \mathsf{hash}\big(\langle y_\zeta, x_f \rangle\big)$$

The DAA protocol assumes an authentic channel between the TPM and the issuer in order to authenticate the blinded f-value, and the authors suggest a challenge-response handshake based on the TPM endorsement key as a possible implementation [BCC04]. For the sake of simplicity, we abstract away from the actual cryptographic implementation of such an authentic channel, and we let the TPM send its own identifier together

$$
\begin{array}{lll}
T_f & := & \mathsf{SHash}(\langle \mathsf{SHash}(\langle \mathsf{Private}, \mathsf{Un}\rangle), \mathsf{Un}, \mathsf{Un}\rangle) \\
C_{k_I}(y) & := & \exists id.\mathsf{CERTIFIED}(id, y) \\
T_{k_I} & := & \langle y_U : \mathsf{Blinded}(T_f)\rangle\{C_{k_I}(y_U)\} \\
S_{\mathsf{join}} & := & \mathsf{witness}\ x_f, x_v\ \mathsf{public}\ y_{id}, y_U, y_N, y_\zeta\ \mathsf{in}\ y_U = \mathsf{blind}(x_f, x_v) \wedge y_N = \mathsf{hash}\big(\langle y_\zeta, x_f\rangle\big) \\
S_{\mathsf{sign}} & := & \mathsf{witness}\ x_f, x_{cert}\ \mathsf{public}\ y_{vk}, y_N, y_\zeta, y_m\ \mathsf{in}\ \mathsf{bcheck}(x_{cert}, y_{vk}) \rightsquigarrow x_f \wedge y_N = \mathsf{hash}\big(\langle y_\zeta, x_f\rangle\big) \\
T_{S_{\mathsf{join}}} & := & \mathsf{ZKProof}_{S_{\mathsf{join}}} \left( \begin{array}{l} y_{id} : \mathsf{Un}, y_U : \mathsf{Blinded}(T_f), y_N : \mathsf{Un}, y_\zeta : \mathsf{Un}; \\ \exists x_f, x_v.\, y_U = \mathsf{blind}(x_f, x_v) \ \wedge\ y_N = \mathsf{hash}\big(\langle y_\zeta, x_f\rangle\big) \end{array} \right) \\
T_{S_{\mathsf{sign}}} & := & \mathsf{ZKProof}_{S_{\mathsf{sign}}} \left( \begin{array}{l} y_{vk} : \mathsf{BVerKey}(T_f, y_U.\, C_{k_I}(y_U)), y_N : \mathsf{Un}, y_\zeta : \mathsf{Un}, y_m : \mathsf{Un}; \\ \exists x_f, x_{cert}.\, \exists x_v, x_{id}.\, y_N = \mathsf{hash}\big(\langle y_\zeta, x_f\rangle\big) \\ \quad \wedge\ \mathsf{CERTIFIED}(x_{id}, \mathsf{blind}(x_f, x_v)) \wedge\ \mathsf{SIGNED}(x_f, y_m) \end{array} \right) \\
\mathsf{P}_{\mathsf{join}} & := & \begin{array}{r} \forall id, f, v_1, v_2.(\mathsf{JOIN}(id, \mathsf{blind}(f, v_1)) \wedge \mathsf{OKTPM}(\mathsf{hash}\big(\langle v_2, f\rangle\big)) \\ \Rightarrow \mathsf{CERTIFIED}(id, \mathsf{blind}(f, v_1))) \\ \wedge\ (\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v_1)) \Rightarrow \mathsf{JOINED}(id, f, \mathsf{blind}(f, v_1))) \end{array} \\
\mathsf{P}_{\mathsf{sign}} & := & \forall f, v, m.(\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v)) \wedge \mathsf{SIGNED}(f, m)) \Rightarrow \mathsf{AUTHENTICATED}(m) \\
\mathsf{daa} & := & \begin{array}{l} \mathsf{new}\ k_I : \mathsf{BSigKey}(T_f, y_U.\, C_{k_I}(y_U)).\ \mathsf{new}\ daaseed : \mathsf{Private}.\ \mathsf{new}\ k_{id} : \mathsf{DecKey}(\mathsf{Un}). \\ \mathsf{let}\ f = \mathsf{shash}(\langle \mathsf{shash}(\langle daaseed, \mathsf{hash}(lpk_I)\rangle), cnt, n_0\rangle)\ \mathsf{in} \\ \mathsf{let}\ \zeta_I = \mathsf{hash}\big(\langle n_1, bsn_I\rangle\big)\ \mathsf{in} \\ \mathsf{let}\ N_I = \mathsf{hash}(\langle \zeta_I, f\rangle)\ \mathsf{in} \\ \mathsf{new}\ authch : \mathsf{Ch}(\langle y_{id} : \mathsf{Un}, y_U : \mathsf{Blinded}(T_f)\rangle\{\mathsf{JOIN}(y_{id}, y_U)\}). \\ (\mathsf{tpm} \mid \mathsf{issuer} \mid \mathsf{verifier} \mid \mathsf{assume}\ \mathsf{P}_{\mathsf{join}} \mid \mathsf{assume}\ \mathsf{P}_{\mathsf{sign}}) \end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{tpm} := \\
\quad \mathsf{new}\ v : \mathsf{Blinder}(T_f). \\
\quad \mathsf{let}\ U = \mathsf{blind}(f, v)\ \mathsf{in} \\
\quad (\mathsf{assume}\ \mathsf{JOIN}(id, U) \mid \\
\quad \mathsf{let}\ zkjoin = \mathsf{zk}_{S_{\mathsf{join}}}(f, v;\ id, U, N_I, \zeta_I)\ \mathsf{in} \\
\quad \mathsf{out}(pub, zkjoin).\ \mathsf{out}(authch, \langle id, U\rangle). \\
\quad \mathsf{in}(pub, x). \\
\quad \mathsf{if}\ \mathsf{unblind}(x, v, \mathsf{bvk}(k_I)) \Downarrow x_{cert}\ \mathsf{then} \\
\quad \mathsf{if}\ \mathsf{bcheck}(x_{cert}, \mathsf{bvk}(k_I)) \Downarrow \langle x_f\rangle\ \mathsf{then} \\
\quad \mathsf{if}\ x_f = f\ \mathsf{then} \\
\quad (\mathsf{assert}\ \mathsf{JOINED}(id, f, U) \mid \\
\quad \mathsf{new}\ m : \mathsf{Un}.\ \mathsf{new}\ \zeta : \mathsf{Un}. \\
\quad \mathsf{let}\ N = \mathsf{hash}\big(\langle \zeta, f\rangle\big)\ \mathsf{in} \\
\quad (\mathsf{assume}\ \mathsf{SIGNED}(f, m) \mid \\
\quad \mathsf{let}\ zksign = \mathsf{zk}_{S_{\mathsf{sign}}}(f, x_{cert};\ \mathsf{bvk}(k_I), N, \zeta, m)\ \mathsf{in} \\
\quad \mathsf{out}(pub, zksign)))) \\
\\
\mathsf{issuer} := \\
\quad \mathsf{assume}\ \mathsf{OKTPM}(N_I) \mid \\
\quad !\mathsf{in}(pub, zkjoin). \\
\quad \mathsf{in}(authch, z). \\
\quad \mathsf{let}\ \langle y_{id}, y_U\rangle = z\ \mathsf{in} \\
\quad \mathsf{ver}_{S_{\mathsf{join}}}(zkjoin, y_{id}, y_U, N_I, \zeta_I) \Downarrow \langle\rangle\ \mathsf{then} \\
\quad (\mathsf{assert}\ \mathsf{CERTIFIED}(y_{id}, y_U) \mid \\
\quad \mathsf{out}(pub, \mathsf{bsign}(\langle y_U\rangle, k_I))) \\
\\
\mathsf{verifier} := \\
\quad !\mathsf{in}(pub, zksign). \\
\quad \mathsf{ver}_{S_{\mathsf{sign}}}(zksign, \mathsf{bvk}(k_I)) \Downarrow \langle x_N, x_\zeta, x_m\rangle\ \mathsf{then} \\
\quad \mathsf{assert}\ \mathsf{AUTHENTICATED}(x_m)
\end{array}
$$

Table 2.1.: Our model of DAA

with the blinded f-value over a private channel shared with the issuer. Note that the blinded f-value is still known to the attacker, since it occurs in the public component of the zero-knowledge proof, which is sent over an untrusted channel. Finally, if the zero-knowledge proof received from the TPM passes verification, the issuer sends to the TPM the blind signature $\mathsf{bsign}(\mathsf{blind}(f, v), k_I)$.

#### Type-checking the Join Protocol

The type specified by the user for $S_{\mathsf{join}}$ is

$$T_{S_{\mathsf{join}}} := \mathsf{ZKProof}_{S_{\mathsf{join}}} \left( \begin{array}{l} y_{id} : \mathsf{Un}, y_U : \mathsf{Blinded}(T_f), y_N : \mathsf{Un}, y_\zeta : \mathsf{Un}; \\ \exists x_f, x_v.\, y_U = \mathsf{blind}(x_f, x_v) \ \wedge \ y_N = \mathsf{hash}\big(\langle y_\zeta, x_f \rangle\big) \end{array} \right)$$

and the type of the f-value is $T_f := \mathsf{SHash}(\langle \mathsf{SHash}(\langle \mathsf{Private}, \mathsf{Un} \rangle), \mathsf{Un}, \mathsf{Un} \rangle)$. The formula in $T_{S_{\mathsf{join}}}$ simply gives a logical characterization of the structure of the messages sent by the TPM to the issuer, which is directly guaranteed by the statement $S_{\mathsf{join}}$ of the zero-knowledge proof. The zero-knowledge verification done by the issuer is typed using rules (Proc Ver) and (Sinfer Stmt) and the logical formula $\exists x_f, x_v.\, y_U = \mathsf{blind}(x_f, x_v) \wedge y_N = \mathsf{hash}\big(\langle y_\zeta, x_f \rangle\big)$ is inserted into the typing environment of the issuer. The type of the authentic channel is $\mathsf{Ch}(\langle y_{id} : \mathsf{Un}, y_U : \mathsf{Blinded}(T_f) \rangle \{\mathsf{JOIN}(y_{id}, y_U)\})$. The type system guarantees that the TPM assumes $\mathsf{JOIN}(id, U)$ before sending $id$ and the blinded f-value $U$ on such a channel. Finally, the type of the issuer's signing key is $\mathsf{BSigKey}(T_f, y_U.\, C_{k_I}(y_U))$. The type system guarantees that whenever the issuer produces a certificate for message $M$, $M$ is a blinded secret and there exists $id$ such that $\mathsf{CERTIFIED}(id, M)$ is entailed by the formulas in the typing environment. The authorization policy for the join protocol is as follows:

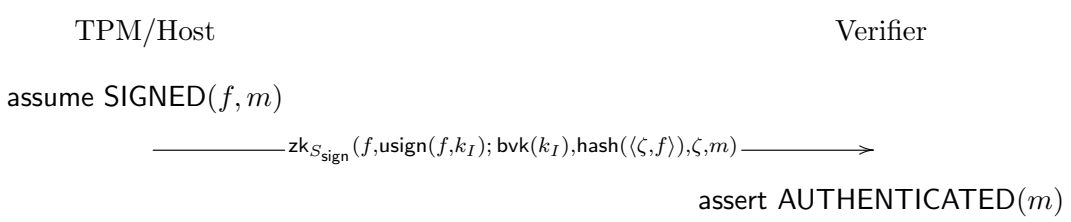
$$\begin{array}{r} \forall id, f, v_1, v_2.(\mathsf{JOIN}(id, \mathsf{blind}(f, v_1)) \wedge \mathsf{OKTPM}(\mathsf{hash}\big(\langle v_2, f \rangle\big)) \\ \Rightarrow \mathsf{CERTIFIED}(id, \mathsf{blind}(f, v_1))) \\ \wedge\ (\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v_1)) \Rightarrow \mathsf{JOINED}(id, f, \mathsf{blind}(f, v_1))) \end{array}$$

This policy allows the issuer to produce a blind signature for TPM $id$ (assertion $\mathsf{CERTIFIED}(id, \mathsf{blind}(f, v_1))$) only if the TPM $id$ has started the join protocol to authenticate $\mathsf{blind}(f, v_1)$ (assumption $\mathsf{JOIN}(id, \mathsf{blind}(f, v_1))$) and the f-value $f$ is associated to a valid TPM (assumption $\mathsf{OKTPM}(\mathsf{hash}\big(\langle v_2, f \rangle\big))$). Additionally, the policy guarantees that whenever a TPM $id$ successfully completes the join protocol (assertion $\mathsf{JOINED}(id, f, \mathsf{blind}(f, v_1))$), the issuer has certified $\mathsf{blind}(f, v_1)$ (assertion $\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v_1))$).

### 2.7.2. The DAA-signing Protocol

After successfully executing the join protocol, the TPM has a valid certificate for its f-value $f$ signed by the issuer. Since only valid TPMs should be able to DAA-sign a

message $m$, the TPM has to convince a verifier that it possesses a valid certificate. Of course, the TPM cannot directly send it to the verifier, since this would reveal $f$. Instead, the TPM produces *zksign*, a zero-knowledge proof that it knows a valid certificate. If the TPM would, however, just send $(zksign, m)$ to the verifier, the protocol would be subject to a trivial message substitution attack. Message $m$ is instead combined with the proof so that one can only replace $m$ by redoing the proof (and this again can only be done by knowing a valid certificate). The overall shape of the DAA-signing protocol is hence as follows:




with $S_{\mathsf{sign}} := \mathsf{witness}\ x_f, x_{cert}\ \mathsf{public}\ y_{vk}, y_N, y_\zeta, y_m\ \mathsf{in}\ \mathsf{bcheck}(x_{cert}, y_{vk}) \rightsquigarrow x_f \wedge y_N = \mathsf{hash}\big(\langle y_\zeta, x_f\rangle\big)\big)$. The zero-knowledge proof guarantees that the secret f-value $f$ is signed by the issuer and that such a value is hashed together with a fresh value $\zeta$[23]. This hash is used in the rogue tagging procedure mentioned above.

### Type-checking the DAA-signing Protocol

The type specified by the user for $S_{\mathsf{sign}}$ is

$$T_{S_{\mathsf{sign}}} := \mathsf{ZKProof}_{S_{\mathsf{sign}}} \left( \begin{array}{l} y_{vk} : \mathsf{BVerKey}(T_f, y_U.\, C_{k_I}(y_U)), y_N : \mathsf{Un}, y_\zeta : \mathsf{Un}, y_m : \mathsf{Un}; \\ \exists x_f, x_{cert}.\, \exists x_v, x_{id}.\, y_N = \mathsf{hash}\big(\langle y_\zeta, x_f\rangle\big) \\ \quad \wedge\ \mathsf{CERTIFIED}(x_{id}, \mathsf{blind}(x_f, x_v)) \wedge\ \mathsf{SIGNED}(x_f, y_m) \end{array} \right)$$

This type guarantees that the f-value of the TPM has been certified by the issuer (assertion $\mathsf{CERTIFIED}(x_{id}, \mathsf{blind}(x_f, x_v))$, captures the constraint on the hash inherited from the statement of the zero-knowledge proof ($y_N = \mathsf{hash}\big(\langle y_\zeta, x_f\rangle\big)$), and states that the user has signed message $m$ (assumption $\mathsf{SIGNED}(x_f, y_m)$). On the verifier's side, the assertion $\mathsf{CERTIFIED}(x_{id}, \mathsf{blind}(x_f, x_v))$ is guaranteed to hold by the verification of the certificate proved by zero-knowledge and by the type of the verification key, while the equality $y_N = \mathsf{hash}\big(\langle y_\zeta, x_f\rangle\big)$ is enforced by the semantics of zero-knowledge verification. Furthermore the type of the verification key guarantees that the f-value used to create the proof is of type $T_f$. Since values of this type cannot be given type $\mathsf{Un}$ (type $T_f$ is disjoint from $\mathsf{Un}$) the proof is generated by a honest TPM, and thus we can apply rule (Proc Ver), and the logical formula is inserted into the typing environment of the continuation process. The authorization policy for the DAA-signing protocol is:

$$\forall f, v, m.(\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v)) \wedge \mathsf{SIGNED}(f, m)) \Rightarrow \mathsf{AUTHENTICATED}(m)$$

[23] In the pseudonymous variant of the DAA-signing protocol $\zeta$ is derived in a deterministic fashion from the basename $bsn_V$ of the verifier. Our analysis can be easily adapted to this variant.

This policy allows the verifier to authenticate message $m$ (assertion $\mathsf{AUTHENTICATED}(m)$) only if the sender proves the knowledge of some certified f-value $f$ associated to some TPM id (assertion $\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v))$) and the zero-knowledge proof includes message $m$ (assumption $\mathsf{SIGNED}(f, m)$). Note that the $id$ of the TPM is existentially quantified, since it is not known to the verifier.

Our type-checker can prove in less than three seconds that the daa process is well-typed. By Theorem 2.35 (Robust Safety), this guarantees that the daa process is robustly safe.

## 2.8. Implementation

We have implemented a type-checker for the type system presented in this chapter. The type-checking phase generates proof obligations that are discharged independently, leading to a scalable and robust analysis. The type-checker relies on first-order logic automated theorem provers [WDF+09, Sch02, RV99] or SMT solvers [dMB08] to discharge proof obligations. We rely on standard input formats for theorem provers (the TPTP format [Sut09] and the old DFG format [WDF+09]) and SMT solvers (STM-LIB 1.2 [RT06]). When the prover fails to discharge a proof obligation, we can easily track that back to a location in the code.

Our implementation of kinding uses the logical characterization from §2.4.5, instead of the (equivalent) syntax directed rules. We check subtyping in a similar way, by relying on an encoding of the subtyping judgement in the authorization logic. This technique has the advantage of efficiently implementing the subtyping judgement, which is very far from being syntax directed. We thus rely on the FOL prover for handling disjunctions, instead of naively backtracking when multiple rules apply. For typing terms we rely on semi-unification for inferring instantiations of the type variables, since most constructors and destructors are let-polymorphic, and asking the user to annotate every constructor and destructor application would have been unacceptable from a usability perspective.[24] Typing processes works similar to a bidirectional type system [PT00], using type inference for synthesizing the type of many of the involved terms, and checking that a term has a certain type when such a type is given (e.g., by an explicit type annotation).

Our implementation is extensible: adding new constructors, destructors, and types can be done easily by changing a configuration file. One still needs to extend Lemma 2.29 (Destructor Consistency) to handle the newly added destructors, but for most primitives that is all that is needed for extending the type system. The implementation[25] is written in OCaml, comprises more than 5kLOC, and is open source, distributed under the Apache License version 2.0.

[24] While the semi-unification problem is undecidable in general [KTU90], broad special cases are decidable [Got11] and simple heuristics work well in practice [SC08].

[25] Type-checker is available at `http://www.infsec.cs.uni-saarland.de/projects/zk-typechecker/`

We have tested our type-checker on a model of the DAA protocol [BCC04] (see §2.7), on protocols automatically strengthened against compromised participants (see §2.6), and on several simpler examples. The analysis of DAA terminated in less than three seconds, while for the simpler examples the analysis time was less than half a second, on a normal laptop. These promising results show that our static analysis technique has the potential to scale up to very large protocols.

## 2.9. Summary

In this chapter, we have introduced the first type system for statically analyzing the security of protocols based on non-interactive zero-knowledge proofs. The type system combines prior work on refinement types for cryptographic protocols, with union types, intersection types, and the novel ability to reason statically about the disjointness of types. We believe that the improvements brought by this type system over the existing type-based analyses for security protocols are useful in general, beyond analyzing protocols based on zero-knowledge proofs. The next chapter will show that our techniques are useful for analyzing concrete implementations of cryptographic protocols.

# Chapter 3

# Analyzing Protocol Implementations

This chapter presents a new type system for verifying the source code of cryptographic protocol implementations. The underlying type theory combines refinement types [BBF+08, BBF+11] with union, intersection, and polymorphic types. Additionally, we use a relation for statically reasoning about the disjointness of types that is very similar to the one we introduced in §2.4.6. This expressive type system extends the scope of existing type-based analyses of protocol implementations [BBF+08, BFG10] to important protocol classes that were not covered so far. In particular, our types statically characterize: (*i*) more usages of asymmetric cryptography, such as signatures of private data and encryptions of authenticated data; (*ii*) authenticity and integrity properties achieved by showing knowledge of secret data; (*iii*) applications based on non-interactive zero-knowledge proofs.

The cryptographic protocols are implemented in $\text{RCF}^{\forall}_{\wedge\vee}$ [BBF+08, BBF+11], a concurrent lambda-calculus, and the cryptographic operations are considered fully reliable building blocks and represented symbolically using a dynamic sealing mechanism [Mor73, SP07, BBF+08], which is based on standard functional programming language constructs. 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 $\text{RCF}^{\forall}_{\wedge\vee}$, the type system, and all the important parts of the soundness proof in the Coq proof assistant. We achieve this by defining a core calculus, which we call Formal-$\text{RCF}^{\forall}_{\wedge\vee}$, and which is obtained from $\text{RCF}^{\forall}_{\wedge\vee}$ by type erasure and by

adopting a locally nameless representation for binders [ACP+08]. This work allowed us to discover three relatively small problems in the soundness proofs of prior type systems with refinement types [BBF+08, BHM08c] and to propose and evaluate fixes for the faulty proofs.

Our type-based analysis is automated, modular, efficient, and provides security proofs for an unbounded number of sessions. We have implemented a type-checker that performed very well in our experiments: it type-checks all our symbolic libraries and sample code totaling more than 1500LOC in around 12 seconds, on a normal laptop. The type-checker features a user-friendly graphical interface for examining typing derivations. The tool-chain we have developed additionally contains the automatic code generator for zero-knowledge proofs, an interpreter, and a visual debugger. The formalization and the implementation are available online[1].

**Outline** The remainder of this chapter is structured as follows: §3.1 discusses related work. §3.2 gives an intuitive overview of our type system and exemplifies the most important concepts on a simple authentication protocol. §3.3 introduces the syntax of $\text{RCF}^{\forall}_{\wedge\vee}$, the language supported by our type-checker. §3.4 presents our type system. §3.5 describes the results of our Coq formalization. In §3.6 we use union and intersection types to give stronger and more natural types to the dynamic sealing-based encoding of asymmetric cryptography. §3.7 presents our dynamic sealing-based encoding of zero-knowledge proofs. §3.8 describes our implementation and experiments. §3.9 discusses related work on union and intersection types. §3.10 provides a summary of this chapter.

Furthermore, Appendix B lists the Formal-$\text{RCF}^{\forall}_{\wedge\vee}$ calculus, the erasure function from $\text{RCF}^{\forall}_{\wedge\vee}$, the operational semantics and the type system of Formal-$\text{RCF}^{\forall}_{\wedge\vee}$; Appendix C provides more details about our encoding of zero-knowledge proofs.

## 3.1. Related Work

Our type system extends the refinement type system by Bengtson et al. [BBF+08] 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 [AB03], but later disappeared in the more advanced type systems for authorization policies. Our extension allows the type

[1] http://www.infsec.cs.uni-saarland.de/projects/F5/

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. [BBF+08] 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 [BCC04], which involves digital signatures on secret TPM identifiers.

In independent parallel work, Bhargavan et al. [BFG10] have developed an additional cryptographic library for a simplified version of the type system proposed by Bengtson et al. [BBF+08]. 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 [BFG10] to new primitives requires expertise and a considerable human effort. In contrast, extending our sealing-based library does not involve any additional proof: one has just to find a well-typed encoding of the desired cryptographic primitive, which is relatively easy. For instance, Eigner [Eig09] reports encoding the sophisticated cryptographic schemes used in the Civitas [CCM08] electronic voting protocol using dynamic seals, in a relatively short amount of time.

The main simplification Bhargavan et al. [BFG10] propose over the type system by Bengtson et al. [BBF+08] 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. [BFG10] is thus weaker than the widely-accepted robust safety property [GJ03, GJ04, FGM07a, BBF+08]; 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 [BBF+08] we also retain the property that *all* attackers are well-typed with respect to our type system (this property is usually called opponent typability [GJ03]), which allows us to prove that our type system enforces robust safety. Despite the two downsides discussed above, Bhargavan et al. [BFG10] manage to solve some of the problems in the original work of Bengtson et al. [BBF+08] 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. [BMU10] 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 [BHU09]; 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. [FKS11], 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 [Eig09] uses our type system to verify eligibility, inalterability, and individual verifiability for a simple implementation of the Civitas electronic voting protocol [CCM08]. 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 [MP11] 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 [GLP05] 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 [CD08] 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. [DGJN11] 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. [BFG10]. 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. proposed a technique [BFGT08] for the verification of F# protocol implementations by automatically extracting ProVerif models [Bla01], using an extension of the functions as processes encoding [Mil92]. The technique was successfully used to verify implementations of real-world cryptographic protocols such as TLS [BCFZ08] and Europay-MasterCard-Visa (EMV) [dRP11]. The underlying analysis using ProVerif is, however, not compositional and is significantly less scalable than type-checking [BFG10]. 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. [AGJ11] and Corin and Manzano [CM11] have proposed techniques for analyzing C programs by extracting abstract models using symbolic execution. The solution by Aizatulin et al. [AGJ11] needs neither a pre-existing protocol description nor manual inspection of source code, and uses existing results for the applied pi calculus [BHU09] 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 [CM11] seems to be able to handle branches, but it cannot yet handle security properties.

Swamy et al. [SCC10] 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 (stateful properties can still be specified and verified within RCF using a refined state monad [BGP11]) and information flow. As

opposed to RCF, however, FINE is not concurrent and cannot easily express Dolev-Yao attackers and cryptographic operations. In very recent work, Swamy et al. [SCF+11] seem to partially address the latter limitation using some of the ideas of Bhargavan et al. [BFG10].

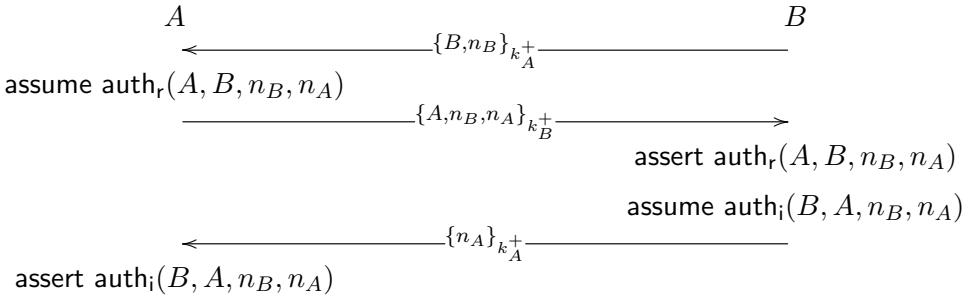
The more technical discussion about the related work on union and intersection types is postponed to §3.9.

## 3.2. Our Type System at Work

Before giving the details of the calculus and the type system, we illustrate the main concepts of our static analysis technique on the Needham-Schroeder-Lowe public-key protocol [Low96] (NSL), which could not be analyzed by the original refinement type system by Bengtson et al. [BBF+08]. For convenience, throughout this section we use some syntactic sugar that is supported by our type-checker and can be obtained from the core calculus presented in §3.3 by standard encodings [BBF+08].

### 3.2.1. Protocol Description and Security Annotations

The Needham-Schroeder-Lowe protocol is depicted below:




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

ciphertext and checking the nonce, $A$ knows that the ciphertext comes from $B$ as the nonce $n_A$ is *private* and only $B$ has access to it.

Following [FGM07a, BBF+08], 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.

### 3.2.2. Types for Cryptography

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

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

Like many of the functions in our cryptographic library, the *encrypt* and *decrypt* functions are polymorphic. Their code is type-checked only once and given an universal type. The type variable $\alpha$ 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 $\mathsf{Un}$ 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 $\alpha$ or by the opponent to encrypt data of type $\mathsf{Un}$. 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 $\mathsf{Un}$. The *decrypt* function takes as input a ciphertext of type $\mathsf{Un}$, 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 $\mathsf{Un}$ or a supertype thereof [BBF+08], which severely limits the expressiveness of the type system and prevents the analysis of a number of protocols, including this very simple example.

### 3.2.3. Type-checking the NSL Protocol

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

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

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

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

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

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

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

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

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

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

Table 3.1.: NSL Initiator Code and Responder Code

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 $\mathsf{assert}\ \mathsf{auth_i}(x_B, x_A, x_{nB}, n_A)$ 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 $x_{nB}$.

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

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

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

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

**Attacker's Interface for NSL**

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

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

For simplicity of presentation, we do not give the attacker the capability to compromise principals, so the famous attack discovered by Lowe [Low96] is not possible even if we were to drop A's identity from the second message.

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

**Test Setup for NSL**

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

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

## 3.3. The RCF$^{\forall}_{\wedge\vee}$ Calculus

The Refined Concurrent FPC (RCF) [BBF+08] is a simple programming language extending the Fixed Point Calculus [Gun92] with refinement types [FP91, ROS98, XP99] and concurrency [Mil99, AG99]. This core calculus is expressive enough to encode a considerable fragment of an ML-like programming language [BBF+08]. In this thesis, we further increase the expressivity of the calculus by adding intersection types [Pie97], union types [Pie91], parametric polymorphism [Rey83, Gir86], 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 §3.5).

Given a phrase of syntax $\phi$, let $\phi\{M/x\}$ denote the substitution of each free occurrence of the variable $x$ in $\phi$ 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.

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 [BBF+08].

**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 |
| $\quad \mathsf{fold}_{\mu\alpha.\,T}\ M$ | $\quad$ recursive value |
| $\quad \Lambda\alpha.\, A$ | $\quad$ type abstraction* (scope of $\alpha$ 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**

| | |
|---|---|
| $\alpha, \beta$ | type variable |
| $T, U, V ::=$ | type |
| $\mathsf{unit}$ | 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 $\alpha$ is $T$) |
| $\alpha$ | 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 $\alpha$ is $T$) |

The unit value () is given type $\mathsf{unit}$. 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 [FP91, ROS98, XP99, BBF$^+$08] 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 $\forall\alpha.\, T$ [Rey83, Gir86] 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 [BBF+08, Pie91, Rey83, Gir86]. 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 §3.5). During type-checking the variable $x$ is given the intersection of the types of $M$ and $N$. When the variable $x$ is not necessary we omit the as clause, as we did in §3.2. The restriction $(\nu a \updownarrow T)A$ generates a globally fresh channel $a$ that can only be used in $A$ to convey values of type $T$. The expression $A \Rsh B$ evaluates $A$ and $B$ in parallel, and returns the result of $B$ (the result of $A$ is discarded). The expression $a!M$ outputs 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 §3.5 and statically enforced by our type system from §3.4.

**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 $\mathrm{RCF}^{\forall}_{\wedge\vee}$ values are the terms of this logic, and equality $M = N$ is interpreted as syntactic equality between values.

## 3.4. Type System

This section presents our type system for enforcing authorization policies on $\mathrm{RCF}^{\forall}_{\wedge\vee}$ code. This extends the type system proposed by Bengtson et al. [BBF+08] with union [Pie91], intersection [Pie97], and polymorphic types [Rey83, Gir86]. 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 $E$ |

### 3.4.1. Well-formed Environments and Entailment

A typing environment $E$ is a list of bindings for variables ($x : T$), type variables ($\alpha$ 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 |
| $\alpha$ | 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 $free(\mu)$**

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

**Well-formed environments and types**

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

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.

**Entailed formulas $E \vdash C$**

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

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

### 3.4.2. 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 $\mathsf{Un}$, since such channels are considered under the complete control of the adversary. However, a system in which only data of type $\mathsf{Un}$ can be communicated over the untrusted network would be too restrictive, e.g., a value of type $\{x : \mathsf{Un} \mid Ok(x)\}$ could not be sent over the network.

Subtyping is commonly used to compare types with type $\mathsf{Un}$. In particular, we allow values having type $T$ that is a subtype of $\mathsf{Un}$, 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 $\mathsf{Un}$ 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. We list all rules for kinding and subtyping, and then explain the more 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)
$$\frac{}{E \vdash \mathsf{unit} :: k}$$

(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*) $\dfrac{E \vdash T}{E \vdash T <: T}$ (Sub Top*) $\dfrac{E \vdash T}{E \vdash T <: \top}$ (Sub Pub Tnt) $\dfrac{E \vdash T :: \mathsf{pub} \quad E \vdash U :: \mathsf{tnt}}{E \vdash T <: U}$

(Sub Refine Left) $\dfrac{E \vdash \{x : T \mid C\} \quad E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'}$ (Sub Refine Right) $\dfrac{E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}$ (Sub Univ*) $\dfrac{E, \alpha \vdash T <: U}{E \vdash \forall \alpha. T <: \forall \alpha. U}$

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

(Sub And LB 1) $\dfrac{E \vdash T_1 <: U \quad E \vdash T_2}{E \vdash T_1 \wedge T_2 <: U}$ (Sub And LB 2) $\dfrac{E \vdash T_1 \quad E \vdash T_2 <: U}{E \vdash T_1 \wedge T_2 <: U}$ (Sub And Greatest) $\dfrac{E \vdash T' <: T_1 \quad E \vdash T' <: T_2}{E \vdash T' <: T_1 \wedge T_2}$

(Sub Or Least) $\dfrac{E \vdash T_1 <: U \quad E \vdash T_2 <: U}{E \vdash T_1 \vee T_2 <: U}$ (Sub Or UB 1) $\dfrac{E \vdash T <: U_1 \quad E \vdash U_2}{E \vdash T <: U_1 \vee U_2}$ (Sub Or UB 2) $\dfrac{E \vdash U_1 \quad E \vdash T <: U_2}{E \vdash T <: U_1 \vee U_2}$

(Sub Sum) $\dfrac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')}$ (Sub Pos Rec*) $\dfrac{E, \alpha \vdash T <: U \quad \alpha \text{ only occurs positively in } T \text{ and } U}{E \vdash \mu\alpha. T <: \mu\alpha. U}$

**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 $\{x : \mathsf{Un} \mid \mathit{Ok}(x)\}$ can be sent on a channel of type $\mathsf{Un}$. 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., $T \to U$ 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 [LW94]. 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). Or intuitively, 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[3] greatest lower bound of the types $T_1$ and $T_2$. Rules (Sub And LB 1) and (Sub And LB 2) ensure that $T_1 \wedge T_2$ is a lower bound: by using reflexivity in the premise we obtain that $T_1 \wedge T_2 <: T_1$ and $T_1 \wedge T_2 <: T_2$. Rule (Sub And Greatest) ensures that $T_1 \wedge T_2$ is greater than any other lower bound: if $T'$ is another lower bound of $T_1$ and $T_2$ then $T'$ is a subtype of $T_1 \wedge T_2$. As far as kinding is concerned, the type $T_1 \wedge T_2$ is public if $T_1$ is public or $T_2$ is public, and it is tainted if both $T_1$ and $T_2$ are tainted. The union type $T_1 \vee T_2$ 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 [DP00], 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 §3.4.4 and §3.9).

**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 $\alpha$. Note that $\alpha$ can be substituted by any type, so we cannot assume anything about $\alpha$ 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 [Pie91]). Recent work by Dunfield [Dun09] 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. [TCGS89]. It differs significantly from Cardelli's Amber rule [AC93, Car97], which is more well-known and which is used by the original RCF [BBF+08]:

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

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

(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 [BBF+08] – 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[4], this did not pose big problems for us in practice[5], where most of the time only positive recursive types [Men91, Urz95] are used. Moreover, this positivity restriction only affects subyping, 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 [BBF+08])

Ligatti et al. [LNBH11] 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 be any easier than for the Amber rule.

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

In RCF [BBF+08] 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, Un $\to$ Un, Un $*$ Un, Un $+$ Un,

[4] 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.

[5] Val Tannen et al. [TGS89] 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.

$\mu\alpha.\,\mathsf{Un}$, 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 $\mathsf{Un}$, a property we want for our $\mathsf{Private}$ type. Perhaps unintuitively, it is not enough that a type is not public and not tainted to make it disjoint from $\mathsf{Un}$ (e.g., $\top \to \top$ is not public and not tainted, still $\lambda x : \top.\, x$ and $\lambda x : \mathsf{Un}.\, x$ are two syntactically equal values that inhabit $\top \to \top$ and $\mathsf{Un} \to \mathsf{Un}$ respectively). A final observation is that, in $\mathrm{RCF}^{\forall}_{\wedge\vee}$, in an inconsistent environment ($E \vdash \mathsf{false}$) *all* types are equivalent and all values inhabit all types. This means that $\mathsf{Private}$ being disjoint from $\mathsf{Un}$ 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[6]. Intuitively, if the attacker would know a value of this type, then he could call it (values of type $\mathsf{Private}_C$ have to be functions), which would exercise the $\mathsf{assert}\ C$ and invalidate the safety of the system, unless $C$ can be derived from the formula log. Type $\mathsf{Private}_C$ resembles a singleton type, in that it contains only values of a very specific form. We use an existential quantifier over values to ensure that there are infinitely many values of this type. The type $\mathsf{Private}$ is obtained as $\mathsf{Private}_{\mathsf{false}}$.

### 3.4.4. 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. We first list the rules of each judgment, and then we explain the some of the ones that are new with respect to [BBF+08].

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

[6] The type $\mathsf{Private}_C$ also appears naturally when reasoning about security despite compromised participants [BGHM09].

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

$$\text{(Val Fold)}\ \frac{E \vdash M : T\{\mu\alpha.T/\alpha\} \quad E \vdash \mu\alpha.T}{E \vdash \mathsf{fold}_{\mu\alpha.T}\ M : \mu\alpha.T} \qquad \text{(Val Unit)}\ \frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}} \qquad \text{(Val Inl)}\ \frac{E \vdash M : T}{E \vdash \mathsf{inl}\ M : T + U} \qquad \text{(Val Inr)}\ \frac{E \vdash M : U}{E \vdash \mathsf{inr}\ M : T + U}$$

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 [DP00] the value restriction is crucial for the soundness of this introduction rule in the presence of side-effects (also see §3.9). Also, unrelated to the value restriction, this rule is not very useful on its own: since we are in a calculus with typing annotations, it is hard to give one annotated value two different types. For instance, if we want to give the identity function type (Private→Private) ∧ (Un→Un) we need to annotate the argument with type Private (i.e., $\lambda x{:}\mathsf{Private}.\, x$) in order to give it type Private→Private, but then we cannot give this value type Un→Un. Following Pierce [Pie91, Pie97] and Reynolds [Rey96] we use the for construct to explicitly alternate type annotations. For instance, the identity function of type (Private→Private) ∧ (Un→Un) can be written as (for $\alpha$ in Private; Un. $\lambda x{:}\alpha.\, x$). By rule (Val For 1*) we can give this value type Private→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** $E \vdash A : T$

$$\text{(Exp Appl)}\ \frac{E \vdash M : (x : T \to U) \quad E \vdash N : T}{E \vdash M\ N : U\{N/x\}} \qquad \text{(Exp Inst*)}\ \frac{E \vdash M : \forall\alpha.U}{E \vdash M\langle T\rangle : U\{T/\alpha\}} \qquad \text{(Exp Subsum)}\ \frac{E \vdash A : T \quad E \vdash T <: T'}{E \vdash A : T'}$$

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

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

$$\text{(Exp Let)}\ \frac{E \vdash A : T \quad E, x : T \vdash B : U \quad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A\ \mathsf{in}\ B : U} \qquad \text{(Exp Assume)}\ \frac{E \vdash \diamond \quad \mathit{free}(C) \subseteq \mathit{dom}(E)}{E \vdash \mathsf{assume}\ C : \{\_ : \mathsf{unit} \mid C\}}$$

$$\text{(Exp Res)}\ \frac{E, a \updownarrow T \vdash A : U \quad a \notin \mathit{fn}(U)}{E \vdash (\nu a \updownarrow T)A : U} \qquad \text{(Exp Send)}\ \frac{E \vdash M : T \quad (a \updownarrow T) \in E}{E \vdash a!M : \mathsf{unit}} \qquad \text{(Exp Recv)}\ \frac{E \vdash \diamond \quad (a \updownarrow T) \in E}{E \vdash a? : T}$$

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

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

(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**

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

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

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 [Pie91] using the (Exp Case*) rule.[7] 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 §3.2). After decryption, the obtained plaintext is given type $T \vee \mathsf{Un}$ since it might come from a honest party as well as from the attacker. We have thus to type-check the remaining code twice, once under the assumption that $x$ is of type $T$, and once assuming that $x$ is of type $\mathsf{Un}$.

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 §3.6.2) and zero-knowledge (see §3.7). 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 $\mathsf{Un}$ and $\mathsf{Private}$ have common values only if the environment is inconsistent (i.e., $E \vdash \mathsf{false}$). Therefore, when comparing

[7] As pointed out by Dunfield and Pfenning [DP04] eliminating union types for expressions that are not in evaluation contexts is unsound in the presence of non-determinism (this is further discussed in §3.9).

values of disjoint types it is safe to add $\mathsf{false}$ 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 [AB03] for verifying secrecy properties of nonce handshakes, but later disappeared in the more advanced type systems for authorization policies.

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

$$\frac{fv(C) = \emptyset}{\vdash \mathsf{Private}_C \between \mathsf{Un} \rightsquigarrow C} \text{ (ND Private Un)} \qquad \frac{}{\vdash T_1 \between T_2 \rightsquigarrow \mathsf{true}} \text{ (ND True)} \qquad \frac{\vdash T_2 \between T_1 \rightsquigarrow C}{\vdash T_1 \between T_2 \rightsquigarrow C} \text{ (ND Sym)}$$

$$\frac{\vdash T_1 \between T_2 \rightsquigarrow C}{\vdash \{x : T_1 \mid C_1\} \between T_2 \rightsquigarrow C} \text{ (ND Refine)} \qquad \frac{\vdash (T\{\alpha/\mu\alpha.\,T\}) \between (U\{\beta/\mu\beta.\,U\}) \rightsquigarrow C}{\vdash (\mu\alpha.\,T) \between (\mu\beta.\,U) \rightsquigarrow C} \text{ (ND Rec)}$$

$$\frac{\vdash T_1 \between U_1 \rightsquigarrow C_1 \quad \vdash T_2 \between U_2 \rightsquigarrow C_2}{\vdash (T_1 * T_2) \between (U_1 * U_2) \rightsquigarrow C_1 \wedge C_2} \text{ (ND Pair)} \qquad \frac{\vdash T_1 \between U_1 \rightsquigarrow C_1 \quad \vdash T_2 \between U_2 \rightsquigarrow C_2}{\vdash (T_1 + T_2) \between (U_1 + U_2) \rightsquigarrow (C_1 \vee C_2)} \text{ (ND Sum)}$$

$$\frac{\vdash T_1 \between U \rightsquigarrow C_1 \quad \vdash T_2 \between U \rightsquigarrow C_2}{\vdash (T_1 \wedge T_2) \between U \rightsquigarrow C_1 \wedge C_2} \text{ (ND And)} \qquad \frac{\vdash T_1 \between U \rightsquigarrow C_1 \quad \vdash T_2 \between U \rightsquigarrow C_2}{\vdash (T_1 \vee T_2) \between U \rightsquigarrow C_1 \vee C_2} \text{ (ND Or)}$$

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

The other inductive rules lift the $\mathsf{NonDisj}$ 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 $\alpha$. We can show that $\mathsf{tree}\langle\mathsf{Private}\rangle$ and $\mathsf{tree}\langle\mathsf{Un}\rangle$ are provably disjoint. By (ND Rec) we need to show that the unfolded types

$\mathsf{Private} + (\mathsf{Private} * \mathsf{tree}\langle\mathsf{Private}\rangle * \mathsf{tree}\langle\mathsf{Private}\rangle)$ and $\mathsf{Un} + (\mathsf{Un} * \mathsf{tree}\langle\mathsf{Un}\rangle * \mathsf{tree}\langle\mathsf{Un}\rangle)$ are disjoint. By (ND Sum) we need to show both that $\mathsf{Private}$ and $\mathsf{Un}$ are disjoint, which is immediate by (ND Private Un), and that the pair types $(\mathsf{Private} * \mathsf{tree}\langle\mathsf{Private}\rangle * \mathsf{tree}\langle\mathsf{Private}\rangle)$ and $(\mathsf{Un} * \mathsf{tree}\langle\mathsf{Un}\rangle * \mathsf{tree}\langle\mathsf{Un}\rangle)$ are disjoint. For the latter, by (ND Pair) it suffices to show that the types of the first components of the pair are disjoint, which follows again by (ND Private Un).

Finally, we remark that the property we called provable disjointness in this section is a tractable (mostly syntax-directed) approximation for the real disjointness of types. This approximation is formally proven sound in Theorem 3.5 from §3.5.

## 3.5. Results of the Formalization

We have formalized the metatheory of $\mathrm{RCF}^{\forall}_{\wedge\vee}$ in the Coq proof assistant [Coq09]. We achieve this by defining Formal-$\mathrm{RCF}^{\forall}_{\wedge\vee}$, a core calculus where terms are encoded using a *locally nameless representation* [Gor93, ACP+08]: 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 [dB72]. 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-$\mathrm{RCF}^{\forall}_{\wedge\vee}$ and $\mathrm{RCF}^{\forall}_{\wedge\vee}$ is that in Formal-$\mathrm{RCF}^{\forall}_{\wedge\vee}$ *all type annotations from values, expressions and formulas are erased.*

**Type erasure for selected values and expressions**

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

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

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

It uses the (Val Refine) rule to give $M\{T_1/\alpha\}$ a singleton type, and then the (Val For 1*) rule to give the *same* singleton type to the value $(\mathsf{for}\ \alpha\ \mathsf{in}\ T_1; T_2.\, M)$. The only way this can possibly work is because the logic equates $M\{T_1/\alpha\}$ and $(\mathsf{for}\ \alpha\ \mathsf{in}\ T_1; T_2.\, M)$, by working on values where all type annotations and the $\mathsf{for}$ 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 [BBF$^+$08], which is also extrinsically typed. In particular the operational semantics of Formal-RCF$^{\forall}_{\wedge\vee}$[8] corresponds directly to the one of the original RCF, which is defined in terms of a heating relation that allows for syntactic rearrangements of concurrent expressions ($e \Rrightarrow e'$) and a standard reduction relation ($e \to e'$). To prevent confusion, in the following we use $e$ to stand for the expressions, $v$ for the values, and $F$ for the formulas of Formal-RCF$^{\forall}_{\wedge\vee}$.

As advertised by Aydemir et al. [ACP$^+$08], in our core language the inductive rules are defined using *cofinite quantification*. This yields strong induction and inversion principles for the relations of the system, and obviates the need for reasoning about alpha-equivalence.

**Two of the rules using cofinite quantification**

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

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

We have proved that the typing judgments of RCF$^{\forall}_{\wedge\vee}$ are preserved by type erasure. This proof relies on standard [ACP$^+$08] renaming lemmas for the Formal-RCF$^{\forall}_{\wedge\vee}$ judgments (we use $\Vdash$ to denote Formal-RCF$^{\forall}_{\wedge\vee}$ judgments).

**Lemma 3.1** (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 3.2** (Adequacy of RCF$^{\forall}_{\wedge\vee}$ Type System)**.**
*For all typing judgments* $\mathcal{J}$*, if* $E \vdash \mathcal{J}$ *then* $[\![E]\!] \Vdash [\![\mathcal{J}]\!]$.

The main result we have proved for the type system is that well-typed expressions are robustly safe. As in [BBF$^+$08], the property follows from the subject-reduction property of the type system. We also list a couple of important lemmas and theorems used in the proof. The high-level structure of our proofs is similar to the one of our proofs for the spi-calculus (§2.5).

**Lemma 3.3** (Inconsistent Environment)**.** *If* $E \Vdash \mathsf{false}$*,* $E \Vdash T$ *and* $free(e) \subseteq dom(E)$ *then* $E \Vdash e : T$.

[8] 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).

**Lemma 3.4** (Transitivity of Subtyping)**.** *If* $E \Vdash T_1 <: T_2$ *and* $E \Vdash T_2 <: T_3$ *then* $E \Vdash T_1 <: T_3$.

**Theorem 3.5** (Non-disjoint)**.** *If* $\Vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow F$ *and* $v$ *is a closed value so that* $E \Vdash v : T_1$ *and* $E \Vdash v : T_2$*, then* $E \Vdash F$.

**Theorem 3.6** (Reduction Preserves Types)**.** *If* $fv(e) = \emptyset$*,* $E \Vdash e : T$*, and* $e \to e'$ *then* $E \Vdash e' : T$.

**Definition 3.7** (Safety)**.** A closed expression $e$ is safe if and only if, in all evaluations of $e$, all assertions succeed.

**Theorem 3.8** (Safety)**.** *If* $\emptyset \Vdash e : T$ *then* $e$ *is safe.*

**Definition 3.9** (Opponent)**.** An opponent is an expression $e$ that does not contain asserts, free variables or names.

**Definition 3.10** (Robust Safety)**.** An expression $e$ is robustly safe if the application $O\ e$ is safe for any opponent $O$.

**Theorem 3.11** (Robust Safety for Formal-RCF$^{\forall}_{\wedge\vee}$)**.**
*If* $\emptyset \Vdash e : [\![\mathsf{Un}]\!]$ *then* $e$ *is robustly safe.*

**Corollary 3.12** (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. [BBF+08] (which is, however, a property of contexts, not of values), we define a notion of *robustly private values*.

**Definition 3.13** (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 3.11 (Robust Safety) that every value of type $\mathsf{Private}_C$ is robustly private in $e$ unless $C$, for any well-typed expression $e$.

**Theorem 3.14** (Value of Type Private ⇒ 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 [Coq09], together with most of the necessary lemmas. The only notable exception is Theorem 3.2 (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 [BBF+08, BBF+11, BHM08c]

At the moment, our Coq formalization[9] totals more than 14kLOC,[10] out of which more than 1.5kLOC are just definitions. We used Ott [SNO+10] to generate a large part of these definitions from a 1kLOC long Ott specification, but for the more complex rules we often needed to patch the output of Ott. We used LNgen [AW10] to generate an additional 25kLOC of infrastructure lemmas, which proved invaluable when working with the locally nameless representation.

During the formalization we found and fixed three problems in the paper proofs for the original RCF [BBF+08]. First, the "Public Down/Tainted Up" lemma was applying "Bound Weakening" in the wrong direction in the arrow type case, disregarding contravariance. Fixing this problem was easy, and only required proving a new lemma for replacing tainted bounds. Second, in the original RCF opponents can contain free names, so the proof of Theorem 3.11 (Robust Safety) used Theorem 3.8 (Safety) for a non-empty environment; however, safety was proved only for empty environments. We fixed this by not allowing the opponents to contain free names, since they can already generate names using the $(\nu a \updownarrow T)A$ expression.

**Lemma 3.15** (Replacing Tainted Bounds)**.**
*If* $E, x : T', E' \vdash U :: k$*, and* $E \vdash T$*, and* $E \vdash T' :: \mathtt{tnt}$ *then* $E, x : T, E' \vdash U :: k$*.*

Finally, the proof of the "Strengthening" lemma in the original RCF [BBF+08], and also in other refinement type systems for security [BHM08c], is wrong, and the status of the lemma in its original form is still unclear.

**Claim 3.16** (Strengthening)**.**
*If* $E, \mu, E' \vdash \mathcal{J}$ *and* $dom(\mu) \cap (free(E') \cup free(\mathcal{J})) = \emptyset$ *and* $E, E' \vdash forms(\mu)$*, 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 $dom(\mu) \cap (free(E') \cup free(A, T')) = \emptyset$ and $E, E' \vdash forms(\mu)$. However, in order to apply the induction hypothesis for $E, \mu, E' \vdash A : T$ we would need as a freshness condition that $dom(\mu) \cap free(T) = \emptyset$, 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 (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.

[9] `http://www.infsec.cs.uni-saarland.de/projects/F5/`
[10] All code size figures include whitespace and comments.

## 3.6. Implementation of Symbolic Cryptography

In contrast to process calculi for cryptographic protocols [AG99, AF01], $\mathrm{RCF}^{\forall}_{\wedge\vee}$ does not have any built-in construct to model cryptography. Cryptographic primitives are instead encoded using a dynamic sealing mechanism [Mor73], which is based on standard $\mathrm{RCF}^{\forall}_{\wedge\vee}$ constructs. The resulting symbolic cryptographic libraries are type-checked using the regular typing rules. The main advantage is that, adding a new primitive to the library does not involve changes in the calculus or in the soundness proofs: one has just to find a well-typed encoding of the desired cryptographic primitive. In addition, Backes et al. have recently [BMU10] 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. §3.6.1 overviews the dynamic sealing mechanism used in [BBF+08] to encode symbolic cryptography, while §3.6.2 and §3.6.3 show how our expressive type system can be used to improve this encoding and extend the class of supported protocols.

### 3.6.1. Dynamic Sealing

The notion of *dynamic sealing* was initially introduced by Morris [Mor73] as a protection mechanism for programs. Later, Sumii and Pierce [SP03, SP07] studied the semantics of dynamic sealing in a $\lambda$-calculus, observing a close correspondence with symmetric encryption. So the original spi-calculus [AG99], which baked in symmetric encryption, can essentially be seen as the pi-calculus [Mil99] with dynamic sealing.

In RCF [BBF+08] 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 $\mathsf{Un}$, 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 $\mathsf{Un}$, 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 §3.7.1). Moreover, if the type of a message encrypted with a public key is not tainted, then the public key is not public and cannot be given to the adversary. This may be problematic, for instance, when modeling authentication protocols based on public keys as the NSL protocol (see §3.2), where the type of the encrypted messages is neither public nor tainted.

### 3.6.2. Digital Signatures

In this section, we focus on digital signatures and show how union and intersection types can be used to solve the aforementioned problems. The signing key consists of the seal itself and is given type $\mathsf{SigKey}\langle T\rangle \triangleq \mathsf{Seal}\,\langle T\rangle$, as in the original RCF library [BBF+08]. The verification key, instead, is encoded as a function that $(i)$ takes the signature $x$ and the signed message $t$ as input; $(ii)$ calls the unsealing function to retrieve the message $y$ bound to $x$ in the secret list; and $(iii)$ returns $y$ if $y$ is equal to $t$ and fails otherwise. In this encoding, the verifier has to know the signed message in order to verify the signature. This is reasonable as, for efficiency reasons, one usually signs a hash of the message as opposed to the message in plain.

**Symbolic implementation of signing-verification key pair**

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

The type $\mathsf{VerKey}\langle T\rangle$ of a verification key is defined as $\mathsf{Un} \to \big((x : \top \to \{y : T \mid x = y\}) \wedge (\mathsf{Un} \to \mathsf{Un})\big)$. The verification key takes the signature of type $\mathsf{Un}$ as first argument. The second part of this type is an intersection of two types: The type $x : \top \to \{y : T \mid x = y\}$ is used to type-check honest callers: the signed message $x$ has any type (top type) and the message $y$ returned by the unsealing function has the stronger type $T$, which means that the unsealing function casts the type of the signed message from $\top$ down to $T$. This is safe since the sealing function is not public and can only be used to sign messages

of type $T$. The type $\mathsf{Un} \to \mathsf{Un}$ makes $\mathsf{VerKey}\langle T\rangle$ always public.[11] Hence, in contrast to [BBF+08], 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 [BCC04]). Finally, we present the typed interface of the functions to create and check signatures:

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

We type-check *sign* and *check* twice, to give them intersection types whose right-hand side is $\mathsf{Un}$. 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 §3.7 (dishonest verifier cases).

### 3.6.3. Public-Key Encryption

For public-key encryption we simply use a seal of type $\mathsf{Seal}\,\langle T \vee \mathsf{Un}\rangle$, i.e., $\mathsf{PrivKey}\langle T\rangle \triangleq \mathsf{Seal}\,\langle T \vee \mathsf{Un}\rangle$ and $\mathsf{PubKey}\langle T\rangle \triangleq (T \vee \mathsf{Un}) \to \mathsf{Un}$. This allows us to obtain the types described in §3.2.2. In contrast to [BBF+08], the encryption key is always public, even if the type $T$ of the encrypted message is not tainted.[12]

## 3.7. Encoding of Zero-knowledge

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

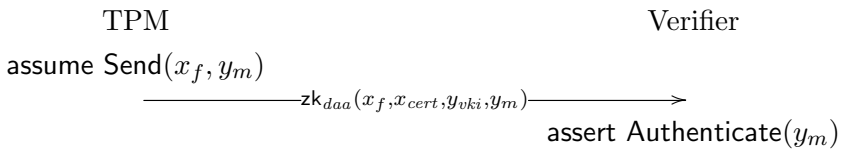
### 3.7.1. Illustrative Example: Simplified DAA-sign

We are going to illustrate our technique on a simplified variant of the Direct Anonymous Attestation (DAA) protocol [BCC04]. This simplified variant of the protocol was also considered in §2.2. 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

[11] 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.

[12] 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.

will only learn that a valid TPM signed that message, but without revealing the TPM's identity. The DAA protocol is composed of two sub-protocols: the *join protocol* and the *DAA-signing protocol*. The join protocol allows a TPM to obtain a certificate $x_{cert}$ from an entity called the issuer. This certificate is just a signature on the TPM's secret identifier $x_f$. The DAA-signing protocol enables a TPM to authenticate a message $y_m$ by proving to the verifier the knowledge of a valid certificate, but without revealing the TPM's identifier or the certificate. In this section, we focus on the DAA-signing protocol and we assume that the TPM has already completed the join protocol and received the certificate from the issuer. In the DAA-signing protocol the TPM sends to the verifier a zero-knowledge proof.




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

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

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

### 3.7.2. High-level Specification

Our high-level specification of non-interactive zero-knowledge proofs is similar in spirit to the symbolic representation of zero-knowledge proofs in a process calculus [BMU08, BHM08c]. For a specification the user needs to provide: (1) variables representing the witnesses and public values of the proof, (2) a Boolean formula over these variables representing the statement of the proof, (3) types for the variables, and, if desired, (4) a promise, i.e., a logical formula that is conveyed by the proof only if the prover is honest.

**High-level specification of simplified DAA**

$\mathsf{zkdef}\ daa =$
$\quad \mathsf{witness} = [x_f : T_{vki}, x_{cert} : \mathsf{Un}]$
$\quad \mathsf{matched} = [y_{vki} : \mathsf{VerKey}\langle T_{vki}\rangle]$
$\quad \mathsf{public} = [y_m : \mathsf{Un}]$
$\quad \mathsf{statement} = [x_f = \mathsf{check}\langle T_{vki}\rangle\ y_{vki}\ x_{cert}\ x_f]$

$\mathsf{promise} = [\mathsf{Send}(x_f, y_m)]$
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, so it is public.

**Statement.** The statement conveyed by a zero-knowledge proof is in general a positive Boolean formula over equality checks. In our 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 $\mathsf{Un}$. 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 $\mathsf{Un}$. Even though it has type $\mathsf{Un}$, 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 [BCC04, LHH+07]), 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.

### 3.7.3. Automatic Code Generation

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

**Generated typed interface for simplified DAA**

$\mathsf{create}_{daa} : T_{daa} \vee \mathsf{Un} \to \mathsf{Un}$ $\qquad$ $\mathsf{public}_{daa} : \mathsf{Un} \to \mathsf{Un}$
$\mathsf{verify}_{daa} : \mathsf{Un} \to ((y_{vki} : \mathsf{VerKey}\langle T_{vki}\rangle \to U_{daa}) \wedge \mathsf{Un} \to \mathsf{Un})$
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 $\mathsf{create}_{daa}$ 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 $\mathsf{Un}$ 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 $\mathsf{Un}$ so that it can be sent over the public network.

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

The function $\mathsf{verify}_{daa}$ 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 $\mathsf{Un}$. When called by a protocol participant, however, it takes as argument a candidate zero-knowledge proof of type $\mathsf{Un}$ and the verification key of the issuer with type $\mathsf{VerKey}\langle T_{daa}\rangle$. On successful verification, $\mathsf{verify}_{daa}$ returns $y_m$, the only public variable, but with a stronger type than in $\mathsf{public}_{daa}$. The function guarantees that the formula $\exists x_f, x_{cert}.\ \mathsf{OkTPM}(x_f) \wedge \mathsf{Send}(x_f, y_m)$ holds, where the witnesses are existentially quantified. The first conjunct, $\mathsf{OkTPM}(x_f)$, guarantees that if verification succeeds then the statement indeed holds, no matter what the origin of the proof is. This predicate is automatically extracted from the return type of the $check\langle T_{vki}\rangle$ function (see §3.6.2). The second conjunct $\mathsf{Send}(x_f, y_m)$ is the promise of the proof.

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

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

**Generated symbolic implementation for simplified DAA**

$\mathsf{verify}_{daa} = \lambda z : \mathsf{Un}.$
  $\mathsf{for}\ \alpha\ \mathsf{in}\ \mathsf{Un}; \mathsf{VerKey}\langle T_{vki}\rangle.\ \lambda y'_{vki} : \alpha.$
  $\mathsf{let}\ z' = (\mathsf{snd}\ k_{daa})\ z\ \mathsf{in}$ (1)
  $\mathsf{case}\ z'' = z' : \mathsf{Un} \vee T_{daa}\ \mathsf{in}$ (2)
  $\mathsf{let}\ (y_{vki}, y_m, x_f, x_{cert}, \_) = z''\ \mathsf{in}$ (3)
  $\mathsf{if}\ y_{vki} = y'_{vki}\ \mathsf{as}\ y''_{vki}\ \mathsf{then}$ (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 $\mathsf{verify}_{daa}$ 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 the DAA example the decision tree has a trivial structure with only one node.

Since the automatically generated implementation of zero-knowledge proofs relies on types and formulas provided by the user, which may both be wrong, the generated implementation is not guaranteed to fulfill its interface. We use our type-checker to check whether this is indeed the case. If type-checking the generated code against its interface succeeds, then this code can be safely used in protocol implementations. Note that because of the $\mathsf{for}$ and $\mathsf{case}$ constructs the body of $\mathsf{verify}_{daa}$ 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 $\mathsf{Un}$, while the result of the signature verification is of type $T_{vki}$. Since $\vdash \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\} \between \mathsf{Un} \rightsquigarrow \mathsf{false}$ by rules (ND Refine) and (ND Private Un), $\mathsf{false}$ is added to the environment in which $y_m$ is type-checked. The variable $y_m$ has type $\mathsf{Un}$ in this environment, but since this environment is inconsistent $y_m$ can also be given type $U_{daa}$.

## 3.8. Implementation

We have implemented a complete tool-chain for $\mathrm{RCF}^{\forall}_{\wedge\vee}$: it includes a type-checker for the type system described in §3.4, the automatic code generator for zero-knowledge described in §3.7, an interpreter, and a visual debugger.

The type-checker supports an extended syntax with respect to the one from §3.3, including: a simple module system, algebraic data types, recursive functions, type definitions, and mutable references. We use first-order logic with equality as the authorization logic and the type-checker invokes the Z3 SMT solver [dMB08] to discharge proof obligations. The type-checker performed very well in our experiments: it type-checks all our symbolic libraries and samples totaling more than 1.5kLOC in around 12 seconds, on a normal

laptop. The type-checker produces an XML log file containing the complete type derivation in case of success, and a partial derivation that leads to the typing error in case of failure. This can be inspected using our visualizer to easily detect and fix flaws in the protocol implementation. The type-checker also performs very limited type inference: it can infer the instantiation of some polymorphic functions from the type of the arguments, however, the user has to provide all the other typing annotations – we would like to improve the amount of type inference in the future (see §4.2.5 for a discussion).

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 and samples are available online[13].

## 3.9. Related Work on Unions and Intersections

The for construct for explicitly alternating type annotations was introduced by Pierce [Pie91, Pie97] as a generalization of an idea Reynolds [Rey96] used in Forsythe for giving intersection types to annotated lambda abstractions of the form $\lambda x{:}\tau_1..\tau_n.\, e$. In a Church-style system, however, the for construct does not have a clear operational semantics. Compagnoni [Com97] 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 §3.4). As discovered by Davies and Pfenning [DP00] 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 [HL]. Moreover, Davies and Pfenning [DP00] 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 $\mathrm{RCF}^{\forall}_{\wedge\vee}$, but erase it completely when translating values into Formal-$\mathrm{RCF}^{\forall}_{\wedge\vee}$, and use the value restriction on both levels to ensure soundness.

The case construct for eliminating union types was introduced by Pierce [Pie91] 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 [DP04] 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

[13] http://www.infsec.cs.uni-saarland.de/projects/F5/

an evaluation context restriction that recovers soundness, but this is not enough to make type-checking efficient. In recent work, Dunfield [Dun10], 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 [Zei08] tries to explain why phenomena such as the value and evaluation context restrictions can arise synthetically from a logical view of refinement typing.

## 3.10. Summary

In this chapter we have presented a new type system that combines refinement types with union types, intersection types, and polymorphic types. A novelty of the type system is its ability to reason statically about the disjointness of types. This extends the scope of the existing type-based analyses of protocol implementations to important classes of cryptographic protocols that were not covered so far, including protocols based on zero-knowledge proofs. Our type system comes with a mechanized proof of correctness and an efficient implementation[14].

[14]The implementation and formalization are available at
`http://www.infsec.cs.uni-saarland.de/projects/F5/`

# Chapter 4

# Conclusion and Future Work

## 4.1. Conclusion

In this thesis we have shown that the combination of union, intersection, and refinement types and static reasoning about the disjointness of types can be used for analyzing abstract models as well as concrete implementations of cryptographic protocols. To show this we have introduced two security type systems based on these features, one for a variant of the spi-calculus, and the other for a concurrent lambda calculus. We have developed mechanized formalizations of the type systems and proved formally that they guarantee the adherence of all well-typed protocols to their authorization policies, even in the presence of an arbitrary untyped attacker. The high expressive power of the type systems enables the analysis of important protocol classes that were out of scope for the existing type systems for cryptographic protocols. In this thesis, we have used the class of protocols based on non-interactive zero-knowledge proofs as the main running example, since it was not covered by any previous type system and was the original motivation for this work. The techniques we propose in this thesis are, however, also helpful for analyzing security despite compromised participants, for analyzing basic protocols that achieve authenticity by showing knowledge of secret data, and for giving a more faithful dynamic-sealing-based symbolic abstraction of asymmetric cryptography. The type-based analysis technique we have proposed is scalable and provides security proofs for an unbounded number of protocol executions. We have developed efficient implementations of our type systems based on state-of-the-art automatic reasoning tools.

## 4.2. Future Work

In this section we discuss several interesting directions for future work.

### 4.2.1. Semantic Subtyping for Higher-order Languages with Refinements

The subtyping relation of $\text{RCF}^{\forall}_{\wedge\vee}$ is purely syntactic, and this can be counter-intuitive for types like refinement, union and intersection types, about which programmers often have a strong set-theoretic intuition ("types are sets of values", "refinements are set comprehensions", "the intersection of two types is the intersection of their sets of values", etc.). Purely syntactic subtyping is, however, very coarse and often invalidates this intuition. In $\text{RCF}^{\forall}_{\wedge\vee}$ subtyping for refinement types is defined by the rules (Sub Refine Left) and (Sub Refine Right), while intersection types are just greatest-lower bounds and union types are just least-upper bounds. This allows us to prove inversion lemmas that are quite counter-intuitive:

**Lemma 4.1.** *If* $E \vdash \{x : T \mid C\} <: U_1 \to U_2$ *then* $E \vdash T <: U_1 \to U_2$.

**Lemma 4.2.** *If* $E \vdash T_1 \wedge T_2 <: U_1 \to U_2$ *then* $E \vdash T_1 <: U_1 \to U_2$ *or* $E \vdash T_2 <: U_1 \to U_2$.

Semantic subtyping solves this problem by being fully precise and completely in sync with the "types as sets of values" intuition [HVP05]. It also enables the type-checker to provide precise counterexamples when subtyping fails. Semantic subtyping was limited to first-order languages until Frisch et al. [FCB08] realized that for higher-order languages the model of types can be defined independently from values in order to avoid circularities, while still recovering the "types as sets of values" interpretation at a later stage. While in the first-order setting it is relatively easy to mix semantic subtyping with refinement types [BGHL10], the way Frisch et al. avoid circularities by abstracting away from values when defining the semantics of types appears to be incompatible with types that can directly mention values such as refinement types (e.g., refinement types can easily encode singleton types). Supporting semantic subtyping for a higher-order language with refinement types such as $\text{RCF}^{\forall}_{\wedge\vee}$ is currently an open problem.

### 4.2.2. Strong Secrecy and Observational Equivalence for $\text{RCF}^{\forall}_{\wedge\vee}$

The original work of Abadi on secrecy by typing [Aba99], as well as some of the follow up work by Abadi and Blanchet [AB03,AB05] 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 [GJ05]. For instance, the original type system for RCF [BBF+08] 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 §3.5). In recent work Fournet et

al. [FKS11] use parametricity [Mor73, SP03, SP07, BAF08] to prove strong secrecy by typing for a probabilistic variant of RCF. It would be interesting future work 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 [DKR09, BHM08b, MP11] and anonymity properties [CH02, BCC04, BCGS09, BLMP10] 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.

### 4.2.3. Supporting An Intuitionistic Authorization Logic with says Modality

The two authorization logics considered in this work do not have a says modality that attributes logical formulas to principals [ABLP93, Aba03, Aba07]. The says modality is particularly useful in the setting of security despite compromise [GP06, FGM07a], where it can easier encapsulate the effects of compromised participants, and ensure that the assertions made by certain principals will not affect the truth of the assertions made by others. Our design choice not to include the says modality is mostly for pragmatic reasons, since we prefer to be able to discharge proof obligations using the state-of-the-art automatic tools for classical first-order logic, and the says modality does not have a clear meaning in classical logic. For the applications that we have considered so far, we found it relatively easy to encode the effect of participant compromise explicitly into the authorization policy.

The requirements imposed by our soundness proof on the entailment relation of the authorization logic can, however, be satisfied by both classical and intuitionistic logics. We leave it as future work to support an intuitionistic authorization logic with says modality in a way that preserves efficient automation and practicality. One idea would be to translate the says modality into a standard modal logic [GA08], and then to use an encoding of this modal logic into classical first-order logic [HSW99, HS00]. This could be achieved by extending the translation by Garg and Abadi [GA08] to first-order quantifiers and equality. A similar idea would be to directly translate the says modality into intuitionistic first-order logic as done by Garg and Tschantz [GT08], and then use one of the automatic theorem-provers for intuitionistic first-order logic [Tam96, ROK07, MP09]. Finally, it would also be possible to develop special-purpose tools for automating specific authorization logics [Gar09], however, such an effort would not automatically benefit from the constant progress on general-purpose theorem provers.

### 4.2.4. Generalize the Syntactic Reasoning About Type Disjointness

Defining the non-disjointness judgment is challenging in our setting because kinding makes many types overlap. Because of kinding it is not enough to look only at the top-level type constructor to decide if two types can overlap. In the type system for the spi-calculus we use the logical characterization of kinding to capture the effect of kinding on type disjointness (§2.4.6). This allows us to give a general rule for comparing two generative types that have different top-level type constructors. While this covers the most common use cases (e.g., types Private and Un are both generative), it would be interesting to also study the types of terms created by constructors (e.g., when is a pair type disjoint from Un).

A much bigger challenge would be to try to bring more of these ideas to $\mathrm{RCF}^{\forall}_{\wedge\vee}$, where currently the only non-trivial base case for the non-disjointness relation is for Private and Un. The main reason for this is that in $\mathrm{RCF}^{\forall}_{\wedge\vee}$ there are no generative types, and our encoded type Private is the the only type we could find that is disjoint from Un. And even if we found a way to circumvent this problem, it would still be hard to capture the influence of kinding on type disjointness, since defining a logical characterization of kinding for $\mathrm{RCF}^{\forall}_{\wedge\vee}$ would be more challenging because of polymorphism.

Finally, it would be interesting to look for other contexts where reasoning about type disjointness is used, and try to see if any of our ideas are useful there. For instance, in dependently typed calculi such as the Calculus of Inductive Constructions [CH88, PPM90, CPM90] unsatisfiable equality assumptions can be added in the context introduced by a guard expression.

### 4.2.5. Type inference for $\mathrm{RCF}^{\forall}_{\wedge\vee}$

Our type-checker for $\mathrm{RCF}^{\forall}_{\wedge\vee}$ was very 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 fairly 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 [RKJ08, JMR11]. The good news is that intersection and union types can be very useful when devising precise type inference algorithms [BHM08c, Kob09].

### 4.2.6. Automatically Generating Concrete Cryptographic Implementations from Zero-knowledge Statement Specifications

In §3.7 of this thesis we have presented a general technique to automatically generate a *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. While several classes of statements admit efficient zero-knowledge proofs [CDS94, CL02, GS08, AFG+10], the effort of finding the right cryptographic schemes and implementing them efficiently is still substantial. Devising a library of reusable cryptographic schemes that can be used to implement zero-knowledge proofs can help in reducing this effort [BLMP10, BMP11, MP11]. Even more useful would be to automate the whole implementation process by devising a code generator that starts from a high-level specification of the statement and produces an efficient cryptographic implementation, by automatically selecting the right schemes. There has been interest recently in building such code generators for a particular kind of efficient zero-knowledge proofs called sigma-protocols [BBH+09, ABB+10, MEK+10]. The level of abstraction at which the zero-knowledge statements are specified in these tools is, however, much lower than our specifications from §3.7 and the specification language seems tailored towards sigma-protocols. It would be interesting to devise a more abstract specification language that can target a larger set of efficient zero-knowledge proof systems, including the techniques based on bilinear groups [GS08, AFG+10].

# Appendix A

# Typing Blind Signatures and Secret Hashes

For type-checking a model of the complete DAA protocol (§2.7) we extend the type system from §2.4 with blind signatures and secret hashes.

The only difference between secret hashes and the regular hashes defined in Chapter 2 is that the type of secret hashes has a more restrictive kinding rule (Kind SHash). While regular hashes are always public, and can thus be sent to the attacker, secret hashes are in general not public. For instance the $f$-value in the DAA protocol is a secret hash; revealing the $f$-value would allow attackers to impersonate the TPM (in fact the DAA protocol has a rogue tagging mechanism for rejecting $f$-values that are known to be leaked). Our type system statically enforces that secret hashes are not leaked to the attacker. An important consequence is that the type $\mathsf{SHash}(T)$ is disjoint from $\mathsf{Un}$ if $T$ is disjoint from $\mathsf{Un}$; we capture this in rule (ND SHash Un). This new rule is crucial for type-checking the DAA-signing protocol.

In order to prevent the issuer from learning f-values, DAA relies on blind signatures [Cha83]. The TPM sends the blinded f-value $\mathsf{blind}(f, r)$, where $r$ is a random blinding factor, to the issuer, which then produces the blind signature $\mathsf{bsign}(\mathsf{blind}(f, r), k_I)$. The TPM can later unblind the signature obtaining a signature $\mathsf{usign}(f, k_I)$ of the f-value, which can be publicly verified. The unblinding of blind signatures is done by the $\mathsf{unblind}$ destructor, while the verification of the unblinded signature is done by the $\mathsf{bcheck}$ destructor.

**Additional terms**

| | |
|---|---|
| $K, L, M, N ::=$ | terms |
| $\ldots$ | |

| | |
|---|---|
| $\mathsf{blind}(M, L)$ | blind term $M$ using blinding factor $L$ |
| $\mathsf{bsign}(M, K)$ | sign blinded term $M$ using key $K$ |
| $\mathsf{bvk}(K)$ | verification key corresponding to blind signing key $K$ |
| $\mathsf{usign}(M, K)$ | signature obtained after unblinding $M$, which was blind signed with key $K$ |
| $\mathsf{shash}(M)$ | secret hash of term $M$ |

**Additional destructors**

| $D ::=$ | destructors |
|---|---|
| ... | |
| $\mathsf{unblind}(M, L, K)$ | use blinding factor $L$ to unblind blind signature $M$ done with the signing key for verification key $K$ |
| $\mathsf{bcheck}(M, K)$ | check the unblinded signature $M$ using verification key $K$ |

The destructor reduction relation is extended with new rules for unblind and bcheck.

**Additional destructor reduction rules:** $D \Downarrow M$

$$\mathsf{unblind}(\mathsf{bsign}(\mathsf{blind}(M, L), K), L, \mathsf{bvk}(K)) \Downarrow \mathsf{usign}(M, K)$$
$$\mathsf{bcheck}(\mathsf{usign}(M, K), \mathsf{bvk}(K)) \Downarrow M$$

We add 5 types for type-checking protocols based on blind signatures: The type $\mathsf{Blinder}(T)$ describes blinding factors that can only be used to blind terms of type $T$; $\mathsf{Blinded}(T)$ describes the result of blinding a term of type $T$; $\mathsf{BSigKey}(T, z.\, C)$ contains signing keys that can only be used to sign blinded terms of type $T$ for which additionally the formula $C$ holds (i.e., blinded terms of type $\{z : \mathsf{Blinded}(T) \mid C\}$); $\mathsf{BVerKey}(T, z.\, C)$ describes verification keys corresponding to blind signing keys of type $\mathsf{BSigKey}(T, z.\, C)$; and $\mathsf{USigned}(T)$ describes unblinded signatures on terms of type $T$.

**Additional types**

| $T, U, V ::=$ | types |
|---|---|
| ... | |
| $\mathsf{Blinder}(T)$ | blinding factor that only blinds terms of type $T$ |
| $\mathsf{Blinded}(T)$ | the result of blinding a term of type $T$ |
| $\mathsf{BSigKey}(T, x.\, C)$ | signing key that only signs blinded terms of type $T$ for which additionally $C$ holds (scope of $x$ is $C$) |
| $\mathsf{BVerKey}(T, x.\, C)$ | verification key corresponding to blind signing key of type $\mathsf{BSigKey}(T, x.\, C)$ (scope of $x$ is $C$) |
| $\mathsf{USigned}(T)$ | unblinded signature on a term of type $T$ |
| $\mathsf{SHash}(T)$ | secret hash of a term of type $T$ |

We call a type $T$ generative iff $T \in \{\ldots, \mathsf{Blinder}(T), \mathsf{BSigKey}(T, x.\, C)\}$

The kinding and subtyping rules for blind signatures are quite similar to but not exactly the same as the ones for symmetric encryption and regular signatures. Blinding factors behave like symmetric encryption keys, so $\mathsf{Blinder}(T)$ has the same kinding and subtyping rules as $\mathsf{SymKey}(T)$. The type of blinded terms $\mathsf{Blinded}(T)$ is always public, but in general not tainted (unlike the result type of the senc constructor, which is Un). The types of blind signing and verification keys $\mathsf{BSigKey}(T, x.\,C)$ and $\mathsf{BVerKey}(T, x.\,C)$ have the same kinding rules as $\mathsf{SigKey}(\{x : \mathsf{Blinded}(T) \mid C\})$ and, respectively, $\mathsf{VerKey}(\{x : \mathsf{Blinded}(T) \mid C\})$. Subtyping is, however, stronger for $\mathsf{BSigKey}(T, x.\,C)$ and $\mathsf{BVerKey}(T, x.\,C)$ compared to the types for regular signatures. In particular we can show that if $E \vdash \mathsf{BSigKey}(T_1, x.\,C_1) <:> \mathsf{BSigKey}(T_2, x.\,C_2)$ then $E \vdash T_1 <:> T_2$. This property is crucial when proving blind signatures sound, but it does not hold for regular signature types of refinement types (i.e., $\mathsf{SigKey}(\{x : \mathsf{Blinded}(T) \mid C\})$). Another difference compared to regular signatures is that the type $\mathsf{BVerKey}(T, x.\,C)$ is invariant while $\mathsf{VerKey}(T)$ is covariant. Finally, the kinding and subtyping rules for $\mathsf{USigned}(T)$ are the same as for $\mathsf{Signed}(T)$.

**Additional kinding rules:** $E \vdash T :: k$ **for** $k \in \{\mathsf{pub}, \mathsf{tnt}\}$

(Kind Blinder)
$$\frac{E \vdash T :: \mathsf{pub} \quad E \vdash T :: \mathsf{tnt}}{E \vdash \mathsf{Blinder}(T) :: k}$$

(Kind Blinded Pub)
$$\frac{E \vdash T\ ok}{E \vdash \mathsf{Blinded}(T) :: \mathsf{pub}}$$

(Kind Blinded Tnt)
$$\frac{E \vdash T :: \mathsf{pub} \quad E \vdash T :: \mathsf{tnt}}{E \vdash \mathsf{Blinded}(T) :: \mathsf{tnt}}$$

(Kind BSigKey)
$$\frac{E \vdash T :: \mathsf{pub} \quad E \vdash T :: \mathsf{tnt} \quad E, x : \top \vdash C}{E \vdash \mathsf{BSigKey}(T, x.\,C) :: k}$$

(Kind BVerKey Pub)
$$\frac{E \vdash T\ ok \quad E, x : \top \vdash C\ ok}{E \vdash \mathsf{BVerKey}(T, x.\,C) :: \mathsf{pub}}$$

(Kind BVerKey Tnt)
$$\frac{E \vdash T :: \mathsf{pub} \quad E \vdash T :: \mathsf{tnt} \quad E, x : \top \vdash C}{E \vdash \mathsf{BVerKey}(T, x.\,C) :: \mathsf{tnt}}$$

(Kind USigned Pub)
$$\frac{E \vdash T :: \mathsf{pub}}{E \vdash \mathsf{USigned}(T) :: \mathsf{pub}}$$

(Kind USigned Tnt)
$$\frac{E \vdash T\ ok}{E \vdash \mathsf{USigned}(T) :: \mathsf{tnt}}$$

(Kind SHash)
$$\frac{E \vdash T :: \mathsf{pub} \quad E \vdash T :: \mathsf{tnt}}{E \vdash \mathsf{SHash}(T) :: k}$$

**Additional subtyping rules:** $E \vdash T <: U$

(Sub Blinder Inv)
$$\frac{E \vdash T <:> U}{E \vdash \mathsf{Blinder}(T) <: \mathsf{Blinder}(U)}$$

(Sub Blinded Inv)
$$\frac{E \vdash T <:> U}{E \vdash \mathsf{Blinded}(T) <: \mathsf{Blinded}(U)}$$

(Sub BSigKey Inv)
$$\frac{E \vdash T_1 <:> T_2 \quad E, x : \top \vdash C_1 \Leftrightarrow C_2}{E \vdash \mathsf{BSigKey}(T_1, x.\,C_1) <: \mathsf{BSigKey}(T_1, x.\,C_2)}$$

(Sub USigned Inv)
$$\frac{E \vdash T <:> U}{E \vdash \mathsf{USigned}(T) <: \mathsf{USigned}(U)}$$

(Sub BVerKey Inv)
$$\frac{E \vdash T_1 <:> T_2 \quad E, x : \top \vdash C_1 \Leftrightarrow C_2}{E \vdash \mathsf{BVerKey}(T_1, x.\, C_1) <: \mathsf{BVerKey}(T_1, x.\, C_2)}$$

(Sub SHash Inv)
$$\frac{E \vdash T <:> U}{E \vdash \mathsf{SHash}(T) <: \mathsf{SHash}(U)}$$

The secret hash type $\mathsf{SHash}(T)$ is public only if $T$ is both public and tainted, which allows us to add a strong non-disjointness rule comparing $\mathsf{SHash}(T)$ with $\mathsf{Un}$.

**Additional non-disjointness rules** $E \vdash T \between U \rightsquigarrow C$

(ND SHash Un)
$$\frac{E \vdash T \between \mathsf{Un} \rightsquigarrow C}{E \vdash \mathsf{SHash}(T) \between \mathsf{Un} \rightsquigarrow C}$$

The new term typing rules are as one would expect. Rule (Term BSign) reflects again the correspondence between $\mathsf{BSigKey}(T, z.\, C)$ and $\mathsf{SigKey}(\{z : \mathsf{Blinded}(T) \mid C\})$. Rule (Term USign) is not directly used by our type-checker, since protocols do not contain terms using $\mathsf{usign}$; instead terms that use $\mathsf{usign}$ are generated only by the $\mathsf{unblind}$ destructor. It is, however, crucial for Lemma A.1 (Unblind consistent) that the result of the $\mathsf{unblind}$ destructor can still be type-checked.

**Additional term typing rules:** $E \vdash M : T$

(Term Blind)
$$\frac{E \vdash M : T \quad E \vdash L : \mathsf{Blinder}(T)}{E \vdash \mathsf{blind}(M, L) : \mathsf{Blinded}(T)}$$

(Term BVK)
$$\frac{E \vdash K : \mathsf{BSigKey}(T, z.\, C)}{E \vdash \mathsf{bvk}(K) : \mathsf{BVerKey}(T, z.\, C)}$$

(Term BSign)
$$\frac{E \vdash N : \{z : \mathsf{Blinded}(T) \mid C\} \quad E \vdash K : \mathsf{BSigKey}(T, z.\, C)}{E \vdash \mathsf{bsign}(N, K) : \mathsf{Un}}$$

(Term SHash)
$$\frac{E \vdash M : T}{E \vdash \mathsf{shash}(M) : \mathsf{SHash}(T)}$$

(Term USign)
$$\frac{E \vdash M : \{x : T \mid \exists y.\, C\{\mathsf{blind}(x, y)/z\}\} \quad E \vdash K : \mathsf{BSigKey}(T, z.\, C)}{E \vdash \mathsf{usign}(M, K) : \mathsf{USigned}(T)}$$

We introduce two new destructor typing rules. Rule (Dtor BCheck) is particularly interesting, since it returns a very strong type. When successfully checking a blind signature with a verification key of type $\mathsf{BVerKey}(T, z.\, C)$ we know not only that the resulting term $M$ has type $T$, but also that the formula $C$ holds for the blinding of $M$ with some unknown blinding factor $y$. We use existential quantification in the authorization logic to reflect the fact that the blinding factor $y$ is unknown to the principal signing $M$ blindly.

**Additional destructor typing rules:** $E \vdash D : T$

(Dtor Unblind)

$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash L : \mathsf{Blinder}(T) \quad E \vdash K : \mathsf{BVerKey}(T, z.\, C)}{E \vdash \mathsf{unblind}(M, L, K) : \mathsf{USigned}(T)}$$

(Dtor BCheck)

$$\frac{E \vdash M : \mathsf{USigned}(T) \quad E \vdash K : \mathsf{BVerKey}(T, z.\, C)}{E \vdash \mathsf{bcheck}(M, K) : \{x : T \mid \exists y.\, C\{\mathsf{blind}(x, y)/z\}\}}$$

We have extended the Coq proof of destructor consistency (Lemma 2.29) to cover blind signatures.

**Lemma A.1** (Unblind consistent)**.** *For all closed terms $M$, $L$ and $K$ such that $E \vdash \mathsf{bsign}(\mathsf{blind}(M, L), K) : \mathsf{Un}$ and $E \vdash L : \mathsf{Blinder}(T)$ and $E \vdash \mathsf{bvk}(K) : \mathsf{BVerKey}(T, z.\, C)$ we have that $E \vdash \mathsf{usign}(M, K) : \mathsf{USigned}(T)$.*

**Lemma A.2** (BCheck consistent)**.**
*For all closed terms $N$ and $K$ such that $E \vdash \mathsf{usign}(N, K) : \mathsf{USigned}(T)$ and $E \vdash \mathsf{bvk}(K) : \mathsf{BVerKey}(T, z.\, C)$ we have that $E \vdash N : \{x : T \mid \exists y.\, C\{\mathsf{blind}(x, y)/z\}\}$.*

Finally, we extend the statement-based inference judgement with two additional rules. The rule for secret hashes (Sinfer SHash) is the same as for regular hashes (Sinfer Hash), while the rule for checking a blind signature (Sinfer BCheck) closely corresponds to (Sinfer Check) in §2.4.9.

**Additional statement-based inference rules:** $E \vdash [|B|]_{E_{old}, C_{old}} \rightsquigarrow E_{new}, C_{new}$

(Sinfer BCheck)

$$\frac{E, E_{old}, \{C_{old}\} \vdash v_K : \mathsf{BVerKey}(T, z.\, C) \quad T' = \{x : \mathsf{Blinded}(T) \mid C\} \quad E \vdash T'\ ok \quad E \vdash (E_{old}, C_{old})[v_N : \{\!|\, \mathsf{fkind}(E, T', \mathsf{tnt}) \,|\!\} \vee T'] \rightsquigarrow E_{new}, C_{new}}{E \vdash [|\mathsf{bcheck}(v_M, v_K) \rightsquigarrow v_N|]_{E_{old}, C_{old}} \rightsquigarrow E_{new}, C_{new}}$$

(Sinfer SHash)

$$\frac{E, E_{old}, \{C_{old}\} \vdash v_M : \mathsf{SHash}(T) \quad E \vdash T\ ok \quad E \vdash (E_{old}, C_{old})[v_N : \{\!|\, \mathsf{fkind}(E, T, \mathsf{tnt}) \,|\!\} \vee T] \rightsquigarrow E', C'}{E \vdash [|v_M = \mathsf{shash}(v_M)|]_{E_{old}, C_{old}} \rightsquigarrow E_{new}, C_{new}}$$

# Appendix B

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

## B.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 $\alpha$ | 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 $\alpha$ | type variable |
| ee_kind $\alpha$ $k$ | 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 |

## B.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**

$\llbracket M\ N \rrbracket = \mathsf{e\_app}\ \llbracket M \rrbracket\ \llbracket N \rrbracket$
$\llbracket M\langle T \rangle \rrbracket = \mathsf{e\_inst}\ \llbracket M \rrbracket$
$\llbracket \mathsf{let}\ x = A\ \mathsf{in}\ B \rrbracket = \mathsf{e\_let}\ \llbracket A \rrbracket\ (\mathsf{close}_x\ \llbracket B \rrbracket)$
$\llbracket \mathsf{let}\ (x, y) = M\ \mathsf{in}\ A \rrbracket =$
  $\mathsf{e\_let}\ (\mathsf{e\_first}\ \llbracket M \rrbracket)$
    $(\mathsf{e\_second}\ \llbracket M \rrbracket\ (\mathsf{close}_y\ (\mathsf{close}_x\ \llbracket A \rrbracket)))$
$\llbracket \mathsf{match}\ M\ \mathsf{with}\ \mathsf{inl}\ x \Rightarrow A \mid \mathsf{inr}\ y \Rightarrow B \rrbracket =$
  $\mathsf{e\_match}\ \llbracket M \rrbracket\ (\mathsf{close}_x\ \llbracket A \rrbracket)\ (\mathsf{close}_y\ \llbracket B \rrbracket)$
$\llbracket \mathsf{unfold}_{\mu\alpha.\,T}\ M \rrbracket = \mathsf{e\_unfold}\ \llbracket M \rrbracket$
$\llbracket \mathsf{case}\ x = M : T \vee U\ \mathsf{in}\ A \rrbracket = \mathsf{e\_let}\ \llbracket M \rrbracket\ (\mathsf{close}_x\ \llbracket A \rrbracket)$

$\lfloor\!\lfloor \mathsf{if}\ M = N\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B \rfloor\!\rfloor =$
$\quad \mathsf{e\_if}\ \lfloor\!\lfloor M \rfloor\!\rfloor\ \lfloor\!\lfloor N \rfloor\!\rfloor\ (\mathsf{close}_x\ \lfloor\!\lfloor A \rfloor\!\rfloor)\ \lfloor\!\lfloor B \rfloor\!\rfloor$
$\lfloor\!\lfloor (\nu a \updownarrow T)A \rfloor\!\rfloor = \mathsf{e\_new}\ (\mathsf{close}_a\ \lfloor\!\lfloor A \rfloor\!\rfloor)$
$\lfloor\!\lfloor A \Rsh B \rfloor\!\rfloor = \mathsf{e\_fork}\ \lfloor\!\lfloor A \rfloor\!\rfloor\ \lfloor\!\lfloor B \rfloor\!\rfloor$
$\lfloor\!\lfloor a!N \rfloor\!\rfloor = \mathsf{e\_send}\ a\ \lfloor\!\lfloor N \rfloor\!\rfloor$
$\lfloor\!\lfloor a? \rfloor\!\rfloor = \mathsf{e\_recv}\ a$
$\lfloor\!\lfloor \mathsf{assume}\ C \rfloor\!\rfloor = \mathsf{e\_assume}\ \lfloor\!\lfloor C \rfloor\!\rfloor$
$\lfloor\!\lfloor \mathsf{assert}\ C \rfloor\!\rfloor = \mathsf{e\_assert}\ \lfloor\!\lfloor C \rfloor\!\rfloor$

**Erasure for types**

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

**Erasure for typing environments**

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

## B.3. Local Closure

**Locally closed values, formulas and expressions**

$$\frac{}{\mathsf{lc}\ (\mathsf{v\_var\_f}\ x)} \qquad \frac{}{\mathsf{lc}\ (\mathsf{v\_unit})} \qquad \frac{\forall x.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ e)}{\mathsf{lc}\ (\mathsf{v\_lam}\ e)}$$

$$\frac{\mathsf{lc}\ e}{\mathsf{lc}\ (\mathsf{v\_tlam}\ e)} \qquad \frac{\mathsf{lc}\ v_1 \quad \mathsf{lc}\ v_2}{\mathsf{lc}\ (\mathsf{v\_pair}\ v_1\ v_2)} \qquad \frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{v\_inx}\ h\ v)}$$

$$\frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{v\_fold}\ v)} \quad \frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{f\_pred}\ P\ v)} \quad \frac{\mathsf{lc}\ v_1 \quad \mathsf{lc}\ v_2}{\mathsf{lc}\ (\mathsf{f\_eq}\ v_1\ v_2)}$$

$$\frac{\mathsf{lc}\ F_1 \quad \mathsf{lc}\ F_2}{\mathsf{lc}\ (\mathsf{f\_and}\ F_1\ F_2)} \quad \frac{\mathsf{lc}\ F_1 \quad \mathsf{lc}\ F_2}{\mathsf{lc}\ (\mathsf{f\_or}\ F_1\ F_2)} \quad \frac{\mathsf{lc}\ F}{\mathsf{lc}\ (\mathsf{f\_not}\ F)}$$

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

$$\frac{\mathsf{lc}\ v}{\mathsf{lc}\ (\mathsf{e\_val}\ v)} \quad \frac{\mathsf{lc}\ v_1 \quad \mathsf{lc}\ v_2}{\mathsf{lc}\ (\mathsf{e\_app}\ v_1\ v_2)}$$

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

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

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

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

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

$$\frac{\forall a.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{name\_f}\ a)}\ e)}{\mathsf{lc}\ (\mathsf{e\_new}\ e)} \quad \frac{\mathsf{lc}\ e_1 \quad \mathsf{lc}\ e_2}{\mathsf{lc}\ (\mathsf{e\_fork}\ e_1\ e_2)}$$

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

**Locally closed types**

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

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

$$\frac{\forall \alpha.\ \mathsf{lc}\ (\mathsf{open}_{(\mathsf{t\_var\_f}\ \alpha)}\ T)}{\mathsf{lc}\ (\mathsf{t\_rec}\ T)} \quad \frac{}{\mathsf{lc}\ (\mathsf{t\_var\_f}\ \alpha)}$$

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

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

**Locally closed environment entries**

$$\frac{}{\mathsf{lc}\ (\mathsf{ee\_tvar}\ \alpha)} \qquad \frac{}{\mathsf{lc}\ (\mathsf{ee\_kind}\ \alpha\ k)}$$

$$\frac{\mathsf{lc}\ T}{\mathsf{lc}\ (\mathsf{ee\_var}\ x\ T)} \qquad \frac{\mathsf{lc}\ T}{\mathsf{lc}\ (\mathsf{ee\_chan}\ a\ T)} \qquad \frac{\mathsf{lc}\ F}{\mathsf{lc}\ (\mathsf{ee\_ok}\ F)}$$

## B.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)
$$\frac{\mathsf{lc}\ e}{\mathsf{e\_fork}\ (\mathsf{e\_val}\ \mathsf{v\_unit})\ e \Rrightarrow e}$$

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

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

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

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

## B.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)
$$\frac{\mathsf{lc}\ v \qquad S \models F}{S\{v/x\} \models F\{v/x\}}$$

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

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

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

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

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

## B.6. Typing Judgements

**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)
$$\frac{\begin{array}{c}\mathsf{lc}\ T \quad E \Vdash \diamond \quad \mathsf{fv\_type}\ T \subseteq \mathsf{dom\_v}\ E \\ \mathsf{fn\_type}\ T \subseteq \mathsf{dom\_n}\ E \quad \mathsf{ftv\_type}\ T \subseteq \mathsf{dom\_tv}\ E\end{array}}{E \Vdash T}$$

**Entailed formula** $E \Vdash F$

(entails derive)
$$\frac{\begin{array}{c}E \Vdash \diamond \quad \mathsf{fv\_form}\ F \subseteq \mathsf{dom\_v}\ E \quad \mathsf{fn\_form}F \subseteq \mathsf{dom\_n}\ E \\ (\mathsf{forms\_env}\ E) \models F\end{array}}{E \Vdash F}$$

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

(kind pair)
$$\frac{\begin{array}{c}E \Vdash T :: k \\ \forall x \notin L.\ ((\mathsf{ee\_var}\ x\ T) :: E) \Vdash (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U) :: k\end{array}}{E \Vdash (\mathsf{t\_pair}\ T\ U) :: k}$$

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

(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)

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

$\alpha$ **has variance** $\eta$ **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)

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

(hv pair)

$$\frac{\begin{array}{c}\mathsf{has\_variance}\ \alpha\ \eta\ T \\ \forall x \notin L.\ \mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ U)\end{array}}{\mathsf{has\_variance}\ \alpha\ \eta\ (\mathsf{t\_pair}\ T\ U)}$$

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

(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)

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

(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)

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

(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** $\Vdash \mathsf{NonDisj}\ T\ U \rightsquigarrow F$

(nd private un)

$$\frac{\mathsf{lc}\ F \quad fv(F) = \emptyset}{\Vdash \mathsf{NonDisj}\ [\![\mathsf{Private}_F]\!]\ \mathsf{t\_unit} \rightsquigarrow F}$$

(nd true)

$$\frac{\mathsf{lc}\ T_1 \quad \mathsf{lc}\ T_2}{\Vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow \mathsf{f\_true}}$$

(nd sym)

$$\frac{\Vdash \mathsf{NonDisj}\ T_2\ T_1 \rightsquigarrow F}{\Vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow F}$$

(nd refine)

$$\frac{\Vdash \mathsf{NonDisj}\ T_1\ T_2 \rightsquigarrow F \quad \forall x \notin L.\ \mathsf{lc}\ \mathsf{open}_{(\mathsf{v\_var\_f}\ x)}\ F_1}{\Vdash \mathsf{NonDisj}\ (\mathsf{t\_refine}\ T_1\ F_1)\ T_2 \rightsquigarrow F}$$

(nd pair)

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

(nd sum)

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

(nd rec)

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

(nd and)

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

(nd or)

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

# Appendix C

# 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 verification succeeds then the verification function returns the public values of the proof. The public values can also be obtained with the third function, without checking the validity of the proof.

## C.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 [BMU08, BHM08c]. 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: $\mathsf{matched}$ if the value of $x$ is revealed by the proof and the verifier checks the value of $x$ for equality with a known value, $\mathsf{public}$ if $x$ has a public value obtained by the verifier after checking the proof, $\mathsf{witness}$ 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 $\mathrm{RCF}^{\forall}_{\wedge\vee}$ functions applied to variables, as well as conjunctions and disjunctions of such basic statements.

**Syntax of zero-knowledge statements**

| $S, R ::=$ | statements |
|---|---|
| $x = f\langle\widetilde{T}\rangle\ x_1 \dots x_n$ | 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 $\mathrm{RCF}^{\forall}_{\wedge\vee}$ functions to their arguments we obtain a valid Boolean formula. We assume that the $\mathrm{RCF}^{\forall}_{\wedge\vee}$ functions occurring in the statement have deterministic behavior[1], 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 $\mathsf{check}$ 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.

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

**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 $\mathsf{Un}$. Even though it has type $\mathsf{Un}$, it would break the anonymity of the user to give the certificate sort $\mathsf{public}$, since the verifier could then always distinguish if two consecutive requests come from the same user or not (as in the pseudonymous version of DAA). While we assume that if $sort_{\mathscr{S}}(x) = \mathsf{public}$ 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[2], 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.

## C.2. Automatic Code Generation

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

## C.3. Typed Interface

The interface generated for a specification $\mathscr{S}$ contains three functions[3] that share hidden state (a seal for values of type $\tau_{\mathscr{S}}$):

$$\begin{aligned}
&\mathsf{create}_{\mathscr{S}} \;:\; \tau_{\mathscr{S}} \to \mathsf{Un} \\
&\quad \text{where } \tau_{\mathscr{S}} = \mathsf{t_or}\ \mathsf{Un}\ \textstyle\sum_{x \in \mathsf{vars}_{\mathscr{S}}} x : t_{\mathscr{S}}(x).\{P_{\mathscr{S}}\} \\
&\mathsf{public}_{\mathscr{S}} \;:\; \mathsf{Un} \to \mathsf{Un} \\
&\mathsf{verify}_{\mathscr{S}} \;:\; \mathsf{Un} \to (\mathsf{Un} \;\wedge \prod_{y \in \mathsf{matched}_{\mathscr{S}}} y : t_{\mathscr{S}}(y). \\
&\qquad\qquad \sum_{z \in \mathsf{returned}_{\mathscr{S}}} z : t_{\mathscr{S}}(y).\{ \underset{\widetilde{x} = \mathsf{witness}_{\mathscr{S}}}{\exists \widetilde{x}.} F(S_{\mathscr{S}}, E) \wedge P_{\mathscr{S}}\}\ )
\end{aligned}$$

[2] Signature proofs of knowledge have this property [BCC04, LHH+07].

[3] 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 $\widetilde{y} = \mathsf{matched}_{\mathscr{S}}$.

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 $\mathsf{Un}$ 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 $\mathsf{Un}$ so that it 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 $\mathsf{Un}$ and returns the tuple of public values, also at type $\mathsf{Un}$.

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 $\mathsf{Un}$. When called by a protocol participant, however, it takes as argument a candidate zero-knowledge proof of type $\mathsf{Un}$ 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 [BHM08c]), 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** $F(S, E)$

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

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 §3.6, 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 §3.4.1. Finally, after removing the trivial equality we obtain that:

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

## C.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.

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

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

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 $\tau_S$ is a union type. In case $z'$ has type $\mathsf{Un}$ then the declared return type $\mathsf{Un}$ 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 $\mathsf{Un}$.

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 $\mathsf{Un}$, and values for the matched variables. Since the type $\mathsf{Verify}_S$ contains an intersection type ($\mathsf{Un}$ is one of the branches and this makes the type $\mathsf{Verify}_S$ public) we use a $\mathsf{for}$ 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 $\mathsf{Un}$. 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(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.

$ver_S = \lambda k_{unsealing} : \mathsf{Un} \to \tau_S.\, \lambda z : \mathsf{Un}.$
$\quad \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{\mathsf{Un}}; \widetilde{t_S(y)}.$
$\quad \lambda y'_1 : \alpha_1. \ldots \lambda y'_n : \alpha_n.$
$\quad (*1*)\ \mathsf{let}\ z' = k_{unsealing}\ z\ \mathsf{in}$
$\quad (*2*)\ \mathsf{case}\ z'' = z' : \mathsf{Un} \vee \sum_{x \in \mathsf{vars}(S)} x : t_S(x).\{P_S\}\ \mathsf{in}$
$\quad (*3*)\ \mathsf{let}\ (\widetilde{y}, \widetilde{z}, \widetilde{x}) = z''\ \mathsf{in}$
$\quad (*4*)\ \mathsf{if}\ (\widetilde{y}) = (y')\ \mathsf{as}\ (\widetilde{y''})\ \mathsf{then}$
$\quad (*5*)\quad$ "exp(prime($S$), $\{\widetilde{y''}/\widetilde{y}\}$)"
$\qquad\quad \mathsf{else}\ \mathit{failwith}$ "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 $R = (x = f\langle\widetilde{T}\rangle\ x_1 \ldots x_n)$. By slightly abusing notation, we denote this decomposition as $S\,[R_1, \ldots, R_n]$. We then convert $S\,[R_1, \ldots, R_n]$ into a decision tree. Decisions trees are defined by the following grammar:

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

We implement this as a function called prime, that given a decomposed statement $S\,[R_1, \ldots, R_n]$ produces its prime tree, i.e., an ordered and reduced decision tree; we refer the interested reader to [Bry86, SB08a] for the details.

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

**Converting Decision Trees to Expressions**

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

Other than the decision tree, this function takes as argument a substitution $\sigma$ 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 $(\sigma(x_1), \ldots, \sigma(x_n))$, 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 $x = f\langle\widetilde{T}\rangle\ x_1 \ldots x_n$ 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 $\sigma(x_1) \ldots \sigma(x_n)$. 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 $\sigma$ 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.

## C.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.

# Bibliography


[AB03] Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 3(298):387–415, 2003. 14, 83, 104, 119

[AB05] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005. 7, 20, 24, 119

[Aba99] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999. 14, 119

[Aba03] Martín Abadi. Logic in access control. In *Proc. 18th IEEE Symposium on Logic in Computer Science (LICS)*, pages 228–233. IEEE Computer Society Press, 2003. 5, 120

[Aba07] Martín Abadi. Access control in a core calculus of dependency. *Electronic Notes on Theoretical Computer Science*, 172:5–31, 2007. 120

[ABB+05] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In *Proc. 17th International Conference on Computer Aided Verification (CAV 2005)*, pages 281–285. Springer-Verlag, 2005. 15

[ABB+10] José Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas 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. 15, 122

[ABLP93] Martín Abadi, Michael Burrows, Butler W. Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):706–734, 1993. 120

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, 1993. 99

[AC96] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer, 1996. 40

[ACC+08] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos 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. 5

[ACP+08] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proc. 35th Symposium on Principles of Programming Languages (POPL '08)*, pages 3–15, 2008. 61, 83, 105, 106

[Adi08] Ben Adida. Helios: Web-based open-audit voting. In *Proc 17th USENIX Security Symposium*, pages 335–348, 2008. 6

[AF01] 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. 109

[AFG+10] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Structure-preserving signatures and commitments to group elements. In *Advances in Cryptology - CRYPTO 2010*, pages 209–236, 2010. 6, 122

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999. 7, 10, 20, 92, 109

[AGJ11] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. Draft, 2011. 86

[AW10] Brian E. Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Draft available at `http://www.cis.upenn.edu/~sweirich/papers/lngen/`, 2010. 61, 108

[BAF08] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, February–March 2008. 14, 120

[BBF+08] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio 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 [BBF+11]. 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`. 5, 10, 11, 14, 15, 20, 27, 30, 31, 40, 60, 64, 82, 83, 84, 87, 88, 92, 93, 94, 95, 99, 100, 101, 106, 107, 108, 109, 110, 111, 119

[BBF+11] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(2):8, 2011. 10, 11, 82, 108, 155

[BBH+09] Endre Bangerter, Thomas Briner, Wilko Henecka, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. Automatic generation of sigma-protocols. In *6th European Workshop on Public Key Infrastructures, Services and Applications (EuroPKI 2009)*, pages 67–82, 2009. 122

[BBH11] Michael Backes, Alex Busenius, and Cătălin Hriţcu. On the development and formalization of an extensible code generator for real life security protocols. Submitted, October 2011. 24

[BCC04] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proc. 11th ACM Conference on Computer and Communications Security*, pages 132–145. ACM Press, 2004. 6, 7, 10, 11, 14, 16, 52, 75, 76, 81, 84, 111, 113, 120, 147

[BCC+09] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. Randomizable proofs and delegatable anonymous credentials. In *Advances in Cryptology - CRYPTO 2009*, pages 108–125, 2009. 7

[BCD+09a] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Proc. 22th IEEE Symposium on Computer Security Foundations (CSF)*, pages 124–140. IEEE Computer Society Press, 2009. 14

[BCD+09b] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *13th International Conference on Financial Cryptography and Data Security*, pages 325–343, 2009. 15

[BCEM11] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Resource-aware authorization policies for statically typed cryptographic protocols. In *Proc. 24th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, 2011. To appear. 14

[BCFM07] Michael Backes, Agostino Cortesi, Riccardo Focardi, and Matteo 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. 14

[BCFZ08] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Eugen Zălinescu. Cryptographically verified implementations for TLS. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 459–468. ACM Press, 2008. 86

[BCGS09] Patrik Bichsel, Jan Camenisch, Thomas Groß, and Victor Shoup. Anonymous credentials on a standard java card. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS)*, pages 600–610, 2009. 7, 120

[BCJ+06] Frederick Butler, Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Christopher Walstad. Formal analysis of Kerberos 5. *Theoretical Computer Science*, 367(1):57–87, 2006. 5

[BCKL08] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. P-signatures and noninteractive anonymous credentials. In *Proc. 5nd Theory of Cryptography Conference (TCC)*, pages 356–374, 2008. 7

[BCKL09] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. Compact e-cash and simulatable VRFs revisited. In *Proc. 3rd International Conference on Pairing-Based Cryptography (Pairing)*, pages 114–131, 2009. 7

[BFG10] Karthikeyan Bhargavan, Cédric Fournet, and Andrew 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. 5, 10, 82, 84, 86, 87

[BFGT08] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1), 2008. 10, 86

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *Proc. 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 103–112, 1988. 6

[BFM07] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563–617, 2007. 14

[BGHL10] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an SMT solver. In *15th ACM SIGPLAN International Conference on Functional programming (ICFP 2010)*, pages 105–116. ACM Press, September 2010. 119

[BGHM09] Michael Backes, Martin P. Grochulla, Cătălin Hriţcu, and Matteo Maffei. Achieving security despite compromise using zero-knowledge. In *22th IEEE Symposium on Computer Security Foundations (CSF 2009)*. IEEE Computer Society Press, July 2009. 7, 8, 11, 13, 14, 15, 52, 60, 68, 71, 72, 101

[BGP11] Johannes Borgström, Andrew D. Gordon, and Riccardo Pucella. Roles, stacks, histories: A triple for hoare. *Journal of Functional Programming*, 21(02):159–207, 2011. 86

[BHB+10] Gilles Barthe, Daniel Hedin, Santiago Zanella Béguelin, Benjamin Grégoire, and Sylvain Heraud. A machine-checked formalization of sigma-protocols. In *Proc. 23th IEEE Symposium on Computer Security Foundations (CSF)*, pages 246–260. IEEE Computer Society Press, 2010. 15

[BHM08a] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, pages 195–209. IEEE Computer Society Press, 2008. 14

[BHM08b] Michael Backes, Cătălin Hriţcu, and Matteo 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. 120

[BHM08c] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Type-checking zero-knowledge. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 357–370. ACM Press, 2008. 11, 13, 60, 83, 108, 112, 121, 145, 148

[BHU09] Michael Backes, Dennis Hofheinz, and Dominique Unruh. CoSP: A general framework for computational soundness proofs. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS)*, pages 66–78, 2009. 85, 86

[Bla01] Bruno 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. 9, 14, 26, 86

[Ble98] Daniel 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. 5

[BLMP10] Michael Backes, Stefan Lorenz, Matteo Maffei, and Kim 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. 7, 23, 52, 120, 122

[BM11] Michael Backes and Esfandiar Mohammadi. Computational soundness of malleable zero-knowledge proofs (abstract). 7th Workshop on Formal and Computational Cryptography (FCC), 2011. 15, 16

[BMM10] Michael Backes, Matteo Maffei, and Esfandiar Mohammadi. Computationally sound abstraction and verification of secure multi-party computations. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8, pages 352–363, 2010. 15

[BMP11] Michael Backes, Matteo Maffei, and Kim Pecina. A security API for distributed social networks. In *18th Annual Network & Distributed System Security Symposium (NDSS'11)*, pages 35–51. Internet Society, 2011. 7, 122

[BMU08] Michael Backes, Matteo Maffei, and Dominique 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. 6, 14, 15, 16, 22, 26, 112, 145

[BMU10] Michael Backes, Matteo Maffei, and Dominique Unruh. Computationally sound verification of source code. In *Proc. 17th ACM Conference on Computer and Communications Security (CCS)*, pages 387–398. ACM Press, 2010. 85, 109

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993. 6

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986. 151

[BSS10] John Bethencourt, Elaine Shi, and Dawn Song. Signatures of reputation. In *14th International Conference on Financial Cryptography and Data Security*, pages 400–407, 2010. 7

[BSSM+07] Abhilasha Bhargav-Spantzel, Anna Cinzia Squicciarini, Shimon K. Modi, Matthew Young, Elisa Bertino, and Stephen J. Elliott. Privacy preserving multi-factor authentication with biometrics. *Journal of Computer Security*, 15(5):529–560, 2007. 7

[BU08] Michael Backes and Dominique Unruh. Computational soundness of symbolic zero-knowledge proofs against active attackers. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, pages 255–269. IEEE Computer Society Press, 2008. 15

[Bus11] Alex Busenius. Mechanized formalization of a transformation from an extensible spi calculus to Java. Master's thesis, Saarland University, April 2011. 24

[Car97] Luca Cardelli. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997. 99

[CCM08] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: A secure voting system. In *Proc. 29th IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society Press, 2008. 6, 10, 84, 85

[CD08] Sagar Chaki and Anupam Datta. ASPIER: An automated framework for verifying security protocol implementations. Technical report, CMU CyLab, October 2008. 86

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology - CRYPTO 1994*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer-Verlag, 1994. 6, 15, 122

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, February 1988. 121

[CH02] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *Proc. 9th ACM Conference on Computer and Communications Security*, pages 21–30, 2002. 7, 120

[Cha83] David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology: CRYPTO'82*, pages 199–203, 1983. 75, 123

[CHK+06] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clonewars: efficient periodic n-times anonymous authentication. In *Proc. 13th ACM Conference on Computer and Communications Security*, pages 201–210, 2006. 7

[CHL06] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Balancing accountability and privacy using e-cash (extended abstract). In *Proc. 5th International Conference on Security and Cryptography for Networks (SCN)*, pages 141–155, 2006. 7

[CJS+08] Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, Joe-Kai Tsay, and Christopher Walstad. Breaking and fixing public-key Kerberos. *Information and Computation*, 206:402–424, February 2008. 5

[CL01] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Advances in Cryptology - EUROCRYPT 2001*, pages 93–118, 2001. 15

[CL02] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In *Proc. 3rd International Conference on Security in Communication Networks (SCN)*, volume 2576 of *Lecture Notes in Computer Science*, pages 268–289. Springer-Verlag, 2002. 6, 122

[CL04] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *Advances in Cryptology - CRYPTO 2004*, pages 56–72, 2004. 7

[CLM07] Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich. Endorsed e-cash. In *Proc. 28th IEEE Symposium on Security & Privacy*, pages 101–115, 2007. 7

[CM11] Ricardo Corin and Felipe Andrés 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, 2011. 86

[CMS10] Jan Camenisch, Sebastian Mödersheim, and Dieter Sommer. A formal model of identity mixer. In *Proc. 5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, pages 198–214. Springer-Verlag, 2010. 15

[Com97] Adriana B. Compagnoni. Subject reduction and minimal types for higher order subtyping. Technical Report ECS-LFCS-97-363, LFCS, University of Edinburgh, August 1997. 116

[Coq09] The Coq proof assistant, 2009. Version 8.2. 105, 107

[CPM90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In *Proceedings of the International Conference on Computer Logic (COLOG-88)*, pages 50–66. Springer-Verlag, 1990. 121

[Cra96] Ronald Cramer. *Modular Design of Secure yet Practical Cryptographic Protocols*. PhD thesis, CWI and University of Amsterdam, 1996. 23

[dB72] 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. 61, 105

[DGJN11] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *Proc. 24th IEEE Symposium on Computer Security Foundations (CSF)*, 2011. To appear. 86

[DGS03] Ivan Damgåard, Jens Groth, and Gorm Salomonsen. The theory and implementation of an electronic voting system. In *Proc. Secure Electronic Voting (SEC)*, pages 77–100, 2003. 6

[DKR09] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009. 14, 120

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of TACAS*, 2008. 14, 29, 80, 115

[DP00] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proc. International Conference on Functional Programming (ICFP 2000)*, pages 198–208, 2000. 99, 102, 116

[DP04] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *Proc. 31th Symposium on Principles of Programming Languages (POPL '04)*, pages 281–292. ACM Press, 2004. 103, 116

[dRP11] Joeri de Ruiter and Erik Poll. Formal analysis of the EMV protocol suite. In *Theory of Security and Applications (TOSCA 2011)*. Springer-Verlag, 2011. To appear. 86

[DRS08] Stéphanie Delaune, Mark Ryan, and Benn Smyth. Automatic verification of privacy properties in the applied pi calculus. To appear in 2nd Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM'08), 2008. 14

[DS81] Dorothy E. Denning and Giovanni M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981. 5

[Dun09] Joshua Dunfield. Greedy bidirectional polymorphism. In *ML Workshop (ML '09)*, pages 15–26, August 2009. 99

[Dun10] Joshua Dunfield. Untangling typechecking of intersections and unions. In *Workshop on Intersection Types and Related Systems (ITRS)*, July 2010. 117

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. 15, 21

[Eig09] Fabienne Eigner. Type-based verification of electronic voting systems. Master's thesis, Saarland University, 2009. 84, 85

[FCB08] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4), 2008. 119

[FGM07a] Cédric Fournet, Andrew D. Gordon, and Sergio 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. 5, 7, 8, 11, 13, 14, 15, 20, 24, 25, 27, 29, 30, 32, 37, 44, 48, 60, 68, 69, 71, 84, 88, 120

[FGM07b] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), 2007. 5, 14

[Fis03] Dennis Fisher. Millions of .Net Passport accounts put at risk. *eWeek*, May 2003. (Flaw detected by Muhammad Faisal Rauf Danka). 5

[FKS11] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *Proc. 18th ACM Conference on Computer and Communications Security (CCS)*, pages 341–350. ACM Press, 2011. 85, 120

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *In Programming Language Design and Implementation (PLDI'91)*, pages 268–277. ACM Press, 1991. 92, 93

[GA08] Deepak Garg and Martín Abadi. A modal deconstruction of access control logics. In *Proc. 11th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, pages 216–230. Springer-Verlag, 2008. 120

[Gar09] Deepak Garg. Proof search in an authorization logic. Technical report, Carnegie Mellon University, School of Computer Science, 2009. CMU-CS-09-121. 120

[Gir86] Jean-Yves Girard. The System F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986. 92, 93, 94, 95

[GJ03] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 4(11):451–521, 2003. 5, 18, 29, 30, 35, 84

[GJ04] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3):435–484, 2004. 14, 29, 30, 32, 84

[GJ05] Andrew D. Gordon and Alan 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. 14, 119

[GLP05] Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2005)*, pages 363–379. Springer, 2005. 85

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):690–728, 1991. Online available at `http://www.wisdom.weizmann.ac.il/~oded/X/gmw1j.pdf`. 6

[Gol01] Oded Goldreich. *Foundations of Cryptography: Basic Tools.* Cambridge University Press, 2001. 6

[Gor93] Andrew 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. 61, 105

[Got11] Eli Gottlieb. Simple, decidable type inference with subtyping. *CoRR*, abs/1104.3116, 2011. 80

[GP06] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 283–296. IEEE Computer Society Press, 2006. 120

[Gro05] Jens Groth. Non-interactive zero-knowledge arguments for voting. In *Proc. 3rd International Conference on Applied Cryptography and Network Security (ACNS)*, pages 467–482, 2005. 6

[Gro09] Martin P. Grochulla. Security despite system compromise with zero-knowledge proofs. Master's thesis, Saarland University, 2009. 8, 14, 15, 68, 72

[GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In *Advances in Cryptology - EUROCRYPT 2008*, pages 415–432, 2008. 6, 122

[GT08] Deepak Garg and Michael Carl Tschantz. From indexed lax logic to intuitionistic logic. Technical Report CMU-CS-07-167, Carnegie Mellon University, School of Computer Science, January 2008. 120

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992. 92

[HBCDF06] Thomas S. Heydt-Benjamin, Hee-Jin Chae, Benessa Defend, and Kevin Fu. Privacy for public transportation. In *6th International Workshop on Privacy Enhancing Technologies (PETS)*, pages 1–19, 2006. 7

[Hig08] Eclipse Higgins: Open source identity framework, 2008. `http://www.eclipse.org/higgins/`. 7

[HJ05] Christian Haack and Alan Jeffrey. Timed spi-calculus with types for secrecy and authenticity. In *Proc. 16th International Conference on Concurrency Theory (CONCUR)*, volume 3653, pages 202–216. Springer-Verlag, 2005. 14

[HJ06] Christian Haack and Alan Jeffrey. Pattern-matching spi-calculus. *Information and Computation*, 204(8):1195–1263, 2006. 14

[HL] Bob Harper and Mark 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`. 116

[HRT10] James Heather, Peter Y. A. Ryan, and Vanessa Teague. Pretty good democracy for more expressive voting schemes. In *Proc. 15th European Symposium on Research in Computer Security (ESORICS)*, pages 405–423. Springer-Verlag, 2010. 6

[HS00] Ullrich Hustadt and Renate A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 67–71. Springer-Verlag, 2000. 120

[HSW99] Ullrich Hustadt, Renate A. Schmidt, and Christoph Weidenbach. MSPASS: Subsumption testing with SPASS. In *Proc. of the 1999 International Workshop on Description Logics (DL'99)*, 1999. 120

[HVP05] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for xml. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, 2005. 119

[IBM] IBM identity governance project. `http://www.zurich.ibm.com/security/idemix/`. 6, 7

[JMR11] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *Proceedings of CAV*, pages 470–485, 2011. 121

[KNON10] Hiroaki Kikuchi, Kei Nagai, Wakaha Ogata, and Masakatsu Nishigaki. Privacy-preserving similarity evaluation and application to remote biometrics authentication. *Soft Computing*, 14(5):529–536, 2010. 7

[Kob09] Naoki 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. 121

[KTU90] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proc. 22nd ACM Symposium on Theory of Computing (STOC '90)*, pages 468–476. ACM, 1990. 80

[LAN02] Helger Lipmaa, N. Asokan, and Valtteri Niemi. Secure vickrey auctions without threshold trust. In *6th International Conference on Financial Cryptography (FC)*, pages 87–101, 2002. 7

[LHH+07] Li Lu, Jinsong Han, Lei Hu, Jinpeng Huai, Yunhao Liu, and Lionel 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. 7, 19, 52, 113, 147

[LNBH11] Jay Ligatti, Michael Nachtigal, Jeremy Blackburn, and Ivory Hernandez. Completely subtyping iso-recursive types. Unpublished Draft, 2011. 100

[Low96] Gavin 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. 5, 10, 87, 91

[LW94] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16:1811–1841, 1994. 98

[MEK+10] Sarah Meiklejohn, C. Christopher Erway, Alptekin Küpçü, Theodora Hinkle, and Anna Lysyanskaya. ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash. In *Proc. 19th USENIX Security Symposium*, pages 193–206, 2010. 122

[Men91] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159–172, 1991. 100

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992. 86

[Mil99] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. 92, 109

[Moh09] Esfandiar Mohammadi. Computational soundness for symbolic zero-knowledge proofs against active attackers under relaxed assumptions. Master's thesis, Saarland University, 2009. 15

[Mor73] James H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973. 10, 82, 109, 120

[MP09] Sean McLaughlin and Frank Pfenning. Efficient intuitionistic theorem proving with the polarized inverse method. In *Automated Deduction – CADE-22: 22st International Conference on Automated Deduction*, pages 230–244. Springer-Verlag, 2009. 120

[MP11] Matteo Maffei and Kim Pecina. Privacy-aware proof-carrying authorization. Position Paper, PLAS 2011, to appear, April 2011. 23, 85, 120, 122

[OSV10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, Second Edition.* Artima, 2010. 40

[Pie91] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991. 24, 31, 92, 94, 95, 99, 102, 103, 116

[Pie97] Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997. 31, 92, 95, 102, 116

[PPM90] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Proc. 5th International Conference on Mathematical Foundations of Programming Semantics*, pages 209–226. Springer-Verlag, 1990. 121

[PRST08] David C. Parkes, Michael O. Rabin, Stuart M. Shieber, and Christopher Thorpe. Practical secrecy-preserving, verifiably correct and trustworthy auctions. *Electronic Commerce Research and Applications*, 7(3):294–312, 2008. 7

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000. 80

[RD10] Alfredo Rial and George Danezis. Privacy-preserving smart metering. Technical Report MSR-TR-2010-150, Microsoft Research, November 2010. `http://research.microsoft.com/apps/pubs/?id=141726`. 7

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983. 92, 93, 94, 95

[Rey96] John 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. 102, 116

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 159–169, 2008. 121

[ROK07] Thomas Raths, Jens Otten, and Christoph Kreitz. The ILTP problem library for intuitionistic logic. *Journal of Automated Reasoning*, 38(1-3):261–271, 2007. 120

[ROS98] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998. 92, 93

[RT06] Silvio Ranise and Cesare Tinelli. The SMT-LIB standard: Version 1.2, August 2006. 80

[RV99] Alexandre Riazanov and Andrei Voronkov. Vampire. In *Proc. 16th International Conference on Automated Deduction (CADE)*, pages 292–296, 1999. 14, 15, 29, 80

[SB08a] Gert Smolka and Chad 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`. 151

[SB08b] Christoph Sprenger and David A. Basin. Cryptographically-sound protocol-model abstractions. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, pages 115–129. IEEE Computer Society Press, 2008. 16

[SC08] Daniel Smith and Robert Cartwright. Java type inference is broken: can we fix it? In *Proc. 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*, pages 505–524. ACM, 2008. 80

[SCC10] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in Fine. In *Proc. 19th European Symposium on Programming (ESOP)*. Springer-Verlag, 2010. 86

[SCF+11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bharagavan, and Jean Yang. Secure distributed programming with value-dependent types. Technical Report MSR-TR-2011-37, Microsoft Research, March 2011. Accepted at ICFP 2011. 87

[Sch02] Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002. 14, 29, 80

[SNO+10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *The Journal of Functional Programming*, 20(1):71–122, 2010. 61, 108

[SP03] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4):521–554, 2003. 109, 120

[SP07] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375(1-3):169–192, 2007. 10, 82, 109, 120

[Sut09] 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. 80

[Szy05] Michael Szydlo. Risk assurance for hedge funds using zero knowledge proofs. In *9th International Conference on Financial Cryptography and Data Security (FC)*, pages 156–171, 2005. 7

[Tam96] Tanel Tammet. A resolution theorem prover for intuitonistic logic. In *Automated Deduction – CADE-13: 13st International Conference on Automated Deduction*, pages 2–16. Springer-Verlag, 1996. 120

[Tar08] Thorsten Tarrach. Spi2F# – A prototype code generator for security protocols. Bachelor's Thesis, Saarland University, 2008. 15

[TCG11] TPM Main. Specification Version 1.2, Revision 116, Trusted Computing Group Published, March 2011. Available at `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`. 16

[TCGS89] Val Tannen, Thierry Coquand, Carl A. Gunter, and Andre 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. 99

[TGS89] Val Tannen, Carl A. Gunter, and Andre Scedrov. Denotational semantics for subtyping between recursive types. Technical Report MS-CIS-89-63, University of Pennsylvania, Department of Computer & Information Science, November 1989. 100

[TSGW09] Amin Tootoonchian, Stefan Saroiu, Yashar Ganjali, and Alec Wolman. Lockr: better privacy for social networks. In *Proc. Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 169–180. ACM Press, 2009. 7

[UPr11] Microsoft U-Prove, Community Technology Preview R2, February 2011. `http://www.microsoft.com/u-prove`. 6, 7

[Urz95] Pawel Urzyczyn. Positive recursive type assignment. In *Proc. 20th International Symposium Mathematical Foundations of Computer Science (MFCS'95)*, pages 382–391, 1995. 100

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In *Automated Deduction – CADE-22 : 22nd International Conference on Automated Deduction*, pages 140–145, 2009. 14, 29, 80

[WL94] Thomas Y. C. Woo and Simon S. Lam. A lesson on authentication protocol design. *Operation Systems Review*, 28(3):24–37, 1994. 5

[WS96] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, 1996. 5

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th Symposium on Principles of Programming Languages (POPL '99)*, pages 214–227. ACM Press, 1999. 92, 93

[Zei08] Noam Zeilberger. Refinement types and computational duality. In *Proc. 3rd Workshop on Programming Languages Meets Program Verification (PLPV)*, pages 15–26. ACM Press, 2008. 117