Type-checking Implementations of Protocols Based on Zero-knowledge Proofs – Work in Progress –

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

[1]Saarland University, Saarbrücken, Germany [2]MPI-SWS

Abstract. We present the first static analysis technique for verifying implementations of cryptographic protocols based on zero-knowledge proofs. Protocols are implemented in $\text{RCF}^{\forall}_{\wedge\vee}$, a core calculus of ML with support for concurrency. Cryptographic primitives are considered as fully reliable building blocks and represented symbolically using a sealing mechanism. Zero-knowledge proofs, in particular, are specified in a high-level language and automatically compiled down to a symbolic implementation using seals. Security properties are formalized as authorization policies and statically enforced by a type system featuring refinement, union, intersection, and polymorphic types. The expressiveness of our type system allows us to analyze not only protocols based on zero-knowledge proofs but also other important protocol classes that were out of the scope of existing static analysis techniques.

1 Introduction

Proofs of security protocols are known to be error-prone and awkward to make for humans. In fact, vulnerabilities have accompanied the design of such protocols ever since early authentication protocols like Needham-Schroeder, over carefully designed de-facto standards like SSL and PKCS, up to current widely deployed products like Microsoft Passport and Kerberos. Hence work towards the automation of such proofs has started soon after the first protocols were developed, focusing on high-level protocol descriptions that are suited to the formalization of security properties and to the automation of security proofs.

Despite this considerable progress on protocol analysis, the verification of security protocol implementations is still a widely unexplored field. The aforementioned high-level protocol descriptions abstract away from most troublesome implementation details, and there is no guarantee that a protocol that has been proven secure within an abstract model stays secure when implemented.

Devising automated techniques for proving the security of protocol implementations is a highly non-trivial task. First, modern applications such as trusted computing [12] and electronic voting [14] rely on *complex cryptographic primitives*, such as zero-knowledge proofs. Automated verification of such applications requires to symbolically abstract these primitives. Process calculi provide convenient mechanisms to define such abstractions, for instance flexible equational theories [3,2], which rendered an automated analysis of such applications feasible in abstract models [6,4]. This is not the case for standard programming languages,

where one needs to encode these abstractions using the primitives provided by the language. These primitives, however, were not designed for this purpose, which makes providing encodings that are suitable for automatic analysis a difficult task.

Second, high-level protocol specifications are typically compact, since many implementation details are abstracted away, while protocol implementations are much larger. Therefore *efficiency* and *scalability* of the proposed techniques are (even more) crucial when dealing with industrial-size applications.

Third, conducting *security proofs* within abstract protocol models is known to be a difficult task. Providing security proofs for protocol implementations is typically even more difficult since they require extensive reasoning about the behavior of programs in the presence of features such as data structures, recursion, and state.

1.1 Contributions

We present the first static analysis technique for verifying implementations of security protocols based on non-interactive zero-knowledge proofs. Protocols are implemented in RCF [7], a core calculus of ML, and cryptographic primitives are considered fully reliable building blocks and represented symbolically using a sealing mechanism [16,19]. The zero-knowledge proofs are specified in a high-level language and automatically compiled down to a symbolic implementation. The verification of the resulting code relies on a type system with union, intersection, and refinement types. This expressive type system extends the scope of type-based analyses of protocol implementations to important protocol classes not covered so far. In particular, we use union and intersection types to precisely characterize asymmetric cryptography, which allows us to verify protocols based on nested cryptography, signatures of private data, and public-key encryption of authenticated data. The analysis is automated and modular, and was efficient on the examples that we tried.

1.2 Related Work

Our type system builds on the work by Bengtson et al. on the type-based analysis of protocol implementations in F# [7]. Their type system supports most of the features of F# by translation to RCF. The analysis is modular and promises to scale up to large implementations [8]. Their type system, however, is not expressive enough to deal with zero-knowledge proofs and poses some serious restrictions even on the usage of standard cryptographic primitives. For instance, if a key is used to sign a secret message, then the corresponding verification key cannot be made public. These limitations prevent the analysis of many interesting cryptographic applications, such the Direct Anonymous Attestation protocol [12], which relies on zero-knowledge proofs as well as on digital signatures on secret TPM identifiers. We extend this type system with union and intersection types, which significantly increases the expressiveness of the analysis and enables the verification of protocols based on zero-knowledge proofs.

Goubault-Larrecq and Parrennes developed a static analysis technique [15] based on pointer analysis and clause resolution for cryptographic protocols implemented in C. The analysis provides security proofs and is also useful to find bugs in the implementation, but it is limited to secrecy properties, it deals only with standard cryptographic primitives, and it does not offer scalability since the number of generated clauses is very high even on small protocol examples.

Chaki and Datta have recently proposed a technique [13] based on software model checking for the automated verification of secrecy and authentication properties of protocols implemented in C. The analysis provides security proofs for a bounded number of sessions and is effective in discovering attacks. It was used to check secrecy and authentication properties of the SSL handshake protocol for configurations of up to three servers and three clients. The analysis only deals with standard cryptographic primitives, and offers only limited scalability.

Bhargavan et al. proposed a technique [10,9] for the verification of F# protocol implementations by automatically extracting ProVerif models [11]. The analysis provides security proofs and, despite its non-compositional nature, scales remarkably well and was successfully used to verify implementations of real-world cryptographic protocols such as TLS [9]. The considered fragment of F# is, however, very restrictive: it does not include higher-order functions, and it allows only for a very limited usage of recursion and state. These restrictions impose serious limitations on the programming style, and as a consequence the technique can only be used to analyze protocol implementations developed from scratch.

1.3 Outline

Section 2 introduces the calculus and Section 3 the type system we consider. Section 4 illustrates our symbolic implementation of asymmetric cryptography. Section 5 discusses our encoding of zero-knowledge proofs in $\mathrm{RCF}^{\forall}_{\wedge\vee}$ and applies it to a simplified version of the DAA protocol. Section 6 describes our implementation. Section 7 concludes and discusses future work. Due to space constraints, we defer many technical details to an extended version of the paper [5].

2 $\mathrm{RCF}^{\forall}_{\wedge\vee}$

The Refined Concurrent FPC (RCF) [7] is a simple programming language extending the Fixed Point Calculus with refinement types and concurrency. Although very simple, this core calculus is expressive enough to encode a considerable fragment of F#. In this paper, we further increase the expressivity of the calculus by adding intersection types [17], union types, and parametric polymorphism. We call the extended calculus $\mathrm{RCF}^{\forall}_{\wedge\vee}$ and describe it in this and in the following section.

2.1 Syntax and Informal Semantics

The set of *values* is composed of variables, the unit value, functions, pairs, and introduction forms for recursive and polymorphic types (cf. Table 1). Names are

Table 1 Syntax of $\text{RCF}^{\forall}_{\wedge\vee}$ values and expressions

a, b, c	name	$A, B ::=$	expression
x, y, z	variable	M	value
		$M\ N$	function application
$M, N ::=$	value	$M\langle T\rangle$	type instantiation
x	variable	$\mathsf{if}\ M = N\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$	equality check with type-cast
$()$	unit	$\mathsf{let}\ x = A\ \mathsf{in}\ B$	let
$\lambda x : T.\, A$	function	$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ A$	pair split
(M, N)	pair	$\mathsf{unfold}_{\mu\alpha.\,T}\ M$	use recursive value
$\Lambda\alpha.\, A$	polymorphic val.	$\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ A$	introduction of intersections
$\mathsf{fold}_{\mu\alpha.\,T}\ M$	recursive val.	$\mathsf{case}\ x = M : T \vee U\ \mathsf{in}\ A$	elimination of union types
$\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ M,$		$(\nu a \updownarrow T)A$	restriction
where $\{\widetilde{\alpha}\} \cap \mathit{free}(M) \neq \emptyset$	intersection val.	$A \Rsh B$	fork
		$a!M$	send M on channel a
		$a?$	receive message from a
		$\mathsf{assume}\ C$	assumption of formula C
		$\mathsf{assert}\ C$	assertion of formula C

Notation: Given a phrase of syntax ϕ, let $\phi\{M/x\}$ denote the substitution of each free occurrence of the variable x in ϕ with the value M. We use $\widetilde{\phi}$ to denote the sequence $\phi_1, \ldots, \phi_n$ for some n.

generated at run-time and are only used as channel identifiers, while variables are place-holders for values.

An *expression* represents a concurrent computation that may reduce to a value (or may diverge, get "stuck" or deadlock). The *reduction relation* is defined on *computation states* consisting of several expressions being evaluated in parallel; the messages sent on channels but not yet received; and a global log containing assumed formulas[1]. Values are irreducible. The function application $(\lambda x{:}T.\, A)\ M$ reduces to $A\{M/x\}$. A type instantiation $(\Lambda\alpha.\, A)\langle T\rangle$ reduces to $A\{T/\alpha\}$. The conditional $\mathsf{if}\ M = N\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$ reduces to $A\{M/x\}$ if M is syntactically equal to N, and to B otherwise (more details are given in Section 3.4). The introduction form for intersection types $\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ A$ allows A to reduce to a value M, and if M does not contain $\widetilde{\alpha}$ then returns M, otherwise it returns the value $\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ M$. The elimination form for union types $\mathsf{case}\ x = M : T \vee U\ \mathsf{in}\ A$ reduces to $A\{M/x\}$ (more details about intersection and union types are given in Section 3). Intuitively, the restriction $(\nu a \updownarrow T)A$ generates a globally fresh channel a that can only be used in A to convey values of type T. The expression $A \Rsh B$ evaluates A and B in parallel, and returns the result of B (the result of A is discarded). The expression $a!M$ outputs M on channel a and reduces to the unit value $()$. The evaluation of $a?$ blocks until some message M is available on channel a, removes M from the channel, and then returns M. Expression $\mathsf{assume}\ C$, where C is a logical formula, adds C to the global log. The assertion $\mathsf{assert}\ C$ reduces to $()$. If C is entailed by the multiset S of formulas in the global log, we say the assertion *succeeds*; otherwise, we say the assertion *fails*.

[1] We consider first-order logic with equality as the authorization logic.

Table 2 A direct implementation of a sign-then-encrypt protocol in $\mathrm{RCF}^{\forall}_{\wedge\vee}$

$$
\begin{array}{l}
(\nu c \updownarrow T_{\text{chan}}) \\
\mathsf{let}\ y_b = \mathit{mkId}() \ \mathsf{in} \\
\mathsf{let}\ sk = \mathit{mkSK}\,\langle T_{\text{sign}}\rangle\ ()\ \mathsf{in}\ \mathsf{let}\ vk = \mathit{mkVK}\,\langle T_{\text{sign}}\rangle\ sk\ \mathsf{in} \\
\mathsf{let}\ dk = \mathit{mkDK}\,\langle T_{\text{enc}}\rangle\ ()\ \mathsf{in}\ \mathsf{let}\ ek = \mathit{mkEK}\,\langle T_{\text{enc}}\rangle\ dk\ \mathsf{in} \\
(\quad \mathsf{let}\ y = \mathit{getPriv}\ ()\ \mathsf{in} \\
\quad\ \mathsf{assume}\ Ok(y); \\
\quad\ \mathsf{let}\ x = (\mathit{sign}\,\langle T_{\text{sign}}\rangle\ \ sk)\ (y_b, y)\ \mathsf{in} \\
\quad\ \mathsf{let}\ z = (\mathit{encrypt}\,\langle T_{\text{enc}}\rangle\ \ ek)\ (x, (y_b, y))\ \mathsf{in} \\
\quad\ c!z)\ \vec{\cdot} \\
(\quad \mathsf{let}\ z = c?\ \mathsf{in} \\
\quad\ \mathsf{let}\ xy = (\mathit{decrypt}\,\langle T_{\text{enc}}\rangle\ \ dk)\ z\ \mathsf{in} \\
\quad\ \mathsf{let}\ (x, y) = xy\ \mathsf{in} \\
\quad\ \mathsf{let}\ y' = (\mathit{check}\,\langle T_{\text{sign}}\rangle\ vk\ x\ y)\ \mathsf{in} \\
\quad\ \mathsf{let}\ (y_1, y_2) = y'\ \mathsf{in} \\
\quad\ \mathsf{if}\ y_1 = y_b\ \mathsf{then}\ \mathsf{assert}\ Ok(y_2)\ )
\end{array}
$$

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

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*.

Definition 2 (Opponent and Robust Safety). *A closed expression O is an* opponent *if and only if O contains no assertions and O only contains* Un *for the type annotations. A closed expression A is* robustly safe *if and only if the application $O\ A$ is safe for all opponents O.*

2.2 Example: Sign-then-Encrypt

We illustrate the calculus, and later the type system, on the following very simple protocol, in which A first signs and then encrypts a private message for B:

$$A \longrightarrow \left\{\left|[(B,m)]_{k_A^-}\ ,\ (B,m)\right|\right\}_{k_B^+} \longrightarrow B$$

The $\mathrm{RCF}^{\forall}_{\wedge\vee}$ implementation of this protocol is reported in Table 2. We first generate a fresh public channel c and B's identifier y_b. We then create the signing and encryption key-pairs. The sender gets a private value y from the user and assumes the predicate $Ok(y)$. The sender then signs this message together with the receiver's identifier y_b, encrypts this signature together with the signed pair (y_b, y), and outputs the resulting ciphertext on channel c.

The receiver reads the message from channel c, decrypts it, splits the pair, and checks the signature. If all these checks succeed and additionally the second component of the pair is his own identifier, the receiver asserts $Ok(y_2)$, where y_2 is bound to the private value sent by A.

Table 3 Syntax of Types

$T, U, V ::= \top$		top type	$x : T \to U$	dependent function type
	Un	untrusted type	$x : T * U$	dependent pair type
	$\mathsf{Private}$	private type	α	type variable
	$T \wedge U$	intersection type	$\mu\alpha.\, T$	iso-recursive type
	$T \vee U$	union type	$\forall\alpha.\, T$	polymorphic type
	$\{x : T \mid C\}$	refinement type	unit	unit type

Notations:
Let $T \to U \triangleq x : T \to U$ and $T * U \triangleq x : T * U$, where in both cases $x \notin \mathit{free}(U)$. Let $\{C\}$ denote $\{x : \mathsf{unit} \mid C\}$ where $x \notin \mathit{free}(C)$.

3 Type System

This section presents a type system for enforcing authorization policies on $\text{RCF}^{\forall}_{\wedge\vee}$ code. This extends the type system proposed by Bengtson et al. [7] with union types, intersection types [17] and unrestricted parametric polymorphism. This extension enhances the expressivity of the analysis and, in particular, allows us to obtain a more precise characterization of asymmetric cryptography and to provide a faithful encoding of zero-knowledge proofs.

3.1 Types

The syntax of types is reported in Table 3. Our type system has a top type $\top$ that is supertype of all the others (the subtyping relation is introduced in Section 3.3). 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. As in [7], we use refinement types to associate logical formulas to messages. The refinement type $\{x : T \mid C\}$ describes values M of type T for which the formula $C\{M/x\}$ is entailed by the current typing environment. 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. In the following we explain the typing judgements and present the most important typing rules.

3.2 Typing Environment and Entailment

A typing environment E is a list of bindings for type variables (α), names ($a \updownarrow T$, where a stands for a channel conveying values of type T), and variables ($x : T$).

A crucial judgment in the type system is $E \vdash C$, which states that the formula C is derivable from 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 is used for instance when type-checking $\mathsf{assert}\ C$: 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 $E \vdash C\{y/x\}$.

3.3 Subtyping and Kinding

To type-check programs interacting with the attacker, we consider a universal type Un (untrusted), which is the type of data possibly known to the attacker. For instance, all data sent to and received from an untrusted channel have type Un, since such channels are considered under the complete control of the adversary.

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

Refinement types. The refinement type $\{x : T \mid C\}$ is a subtype of T. This allows us to discard logical formulas when they are not needed. For instance, a value of type $\{x : \mathsf{Un} \mid Ok(x)\}$ can be sent on a channel of type Un. Conversely, the type T is a subtype of $\{x : T \mid C\}$ only if $\forall x.C$ is entailed in the current typing environment. By the transitivity of subtyping, a refinement type $\{x : T \mid C\}$ is public if T is public, since then $\{x : T \mid C\} <: T <: \mathsf{Un}$. Conversely, type $\{x : T \mid C\}$ is tainted if T is tainted and additionally $\forall x.C$ holds.

Union and Intersection Types. The rules for subtyping and kinding union and intersection types are reported in Table 4. The type $T_1 \wedge T_2$ is a subtype of U if T_1 or T_2 is a subtype of U (cf. SUB-AND-LB), while T is a subtype of $U_1 \wedge U_2$ if it is subtype of both U_1 and U_2 (cf. SUB-AND-GREATEST). Intuitively, the set of values of type $T_1 \wedge T_2$ is the intersection between the set of values of type T_1 and the set of values of type T_2. The type $T_1 \wedge T_2$ is public if T_1 or T_2 is public, and it is tainted if both T_1 and T_2 are tainted. The rules for union types are dual.

3.4 Typing Values and Expressions

The main judgment of the type system we consider is $E \vdash A : T$, which states that the expression A returns a value of type T. The most important typing rules are reported in Table 5. Most of them are standard, so we focus the explanation only on the rules that are new with respect to [7].

Conditionals. The rule EXP IF exploits intersection types for strengthening the type of the values tested for equality in the conditional $\mathsf{if}\ M = N\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$.

Table 4 Kinding and Subtyping Unions and Intersections

Subtyping

$$\textsc{Sub And LB}\ \frac{\Gamma \vdash T_i <: U}{\Gamma \vdash T_1 \wedge T_2 <: U} \qquad \textsc{Sub And Greatest}\ \frac{\Gamma \vdash T <: U_1 \qquad \Gamma \vdash T <: U_2}{\Gamma \vdash T <: U_1 \wedge U_2}$$

$$\textsc{Sub Or Smallest}\ \frac{\Gamma \vdash T_1 <: U \qquad \Gamma \vdash T_2 <: U}{\Gamma \vdash T_1 \vee T_2 <: U} \qquad \textsc{Sub Or UB}\ \frac{\Gamma \vdash T <: U_i}{\Gamma \vdash T <: U_1 \vee U_2}$$

Kinding

$$\textsc{Kind And Pub}\ \frac{\Gamma \vdash T_i :: \mathsf{pub}}{\Gamma \vdash T_1 \wedge T_2 :: \mathsf{pub}} \qquad \textsc{Kind And Tnt}\ \frac{\Gamma \vdash T_1 :: \mathsf{tnt} \qquad \Gamma \vdash T_2 :: \mathsf{tnt}}{\Gamma \vdash T_1 \wedge T_2 :: \mathsf{tnt}}$$

$$\textsc{Kind Or Pub}\ \frac{\Gamma \vdash T_1 :: \mathsf{pub} \qquad \Gamma \vdash T_2 :: \mathsf{pub}}{\Gamma \vdash T_1 \vee T_2 :: \mathsf{pub}} \qquad \textsc{Kind Or Tnt}\ \frac{\Gamma \vdash T_i :: \mathsf{tnt}}{\Gamma \vdash T_1 \vee T_2 :: \mathsf{tnt}}$$

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

Union Types are introduced by subtyping (T_1 is a subtype $T_1 \vee T_2$ for any T_2), and eliminated using the Exp Case rule. Suppose that M is of type $T_1 \vee T_2$ and M is used in A. Intuitively, since we do not know whether M is of type T_1 or T_2, we have to type-check A under each of the assumptions. Therefore, the expression $\mathsf{case}\ x = M : T_1 \vee T_2\ \mathsf{in}\ A$, where x takes the place of M in A, is of type U only if A is of type U both when x is of type T_1 and when x is of type T_2. This is useful when type-checking code interacting with the attacker. For instance, suppose that a party receives a value M encrypted with a public-key that is used by honest parties to encrypt messages of type T. After decryption, M 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 receiver's code twice, once under the assumption that x is of type T, and once assuming that x is of type Un.

Table 5 Selected Rules for Values and Expressions $E \vdash A : T$

$$\textsc{Val Refine}\ \frac{E \vdash M : T \qquad E \vdash C\{M/x\}}{E \vdash M : \{x : T \mid C\}} \qquad \textsc{Exp Subsum}\ \frac{E \vdash A : T \qquad E \vdash T <: T'}{E \vdash A : T'}$$

$$\textsc{Exp If}\ \frac{\begin{array}{c} E \vdash M : T_1 \qquad E \vdash N : T_2 \\ E, x : T_1 \wedge T_2, \_ : \{x = M \wedge M = N \wedge \mathsf{non\text{-}disj}(T_1, T_2)\} \vdash A : U \\ E, \_ : \{M \neq N\} \vdash B : U \end{array}}{E \vdash \mathsf{if}\ M = N\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B : U}$$

$$\textsc{Exp Assume}\ \frac{E \vdash \diamond \qquad \mathit{free}(C) \subseteq \mathit{dom}(E)}{E \vdash \mathsf{assume}\ C : \{\_ : \mathsf{unit} \mid C\}} \qquad \textsc{Exp Assert}\ \frac{E \vdash C}{E \vdash \mathsf{assert}\ C : \mathsf{unit}} \qquad \textsc{Exp Send}\ \frac{E \vdash M : T \qquad (a \updownarrow T) \in E}{E \vdash a!M : \mathsf{unit}}$$

$$\textsc{Exp Recv}\ \frac{E \vdash \diamond \qquad (a \updownarrow T) \in E}{E \vdash a? : T} \qquad \textsc{Exp For1}\ \frac{E \vdash A\{\widetilde{T}/\widetilde{\alpha}\} : V}{E \vdash \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ A : V} \qquad \textsc{Exp For2}\ \frac{E \vdash A\{\widetilde{U}/\widetilde{\alpha}\} : V}{E \vdash \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ A : V}$$

$$\textsc{Exp And}\ \frac{E \vdash A : T \qquad E \vdash A : U}{E \vdash A : T \wedge U} \qquad \textsc{Exp Case}\ \frac{\begin{array}{c} E \vdash M : T_1 \vee T_2 \\ E, x : T_1 \vdash A : U \qquad E, x : T_2 \vdash A : U \end{array}}{E \vdash \mathsf{case}\ x = M : T_1 \vee T_2\ \mathsf{in}\ A : U}$$

Intersection Types are introduced using the EXP AND and EXP FORi rules (and eliminated by subtyping). Intuitively, the expression for $\widetilde{\alpha}$ in $\widetilde{T}; \widetilde{U}$ do A is of type $T_1 \wedge T_2$ if $A\{\widetilde{T}/\widetilde{\alpha}\}$ is of type T_1 and $A\{\widetilde{U}/\widetilde{\alpha}\}$ is of type T_2. Thus in order to introduce an intersection type, we have to type-check A twice. The type annotations in A can differ between the two checks. The introduction of intersection types is useful, for instance, when type-checking a function that can be used by honest participants as well as by the attacker. For instance, the function for verifying digital signatures has a type of the form $\mathsf{Un} \to ((T \vee \mathsf{Un}) \to T) \wedge (\mathsf{Un} \to \mathsf{Un})$. Honest participants pass a signature together with the signed message, either at type T or at type Un, and get back the signed message with the stronger type T. The attacker that has only access to messages of type Un can still call this function, but the returned type is just Un.

The main result we expect for the type system is that well-typed programs are robustly safe. We are currently in the process of completing the proofs.

Claim (Robust Safety). If $\emptyset \vdash A : \mathsf{Un}$ then A is robustly safe.

3.5 Example

Let us consider again the protocol implementation from Section 2.2. Since we want to model a scenario in which the attacker has control over the network we give channel c type Un (i.e., $T_{\mathsf{chan}} = \mathsf{Un}$). The function *mkId* returns a public

identifier and it is thus of type $\mathsf{unit} \to \mathsf{Un}$. The function *getPriv* is instead of type $\mathsf{unit} \to \mathsf{Private}$, since it returns a private value. Notice that the sender signs a pair composed of a public identifier and a private message y for which the predicate $Ok(y)$ holds. In order to convey the predicate $Ok(y)$ from the sender to the receiver, we instantiate the polymorphic function for creating signing keys with the type $T_{\text{sign}} = \mathsf{Un} * \{z : \mathsf{Private} \mid Ok(z)\}$. Since the sender encrypts the signature together with the two signed values, the encryption function is instantiated with $T_{\text{enc}} = \mathsf{Un} * T_{\text{sign}}$.

We first analyze the sender's code. The message y returned by *getPriv* is of type Private. After $\mathsf{assume}\ Ok(y)$, we can give y type $\{z : \mathsf{Private} \mid Ok(z)\}$ by applying VAL REFINE. The type of the receiver's identifier y_b and the type of y comply with the type of the signing and encryption keys. The sender's code is thus well-typed.

The receiver decrypts the ciphertext and the result is stored in variable xy. This variable is of type $T_{\text{enc}} \vee \mathsf{Un}$[2], which is equivalent by subtyping to $\mathsf{Un} * (T_{\text{sign}} \vee \mathsf{Un})$. The receiver splits the pair, obtaining a variable x of type Un bound to the signature and a variable y of type $T_{\text{sign}} \vee \mathsf{Un}$ bound to the signed pair. We anticipate that these are precisely the types expected by the signature verification function, which returns a value y' of type T_{sign}, since the signing key is only used to sign messages of this type. The signed pair y' is split, obtaining a variable y_1 of type Un bound to the receiver's identifier and a variable y_2 of type $\{z : \mathsf{Private} \mid Ok(z)\}$ bound to the private message. This refinement type allows us to successfully typecheck the $\mathsf{assert}\ Ok(y_2)$. The receiver's code is thus also well-typed, so the whole program is well-typed and hence we expect it is robustly safe.

4 Implementation of Symbolic Cryptography

In this section, we illustrate the typed interface of the functions for digital signatures and public-key cryptography. Section 4.1 overviews the dynamic sealing mechanism used in [7] to encode symbolic cryptography, while sections 4.2 and 4.3 present our improvements to this encoding.

4.1 Dynamic Sealing

The notion of *dynamic sealing* was initially introduced by Morris [16] as a protection mechanism for programs. Later, Sumii and Pierce [19,18] studied the semantics of dynamic sealing within a λ-calculus, observing a close correspondence with symmetric-key cryptographic primitives.

In RCF [7] seals are encoded using pairs, functions, references and lists. A seal is a pair of a *sealing function* and an *unsealing function*, having type:

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

[2] The ciphertext might have been generated by a honest user or by the attacker.

The sealing function takes as input a term M of type α and returns a fresh value a of type Un, after adding the pair (M, a) to a secret list that is stored in a reference. The unsealing function takes as input a value a of type Un, scans the list in search of a pair (M, a), 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 a represents the ciphertext.

Let us take a look at the type $\mathsf{Seal}\,\langle\alpha\rangle$. If α is neither public nor tainted, as it is usually the case for symmetric-key cryptography, neither the sealing function nor the unsealing function are public, meaning that the symmetric key is kept secret. If α is tainted but not public, as usually the case for public-key cryptography, 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 α 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 no matter what is signed. As an example, in the Direct Anonymous Attestation protocol [12] the issuer signs the secret TPM's identifier and the TPM proves the knowledge of this certificate without revealing it by means of a zero-knowledge proof. The verifier, however, needs to have access to the TPM's verification key in order to verify the zero-knowledge proof.

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 and nonce handshakes, such as the Needham-Schroeder-Lowe protocol, where the type of the encrypted messages is neither public nor tainted.

These issues are not due to the sealing itself but to the type of seals, which is simple but not expressive enough to characterize some important usages of asymmetric cryptography.

4.2 Digital Signatures

In this section, we show how union and intersection types can be used to enhance the expressiveness of the type system and solve the aforementioned problems. The type of the seal used in our library for digital signatures is reported below:

$$\mathsf{Seal}_{\text{sign}}\langle\alpha\rangle = s : \mathsf{Un} * \mathsf{Sealing}_{\text{sign}}\langle\alpha\rangle * \mathsf{Unsealing}_{\text{sign}}\langle\alpha\rangle$$
$$\mathsf{Sealing}_{\text{sign}}\langle\alpha\rangle = \alpha \to \mathsf{Un}$$
$$\mathsf{Unsealing}_{\text{sign}}\langle\alpha\rangle = \mathsf{Un} \to \big((x : (\alpha \vee \mathsf{Un}) \to \{y : \alpha \mid x = y\}) \wedge (\mathsf{Un} \to \mathsf{Un})\big)$$

The seal is a triple composed of a seal identifier s of type Un, the sealing function, and the unsealing function. The identifier s is a "ghost variable" that is only used in the refinement types given to the sealing and unsealing function. The logical formulas occurring therein link the sealing function to the unsealing function and sealed values to the corresponding seal. This serves to characterize in the logic some important properties of cryptographic primitives, such as the determinism of the functions implementing signature verification and decryption. For easing the presentation, we will present simpler types obtained by removing such logical formulas by subtyping.

The implementation and the type of the sealing function for digital signatures is similar to the one in [7]. The unsealing function, however, differs both in the implementation and in the type. This function takes as input two arguments: the sealed value a representing a signature and the message M that was signed. Unsealing succeeds only if the pair (M, a) is in the secret list associated to the seal, and in this case it returns M. As described by the type $\mathsf{Unsealing}_{\mathrm{sign}}\langle\alpha\rangle$, the first argument of the unsealing function is of type Un and corresponds to the signature. The second part of this type is an intersection of two types: The type $x : (\alpha \vee \mathsf{Un}) \to \{y : \alpha \mid x = y\}$ is used to type-check honest callers: the signed message x passed to the unsealing function has either type α (e.g., if the message is received encrypted, as in our running example, or from a private channel) or of type Un (e.g., if the signature is received from a public channel), and the message y returned by the unsealing function has the stronger type α, which means that the unsealing function casts the type of the signed message from $(\alpha \vee \mathsf{Un})$ down to α. This is safe since the sealing function is not public and can only be used to sign messages of type α. The type $\mathsf{Un} \to \mathsf{Un}$ makes type $\mathsf{Unsealing}_{\mathrm{sign}}\langle\alpha\rangle$ always public, which allows the attacker to call the unsealing function. Since, in contrast to [7], the unsealing function and hence the verification key can be made public, we can additionally model protocols where the signing key is used to sign private messages while the verification key is public, such as DAA.

Finally, we present our typed interface for digital signatures:

$$
\begin{aligned}
&\mathit{mkSK}\langle\alpha\rangle : \mathsf{unit} \to \mathsf{Seal}_{\mathrm{sign}}\langle\alpha\rangle\\
&\mathit{mkVK}\langle\alpha\rangle : \big(x_{sk} : \mathsf{Seal}_{\mathrm{sign}}\langle\alpha\rangle \to \{x_{vk} : \mathsf{Unsealing}_{\mathrm{sign}}\langle\alpha\rangle \mid \mathit{SKPair}(x_{vk}, x_{sk})\}\big) \wedge \mathsf{Un}\\
&\mathit{sign}\langle\alpha\rangle : (x_{sk} : \mathsf{Seal}_{\mathrm{sign}}\langle\alpha\rangle \to y : \alpha \to \{z : \mathsf{Un} \mid \mathit{Signed}(x_{sk}, y, z)\}) \wedge \mathsf{Un}\\
&\mathit{check}\langle\alpha\rangle\colon \big(x_{vk}\colon \mathsf{Unsealing}_{\mathrm{sign}}\langle\alpha\rangle \to z : \mathsf{Un} \to x : (\alpha \vee \mathsf{Un}) \to\\
&\qquad\qquad \{y : \alpha \mid y = x \wedge \exists S.\mathit{SKPair}(x_{vk}, S) \wedge \mathit{Signed}(S, x, z)\}\big) \wedge \mathsf{Un}
\end{aligned}
$$

The *mkSK* function generates a signing key, which is just a seal, while *mkVK* takes this seal and returns the verification key, which is just the unsealing component of the seal. The *sign* function takes as input the signing key x_{sk} and a message y, and it applies the sealing function to the message obtaining a sealed value z of type $\{z : \mathsf{Un} \mid \mathit{Signed}(x_{sk}, y, z)\}$. The predicate in this refinement type records the signing operation. The *check* function takes as input the verification key x_{vk}, a sealed value z, and the signed message x, and it returns a value of type $\{y : \alpha \mid y = x \wedge \exists S.\mathit{SKPair}(x_{vk}, S) \wedge \mathit{Signed}(S, x, z)\}$. The formula in this refinement type says that the value returned by this function is the signed mes-

sage x and that there exists a signing key S associated to x_{vk} such that z is the value obtained by signing x with S. Notice that we give *mkVK*, *sign* and *check* intersection types similar to $\mathsf{Unsealing}_{\text{sign}}\langle\alpha\rangle$.

4.3 Public-Key Encryption

For public-key encryption we use a seal of type $\mathsf{Seal}\,\langle\alpha \vee \mathsf{Un}\rangle$. The sealing function takes as input a value of type α or one of type Un, depending on whether the encryption is performed by a honest user or by the attacker, and it returns a sealed value of type Un. In contrast to [7], the sealing function is always public, even if the type α of the encrypted message is not tainted. Finally, we present the typed interface of the functions implementing public-key cryptography. The formulas in the refinement types are similar to the ones for digital signatures.

$$
\begin{aligned}
&\mathit{mkDK}\langle\alpha\rangle : \mathsf{unit} \to \mathsf{Seal}\langle\alpha \vee \mathsf{Un}\rangle\\
&\mathit{mkEK}\langle\alpha\rangle : (x_{dk} : \mathsf{Seal}\langle\alpha \vee \mathsf{Un}\rangle \to \{x_{ek} : \mathsf{Sealing}\langle\alpha \vee \mathsf{Un}\rangle \mid \mathit{EKPair}(x_{ek}, x_{dk})\}) \wedge \mathsf{Un}\\
&\mathit{encrypt}\langle\alpha\rangle : (x_{ek} : \mathsf{Sealing}\langle\alpha \vee \mathsf{Un}\rangle \to y : (\alpha \vee \mathsf{Un}) \to \{x : \mathsf{Un} \mid \mathit{Encrypted}(x_{ek}, y, x)\}) \wedge \mathsf{Un}\\
&\mathit{decrypt}\langle\alpha\rangle : (x_{dk} : \mathsf{Seal}\langle\alpha \vee \mathsf{Un}\rangle \to x : \mathsf{Un} \to\\
&\qquad\qquad \{y : \alpha \vee \mathsf{Un} \mid \exists E.\mathit{EKPair}(E, x_{dk}) \wedge \mathit{Encrypted}(E, y, x)\}) \wedge \mathsf{Un}
\end{aligned}
$$

5 Encoding of Zero-knowledge

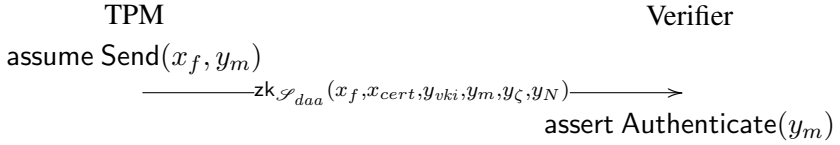
This section describes how we automatically generate the symbolic implementation of non-interactive zero-knowledge proofs, starting from a high-level specification. Intuitively, this implementation resembles an oracle that provides three operations: one for creating zero-knowledge proofs, one for verifying such proofs, and one for obtaining the public values used to create the proofs.

For creating a zero-knowledge proof the caller needs to provide values for the variables in the specification. Some of these values are revealed by the proof to the verifier and to any eavesdropper, while the others (i.e., the witnesses of the proof) are kept secret. A zero-knowledge proof does not reveal any information about these witnesses, other than the validity (or invalidity) of the statement that is being proved. A zero-knowledge proof is valid if the values that are used to create it satisfy the statement of the proof. When creating a proof, however, we do not require that it is valid. Instead, a second operation is provided for verifying the validity of zero-knowledge proofs. The third operation allows one to obtain the public values of zero-knowledge proofs. Note that these three operations need to be available not only to the honest participants, but also to the attacker.

We implement such 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.

5.1 Example: Simplified DAA

As a running example throughout this section, we consider a simplified version of the Direct Anonymous Attestation (DAA) protocol [12]. The goal of the DAA protocol is to enable the TPM to sign arbitrary messages and to send them to an entity called the verifier in such a way that the verifier will only learn that a valid TPM signed that message, but without revealing the TPM's identity. The DAA protocol is composed of two sub-protocols: the *join protocol* and the *DAA-signing protocol*. The join protocol allows a TPM to obtain a certificate x_{cert} from an entity called the issuer. This certificate is just a signature on the TPM's secret identifier x_f. The DAA-signing protocol enables a TPM to authenticate a message y_m by proving to the verifier the knowledge of a valid certificate, but without revealing the TPM's identifier or the certificate. In this section, we focus on the DAA-signing protocol and we assume that the TPM has already completed the join protocol and received the certificate from the issuer. In the DAA-signing protocol the TPM sends to the verifier the following zero-knowledge proof:



where the statement $S_{\mathscr{S}_{daa}}$ is of the form $x_f = \mathsf{check}\ y_{vki}\ x_{cert}\ x_f \ \wedge\ y_N = \mathsf{exp}\ y_\zeta\ x_f$. Intuitively, in the first conjunct 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. The TPM identifier and the certificate are kept secret, while the verification key of the issuer is revealed to the verifier. This zero-knowledge proof additionally conveys a public message y_m that the TPM wants to authenticate with the verifier. Notice that this proof guarantees non-malleability, i.e., the attacker cannot change y_m without redoing the proof, since the certificate and the identifier are kept secret. The second conjunct states that y_N is obtained by exponentiating y_ζ to x_f. If y_ζ is freshly chosen for each DAA-signature then the anonymity of the user is preserved. If, however, y_ζ is fixed for each verifier then two DAA-signatures done by the same TPM can be linked by the verifier, yielding the pseudonymous version of DAA.

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

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

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

5.2 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 [6,4]. For a specification $\mathscr{S}$ 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.

Variables. We use variables to stand for the *witnesses* and public values of a zero-knowledge 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. The value of y_{vki} is matched against the signature verification key of the issuer, which the verifier of the zero-knowledge proof already knows. The payload message y_m, as well as the values of y_N and y_ζ are returned to the verifier of the proof, so they are returned variables.

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.

$$S, R \ ::= \ (x = f\langle\widetilde{T}\rangle\ x_1 \ldots x_n) \ \mid \ S_1 \wedge S_2 \ \mid \ S_1 \vee S_2$$

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 behaviour[3], 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 $S_{\mathscr{S}_{daa}} = (x_f = \mathsf{check}\langle T_{vki}\rangle\ y_{vki}\ x_{cert}\ x_f \ \wedge \ y_N = \mathsf{exp}\ y_\zeta\ x_f)$. This statement is valid for a certain instantiation if the check function returns the value of x_f when the values of y_{vki}, x_{cert}, and x_f are passed as arguments, and if $\mathsf{exp}\ y_\zeta\ x_f$ returns y_N. Note that although the payload message y_m does not occur in the statement, the proof guarantees non-malleability so an attacker cannot change y_m without redoing the proof.

Types. The user also needs to provide a type for all specified variables. In the following we assume a function $t_{\mathscr{S}}$ that assigns a type to each variable in specification $\mathscr{S}$. The DAA-sign protocol does not preserve the secrecy of the signed message, so $t_{\mathscr{S}_{daa}}(y_m) = \mathsf{Un}$. Similarly, we have that $t_{\mathscr{S}_{daa}}(y_\zeta) = t_{\mathscr{S}_{daa}}(y_N) = \mathsf{Un}$. On the other hand, the TPM identifier x_f is given a secret and untainted type: $t_{\mathscr{S}_{daa}}(x_f) = T_{vki} = \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\}$. 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

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

messages of type T_{vki}, so it is given type $\mathsf{Unsealing}_{\mathsf{sign}}\,\langle T_{vki}\rangle$. Finally the certificate x_{cert} is a signature, so it has type Un. Even though it has type Un, it would break the anonymity of the user to make the certificate a public value, since the verifier could then always distinguish if two consecutive requests come from the same user or not (as in the pseudonymous version of DAA).

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[4], 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 $P_{\mathscr{S}_{daa}} = \mathsf{Send}(x_f, y_m)$, since this predicate holds true in the typing environment of a honest TPM.

5.3 Automatic Code Generation

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

Typed Interface The interface generated for a specification $\mathscr{S}$ contains three functions[5] that share hidden state (a seal for values of type $\tau_{\mathscr{S}}$):

$$\mathsf{create}_{\mathscr{S}} : \tau_{\mathscr{S}} \to \mathsf{Un},\ \text{where } \tau_{\mathscr{S}} = \mathsf{Un} \vee \textstyle\sum_{x\in\mathsf{vars}_{\mathscr{S}}} x : t_{\mathscr{S}}(x).\{P_{\mathscr{S}}\}$$

$$\mathsf{public}_{\mathscr{S}} : \mathsf{Un} \to \mathsf{Un}$$

$$\mathsf{verify}_{\mathscr{S}} : \mathsf{Un} \to (\mathsf{Un} \wedge \prod_{y\in\mathsf{matched}_{\mathscr{S}}} y : t_{\mathscr{S}}(y).\ \sum_{z\in\mathsf{returned}_{\mathscr{S}}} z : t_{\mathscr{S}}(y).\{\underset{\widetilde{x}=\mathsf{witness}_{\mathscr{S}}}{\exists\widetilde{x}.} F(S_{\mathscr{S}}, E) \wedge P_{\mathscr{S}}\} \quad)$$

The function $\mathsf{create}_{\mathscr{S}}$ is used to create zero-knowledge proofs for specification $\mathscr{S}$. It takes as argument a tuple containing values for all variables of the proof, or an argument of type Un if it is called by the adversary. In case a protocol participant calls this function, we check that the values have the types provided by the user. Additionally, we check that the promise $P_{\mathscr{S}}$ provided by the user holds in the typing environment of the prover. The returned zero-knowledge proof is given type Un so that it can be sent over the public network. For instance, in the DAA example we have that: $\tau_{\mathscr{S}_{daa}} = \mathsf{Un} \vee ((y_{vki}{:}\mathsf{Unsealing}_{\mathsf{sign}}\,\langle T_{vki}\rangle * y_m{:}\mathsf{Un} * y_\zeta{:}\mathsf{Un} * y_N{:}\mathsf{Un} * x_f{:}T_{vki} * x_{cert}{:}\mathsf{Un}) * \{\mathsf{Send}(x_f, y_m)\})$, where $T_{vki} = \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\}$.

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

[4] Signature proofs of knowledge have this property [12].

[5] 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 $\mathsf{verify}_{\mathscr{S}}$ is used for verifying zero-knowledge proofs. This function can be called by the attacker in which case it returns a value of type Un. When called by a protocol participant, however, it takes as argument a candidate zero-knowledge proof of type Un and the values for the matched variables, which have the user-specified types. On successful verification, this function returns a tuple containing the values of the public variables, again with their respective types. The function guarantees that the formula $\exists\widetilde{x}.F(S_{\mathscr{S}}, E) \wedge P_{\mathscr{S}}$ holds, where the public variables are free and the witnesses are existentially quantified. The first conjunct, $F(S_{\mathscr{S}}, E)$, guarantees that if verification succeeds then the statement indeed holds, no matter what the origin of the proof is. Since the statement itself is not a formula in the logic (as it was for instance the case in [4]), 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.

For instance, in the DAA example, we have that

$$F(S_{\mathscr{S}_{daa}}, E_{std}) = F(x_f = \mathsf{check}\langle T_{vki}\rangle\ y_{vki}\ x_{cert}\ x_f, E_{std}) \wedge F(y_N = \mathsf{exp}\ y_\zeta\ x_f, E_{std})$$

The second conjunct is equal to $Exp(y_\zeta, x_f, y_N)$, since the exponentiation function is typed to $x : \top \to y : \top \to \{z : \mathsf{Un} \mid Exp(x, y, z)\}$ in our library. As explained in Section 4, we have that $E_{std} \vdash \mathsf{check}\langle T_{vki}\rangle : xvk : \mathsf{Unsealing}_{\mathsf{sign}}\langle T_{vki}\rangle \to z : \mathsf{Un} \to x : (T_{vki} \vee \mathsf{Un}) \to \{y : T_{vki} \mid y = x \wedge \exists SK.SKPair(xvk, SK) \wedge Signed(SK, x, z)\}$. So for the first conjunct we obtain the formula: $(x_f = x_f) \wedge (\exists sk.SKPair(y_{vki}, sk) \wedge Signed(sk, x_f, x_{cert})) \wedge \mathsf{OkTPM}(x_f)$. The predicate $\mathsf{OkTPM}(x_f)$ was obtained from the nested refinement type T_{vki}. Finally, after removing the trivial equality we obtain that:

$$F(\mathscr{S}_{daa}, E_{std}) = (\exists sk.SKPair(y_{vki}, sk)\ \wedge Signed(sk, x_f, x_{cert})) \wedge \mathsf{OkTPM}(x_f) \wedge Exp(y_\zeta, x_f, y_N)$$

Generated Implementation The generated implementation for specification $\mathscr{S}$ creates a fresh seal $k_{\mathscr{S}}$ for values of type $\tau_{\mathscr{S}} = \mathsf{Un} \vee \sum_{x \in \mathsf{vars}_{\mathscr{S}}} x : t_{\mathscr{S}}(x).\{P_{\mathscr{S}}\}$. The sealing function of $k_{\mathscr{S}}$ is directly used to implement the $\mathsf{create}_{\mathscr{S}}$ function. The unsealing function of $k_{\mathscr{S}}$ is used to implement the $\mathsf{public}_{\mathscr{S}}$ and $\mathsf{verify}_{\mathscr{S}}$ functions. The implementation of $\mathsf{public}_{\mathscr{S}}$ is also simple: since the zero-knowledge proof is just a sealed value, $\mathsf{public}_{\mathscr{S}}$ unseals it and returns the public values as a tuple. The witnesses are discarded, and the validity of the statement is not checked.

The implementation of the $\mathsf{verify}_{\mathscr{S}}$ function is more involved. This function takes a candidate zero-knowledge proof z of type Un as input, and values for the matched variables. Since the type of $\mathsf{verify}_{\mathscr{S}}$ contains an intersection type (Un is one of the branches and this makes the type of $\mathsf{verify}_{\mathscr{S}}$ public) we use a for construct to introduce this intersection type. If the proof is verified by the attacker we can assume that the matched variables have type Un and need to type the return

value to Un. On the other hand, if the proof is verified by a protocol participant we can assume that for all $y' \in \mathsf{matched}_{\mathscr{S}}$ we have $y' : t_{\mathscr{S}}(y')$, and need to give the returned value type $\sum_{y \in \mathsf{returned}_{\mathscr{S}}} y : t_{\mathscr{S}}(y).\{\exists_{\widetilde{x}=\mathsf{witness}_{\mathscr{S}}} \widetilde{x}.\ F(S_{\mathscr{S}}, E) \wedge P_{\mathscr{S}}\}$. Intuitively, the strong types of the matched values should allow us to guarantee the strong types of the returned public values, as well as the promise $P_{\mathscr{S}}$.

The generated $\mathsf{verify}_{\mathscr{S}}$ function performs the following five steps: (1) it unseals z using "$\mathsf{snd}\ k_{\mathscr{S}}$" 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 $(y_{vki}, y_m, y_\zeta, y_N)$ 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 functions in $S_{\mathscr{S}_{daa}}$ and checking the results for equality with the corresponding values. This last step is slightly complicated by the fact that the statement can contain disjunctions, so we use decision trees. However, for the DAA example the decision tree has a very simple (linear) structure.

$$
\begin{array}{l}
\mathsf{verify}_{\mathscr{S}_{daa}} = \lambda z : \mathsf{Un}. \\
\quad \mathsf{for}\ \alpha\ \mathsf{in}\ \mathsf{Un}; \mathsf{Unsealing}_{\mathrm{sign}}\ \langle T_{vki} \rangle\ \mathsf{do}\ \lambda y'_{vki} : \alpha. \\
\qquad \mathsf{let}\ z' = (\mathsf{snd}\ k_{\mathscr{S}})\ z\ \mathsf{in} \\
\qquad \mathsf{case}\ z'' = z' : \mathsf{Un} \vee (\mathsf{Unsealing}_{\mathrm{sign}}\ \langle T_{vki} \rangle * y_m{:}\mathsf{Un} * \mathsf{Un} * \mathsf{Un} * x_f{:}T_{vki} * \\
\mathsf{Un} * \{\mathsf{Send}(x_f, y_m)\})\ \mathsf{in} \\
\qquad \mathsf{let}\ (y_{vki}, y_m, y_\zeta, y_N, x_f, x_{cert}) = z''\ \mathsf{in} \\
\qquad \mathsf{if}\ (y_{vki}) = (y'_{vki})\ \mathsf{as}\ (y''_{vki})\ \mathsf{then} \\
\qquad\quad \mathsf{if}\ x_f = \mathsf{check}\langle T_{vki} \rangle\ y''_{vki}\ x_{cert}\ x_f\ \mathsf{as}\ x'_f\ \mathsf{then} \\
\qquad\qquad \mathsf{if}\ y_N = \mathsf{exp}\ y_\zeta\ x_f\ \mathsf{as}\ y'_N\ \mathsf{then} \\
\qquad\qquad\quad (y_m, y_\zeta, y'_N, ()) \\
\qquad\qquad \mathsf{else}\ \mathit{failwith}\ () \\
\qquad\quad \mathsf{else}\ \mathit{failwith}\ () \\
\qquad \mathsf{else}\ \mathit{failwith}\ ()
\end{array}
$$

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. Note that because of the for and case constructs the body of $\mathsf{verify}_{\mathscr{S}}$ 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 general, there are two situations in which type-checking the generated implementation fails. First, the types provided by the user for the public witnesses are not public. In this case the implementation of $\mathsf{public}_{\mathscr{S}}$ cannot match its de-

fined type $\mathsf{Un} \rightarrow \mathsf{Un}$. Second, the promise $P_{\mathscr{J}}$ is not justified by the statement and the types of the witnesses. In this case $\mathsf{verify}_{\mathscr{J}}$ cannot match its defined type.

6 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 Section 3, an automatic code generator for zero-knowledge as defined in Section 5, an interpreter, and a visual debugger.

The type-checker supports an extended syntax with respect to the one in the paper, 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 an automatic theorem prover to discharge proof obligations. We have tested the type-checker on the simplified DAA protocol from Section 5.1 and on other simple examples such as nonce handshakes; the analysis only took several seconds in each case. 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 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.

The type-checker, the code generator for zero-knowledge, the interpreter are command-line tools implemented in F#, while the GUIs of the visual debugger and the visualizer for type derivations are specified using WPF (Windows Presentation Foundation). The type-checker consists of around 2000 lines of code, while the whole tool-chain has over 4500 lines of code.

7 Conclusions and Future Work

We have presented a general technique for verifying implementations of cryptographic protocols based on zero-knowledge proofs. The security of $\mathrm{RCF}^{\forall}_{\wedge\vee}$ protocol implementations is verified by a type system combining refinement, union, and intersection types. The analysis is automated and modular, and was efficient on the examples that we tried.

As future work, we plan to investigate the automated generation of concrete cryptographic implementations of zero-knowledge proofs, and thus to complement the generation of symbolic implementations considered in this paper.

Finally, we intend to apply our framework to analyze modern cryptographic applications, such as the full implementation of the Direct Anonymous Attestation protocol and the recently proposed Civitas electronic voting system [14].

Acknowledgments. We thank Cédric Fournet and Andrew D. Gordon for many constructive discussions.

References

1. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Proc. 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, pages 25–41. Springer-Verlag, 2001.
2. M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.
3. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. 28th Symposium on Principles of Programming Languages (POPL)*, pages 104–115. ACM Press, 2001.
4. M. Backes, C. Hriţcu, and M. Maffei. Type-checking zero-knowledge. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 357–370. ACM Press, 2008.
5. M. Backes, C. Hriţcu, M. Maffei, and T. Tarrach. Type-checking implementations of protocols based on zero-knowledge proofs. Long version available at `http://www.infsec.cs.uni-sb.de/~hritcu/publications/zk-rcf-full.pdf`, 2009.
6. M. Backes, M. Maffei, and D. Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *Proc. 29th IEEE Symposium on Security and Privacy*, pages 202–215. IEEE Computer Society Press, 2008.
7. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, pages 17–32, 2008.
8. K. Bhargavan, R. Corin, P.-M. Dénielou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Proc. 22th IEEE Symposium on Computer Security Foundations (CSF)*, 2009. To appear.
9. K. Bhargavan, R. Corin, C. Fournet, and E. Zălinescu. Cryptographically verified implementations for TLS. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 459–468. ACM Press, 2008.
10. K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 139–152. IEEE Computer Society Press, 2006.
11. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE Computer Society Press, 2001.
12. E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proc. 11th ACM Conference on Computer and Communications Security*, pages 132–145. ACM Press, 2004.
13. S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. Technical report, CMU CyLab, October 2008.
14. M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: A secure voting system. In *Proc. 29th IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society Press, 2008.
15. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2005)*, pages 363–379. Springer, 2005.
16. J. Morris. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.
17. B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.
18. E. Sumii and B. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375(1-3):169–192, 2007.
19. E. Sumii and B. C. Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4):521–554, 2003.