Type-checking Implementations of Protocols Based on Zero-knowledge Proofs

Michael Backes[1,2], Cătălin Hriţcu[1], Matteo Maffei[1], Thorsten Tarrach[1]
[1]Saarland University [2]Max Planck Institute for Software Systems (MPI-SWS)

April 27, 2009

Abstract

We present the first static analysis technique for verifying implementations of cryptographic protocols based on zero-knowledge proofs. Protocols are implemented in RCF, a core calculus of ML, and cryptographic primitives are considered fully reliable building blocks and represented symbolically using a sealing mechanism. The statements of the zero-knowledge proofs are specified in a high-level language and automatically compiled down to a symbolic implementation using seals. An expressive type system combining refinement, union, and intersection types allows us to statically characterize the security properties offered by zero-knowledge proofs and, in general, to extend the scope of type-based analyses of protocol implementations to important protocol classes not covered so far.

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 [28, 41], over carefully designed de-facto standards like SSL and PKCS [53, 19], up to current widely deployed products like Microsoft Passport [31] and Kerberos [22]. Hence work towards the automation of such proofs has started soon after the first protocols were developed, focusing on high-level protocol descriptions (e.g., process calculi [46, 7, 6, 30] and strand spaces [52]) that are suited to the formalization of security properties and to the automation of security proofs(e.g., see [38, 1, 18, 3, 5, 27, 32, 29, 26]).

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 [20] and electronic voting [24] 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 [6, 4], which rendered an automated analysis of such applications feasible in abstract models [11, 9]. 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 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 [13], a core calculus of ML, and cryptographic primitives are considered fully reliable building blocks and represented symbolically using a sealing mechanism [40, 50]. The statements of 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 provide a precise characterization of asymmetric cryptography that allows us to verify protocols based on nested cryptography, signatures of private data, and public-key encryption of authenticated data. The analysis is fully automated, efficient, compositional, and provides security proofs for an unbounded number of sessions.

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# [13]. Their type system supports most of the features of F# by translation to RCF. The analysis is compositional and promises to scale up to large implementations [15]. 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 [20], 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 [35] 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 [23] 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 [17, 16] for the verification of F# protocol implementations by automatically extracting ProVerif models [18]. 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 [16]. The considered

Table 1 Syntax of RCF 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	$\mathsf{unfold}\ M$	use recursive value
$\Lambda\alpha.A$	polymorphic val.	$\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ A$	introduction of intersection types
$\mathsf{fold}\ M$	recursive val.	$\mathsf{case}\ x = M\ \mathsf{in}\ A$	elimination of union types
$\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ M$		$(\nu a \updownarrow T)A$	restriction
		$A \Rsh B$	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: We use square brackets to denote optional parts. Given a phrase of syntax ϕ, we let $\phi\{M/x\}$ denote the substitution of each free occurrence of the variable x in ϕ with the value M. Finally, we use $\widetilde{\phi}$ to denote the sequence $\phi_1, \ldots, \phi_n$ for some n.

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 RCF, while Section 3 describes the type system we consider. Section 4 illustrates our symbolic implementation of asymmetric cryptography and zero-knowledge. Section 5 discusses our encoding of zero-knowledge proofs in RCF and applies it to a simplified version of the DAA protocol. Section 6 describes our implementation. Section 7 concludes and discusses future work. The implementation is available at [10].

2 RCF

This section outlines the Refined Concurrent FPC (RCF) [13], a simple programming language extending the Fixed Point Calculus [36] with refinement types [33, 45, 54] and concurrency [8]. Although very simple, this core calculus is expressive enough to encode a big part of F# [13, 14]. In this paper, we further increase the expressivity of the calculus by adding intersection types [42], union types, and parametric polymorphism [44, 34].

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 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 a multiset of expressions being evaluated in parallel; a multiset of messages sent on channels but not yet received; and a global log containing a multiset of assumed formulas. Our technique is general and can use different authorization logics for the formulas[1]. Values are irreducible. The function application $(\lambda x{:}T.A)\ M$ reduces to $A\{M/x\}$. A type instantiation

[1] In our implementation, we consider first-order logic with equality as the authorization logic.

Table 2 A direct implementation of a sign-then-encrypt protocol in RCF

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

$(\Lambda\alpha.A)\langle T\rangle$ 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. The variable x is given the intersection of the types of M and N, which makes the conditional a safe type-cast operator (more details are given in Section 3.4). The introduction form for intersection types $\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ A$ reduces to A, while the elimination form for union types $\mathsf{case}\ x = M\ \mathsf{in}\ A$ 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. 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, written as $S \models C$, we say the assertion *succeeds*; otherwise, we say the assertion *fails*.

Definition 2.1 (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 2.2 (Opponent and Robust Safety) *A closed expression O is an* opponent *if and only if O contains no assertions. A closed expression A is* robustly safe *if and only if the application $O\ A$ is safe for all opponents O.*

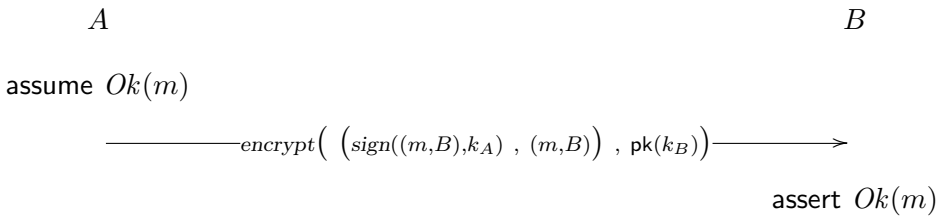
Table 3 Syntax of Types

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

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

2.2 Example: Sign-then-Encrypt

We are going to 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:



The RCF implementation of this protocol is reported in Table 2. We first generate a fresh public channel c and B's identifier x_b. We then create the signing and encryption key-pairs. Since the functions implementing cryptographic primitives are polymorphic, we instantiate them with the types T_{sign} and T_{enc} describing the messages signed and encrypted in this protocol, respectively (the precise definition of these types will be given in Section 3.5). The sender gets a private string y from the user and assumes the predicate $Ok(y)$. The sender then signs this message together with the receiver's identifier x_b, encrypts this signature together with the signed pair (x_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 string sent by A.

3 Type System

This section presents a type system for enforcing authorization policies on RCF code. This extends the type system proposed by Bengtson et al. [13] with union types, intersection types [42] and unrestricted parametric polymorphism [44, 34]. 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

Table 4 Entailment $E \models C$

$$\textsc{Derive}\quad \frac{E \vdash \diamond \qquad \mathit{free}(C) \subseteq \mathit{dom}(E) \qquad \mathit{forms}(E) \models C}{E \models C}$$

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

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 [13], we use refinement types [33, 45, 54] 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. The iso-recursive type $\mu\alpha.T$ is the type of all values $\mathsf{fold}\ (M)$ such that M is of type $T\{\mu\alpha.T/\alpha\}$. The polymorphic type [44, 34] $\forall\alpha.T$ (i.e., universal type) describes values $\Lambda\alpha.A$ such that $A\{T'/\alpha\}$ is of type $T\{T'/\alpha\}$ for all T' .

In the following we explain the typing judgements and present the most important typing rules. A complete definition of the type system is given in Appendix B.

3.2 Typing Environment and Entailment

A typing environment E is a list containing type variables α, bounded type variables $\alpha :: k$, subtyping constraints $\alpha <: \alpha'$ between type variables, bindings for names $a \updownarrow T$, where a stands for a a channel conveying values of type T, and type bindings for variables $x : T$.

A crucial judgment in the type system is $E \models C$, which states that the formula C is derivable from E. Intuitively, our type system ensures that whenever $E \models C$ 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. This judgment is formalized in Table 4. 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 $E \models C\{y/x\}$. 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. Notice that 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.

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 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. In our type system, we introduce an additional property for refinement types, i.e., $\{x : T \mid C\}$ is subtype of any other type in an environment in which $\forall x.\neg C$ holds: Intuitively, the type $\{x : T \mid C\}$ is not populated in an environment entailing $\forall x.\neg C$, unless the environment is inconsistent in which case all types become equivalent by subtyping. By relying on this property, we define the bottom type $\bot$ as $\{x : \mathsf{Un} \mid \mathsf{false}\}$. By the transitivity of subtyping, a refinement type $\{x : T \mid C\}$ is public if T is public, since then $\{x : T \mid C\} <: T <: \mathsf{Un}$, or if the formula $\forall x.\neg C$ is entailed by the environment, since then $\{x : T \mid C\} <:> \bot <: \mathsf{Un}$. Conversely, type $\{x : T \mid C\}$ is tainted if T is tainted and additionally $\forall x.C$ holds.

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 accepts all the inputs and returns a subset of the outputs of the latter. Consequently, 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 $(T \to U) <: (\mathsf{Un} \to \mathsf{Un}) <:> \mathsf{Un}$. Conversely, $T \to U$ is tainted if T is public and U is tainted. By relying on this property, we define the type $\mathsf{Private}$ as $\bot \to \bot$, which is neither public nor tainted (again, unless the environment is inconsistent).

Union and Intersection Types. The rules for subtyping and kinding union and intersection types are reported in Table 5. The type $T \wedge U$ is a subtype of T' if T or U is a subtype of T' (cf. Sub-And-LB), while T' is a subtype of $T \wedge U$ if it is subtype of both T and U (cf. Sub-And-Greatest). Intuitively, the set of values of type $T \wedge U$ is the intersection between the set of values of type T and the set of values of type U. Similarly, $T \wedge U$ is public if T or U is public, and it is tainted if both T and U are tainted. The dual rules for union types are given in Table 5.

3.4 Typing Values and Expressions

The main judgment of our type system 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 6. Most of them are standard, so we focus the explanation only on the rules that are new with respect to [13].

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

Table 5 Kinding and Subtyping Unions and Intersections

Subtyping

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

$$\textsc{Sub Or UB}\ \frac{\Gamma \vdash T <: U_i}{\Gamma \vdash T <: U_1 \vee U_2}$$

Kinding

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

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

cannot be disjoint. However, certain types such as Un and $\mathsf{Private}$ have common values only if the environment is inconsistent, i.e., $E \models \mathsf{false}$. Therefore, in such a case 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. The same idea has been applied in [2] for verifying the secrecy of nonce handshakes.

Union Types are introduced by subtyping, 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 $\mathsf{case}\ x = M\ \mathsf{in}\ A$, 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 both under the assumption that x is of type T and under the assumption that x is of type Un.

Intersection Types are introduced using the Exp For rule (and eliminated by subtyping). The expression $\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ A$ 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.

Table 6 Selected Rules for Values and Expressions $E \vdash A : T$

$$\textsc{Val Fun}\ \frac{E, x : T \vdash A : U}{E \vdash \lambda x : T.A : (x : T \to U)} \qquad \textsc{Val Refine}\ \frac{E \vdash M : T \qquad E \models C\{M/x\}}{E \vdash M : \{x : T \mid C\}} \qquad \textsc{Exp Subsum}\ \frac{E \vdash A : T \qquad E \vdash T <: T'}{E \vdash A : T'}$$

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

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

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

$$\textsc{Exp Recv}\ \frac{E \vdash \diamond \qquad (a \updownarrow T) \in E}{E \vdash a? : T} \qquad \textsc{Exp For}\ \frac{i \in \{1, 2\} \qquad E \vdash A\{\widetilde{T^i}/\widetilde{\alpha}\} : U}{E \vdash \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T^1}; \widetilde{T^2}\ \mathsf{do}\ A : U} \qquad \textsc{Exp And}\ \frac{E \vdash A : T \qquad E \vdash A : U}{E \vdash A : T \wedge U}$$

$$\textsc{Exp Case}\ \frac{E \vdash M : T_1 \vee T_2 \qquad E, x : T_1 \vdash A : U \qquad E, x : T_2 \vdash A : U}{E \vdash \mathsf{case}\ x = M\ \mathsf{in}\ A : U}$$

3.5 Example

Let us consider again the protocol implementation from Section 2.2. Since we want to prove the safety of the code when executed in parallel with the opponent, we give channel c type Un (i.e., $T_{\text{chan}} = \mathsf{Un}$). The function *mkId* returns a public identifier and it is thus of type $\mathsf{unit} \to \mathsf{Un}$. The function *getPrivString* is instead of type $\mathsf{unit} \to \mathsf{Private}$, since it returns a private string . 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 $T_{\text{enc}} = \mathsf{Un} * \mathsf{Un} * \{z : \mathsf{Private} \mid Ok(z)\}$.

We first analyze the sender's code. The message y returned by *getPrivString* is of type $\mathsf{Private}$. 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 x_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 $\mathsf{Un} * \{z : \mathsf{Private} \mid Ok(z)\} \vee \mathsf{Un} * \mathsf{Un}^2$, which is equivalent by subtyping to $\mathsf{Un} * (\{z : \mathsf{Private} \mid Ok(z)\} \vee \mathsf{Un})$. The receiver splits the pair, obtaining a variable x of type Un bound to the signature and a variable y of type $(\{z : \mathsf{Private} \mid Ok(z)\} \vee \mathsf{Un})$ bound to the signed pair. We anticipate here that these are precisely the types expected by the signature verification function,

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

which returns a value y' of type $\mathsf{Un} * \{z : \mathsf{Private} \mid Ok(z)\}$, since the signing key is secret and 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 type-check the $\mathsf{assert}\ Ok(y_2)$. The receiver's code is thus also well-typed, so the whole program is well-typed and hence 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 sealing mechanism used in [13] 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 [40] as a protection mechanism for programs. Later, Sumii and Pierce [50, 49] studied the semantics of dynamic sealing within a λ-calculus, observing a close correspondence with symmetric-key cryptographic primitives.

In RCF [13], a seal is a pair of a *sealing function* and an *unsealing function*. The type of a seal is:

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

The sealing function takes as input a term M of type α and returns a fresh value a of type Un, after storing the pair (M, a) in a secret list . The unsealing function takes as input a value a of type Un, scans the list in search for a pair (M, a), and returns M. Only the sealing function and the unsealing function can access this 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 verification keys are always public , regardless of what is signed. As an example, in the Direct Anonymous Attestation protocol [20] 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 [37], where the type of the encrypted messages is neither public nor tainted.

These issues are not due to sealing itself but to the type $\mathsf{Seal}\langle\alpha\rangle = \alpha \to \mathsf{Un} * \mathsf{Un} \to \alpha$ for seals, which is simple but not expressive enough to characterize all 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:

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

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 [13]. 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[3], which allows the attacker to call the unsealing function. Since, in contrast to [13] , the unsealing function and hence the verification key is always public, we also can 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

[3] 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.

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 message 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 $\mathsf{Unsealing}_{\mathrm{sign}}\langle\alpha\rangle$. While making these functions available to the adversary is not strictly necessary[4], this is convenient for the encoding of zero-knowledge we describe in Section 5.

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 or a honest user by the attacker, and it returns a sealed value of type Un. The unsealing function takes as input a sealed value of type Un and it returns the unsealed value of type $\alpha \vee \mathsf{Un}$. In contrast to [13], the sealing function is always public, even if the type α of the encrypted message is not tainted[5]. 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 discussed in Section 4.2.

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

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 of their statements. Intuitively, this implementation resembles an oracle that provides three operations: one for creating zero-knowledge proofs of the statement, one for verifying proofs of the statement, and one for obtaining the public witnesses of such proofs.

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

We implement such a zero-knowledge oracle in RCF as three functions that share a secret seal. In order to create a zero-knowledge proof the first function seals the witnesses provided by

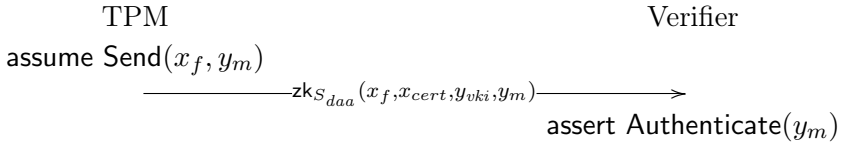
[4]The attacker can already sign messages using a signing key to which he has access or obtain the verification key corresponding to it by projecting the corresponding components in the triple representing the signing key. He can also verify signatures by directly using the unsealing function inside the public verification key.

[5]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.

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 witnesses, 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 witnesses of the proof. The public witnesses can also be obtained with the third function, without checking the validity of the statement.

5.1 Example: Simplified DAA

As a running example throughout this section, we consider a simplified version of the Direct Anonymous Attestation (DAA) protocol [20]. The goal of the DAA protocol is to enable the TPM to sign arbitrary messages and to send them to an entity called the verifier in such a way that the verifier will only learn that a valid TPM signed that message, but without revealing the TPM's identity. The DAA protocol is composed of two sub-protocols: the *join protocol* and the *DAA-signing protocol*. The join protocol allows a TPM to obtain a certificate x_{cert} from an entity called the issuer. This certificate is just a signature on the TPM's secret identifier x_f. The DAA-signing protocol enables a TPM to authenticate a message y_m and to prove to the verifier the knowledge of a valid certificate 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 non-interactive zero-knowledge proof:



where the statement S_{daa} is of the form $x_f = \mathsf{check}\ y_{vki}\ x_{cert}\ x_f$. Intuitively, 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.

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 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 to the symbolic representation of zero-knowledge proofs in a process calculus [11, 9]. The user needs to specify: (1) the logical statement of the proof, (2) types for the variables in the statement, and, if desired, (3) an additional logical formula that is conveyed by the proof.

Statements. The statements conveyed by zero-knowledge proofs are positive Boolean formulas. They are formed using equalities between variables and RCF functions applied to variables, as well as conjunctions and disjunctions of such basic statements. The precise syntax of statements is given in Table 7.

Table 7 Syntax of statements

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

Intuitively, a statement is valid if after substituting all variables with the corresponding witnesses and applying all RCF functions to their arguments we obtain a valid Boolean formula. We assume that the RCF functions occurring in the statement have deterministic behaviour[6], i.e., when called twice with the same argument they return the same value. This is made more formal in Figure 8.

Table 8 Semantics of statements $[\![S]\!]_\sigma \in \{\mathsf{true}, \mathsf{false}\}$

$$
\begin{array}{rcl}
[\![x = f\langle\widetilde{T}\rangle\ x_1 \dots x_n]\!]_{\sigma,\phi} & = & \begin{cases} \mathsf{true} & \text{if } \sigma(x) = [\![f\langle\widetilde{T}\rangle]\!]_\phi(\sigma(x_1), \dots, \sigma(x_n)) \\ \mathsf{false} & \text{otherwise} \end{cases} \\
[\![S_1 \wedge S_2]\!]_{\sigma,\phi} & = & [\![S_1]\!]_{\sigma,\phi} \wedge [\![S_2]\!]_{\sigma,\phi} \\
[\![S_1 \vee S_2]\!]_{\sigma,\phi} & = & [\![S_1]\!]_{\sigma,\phi} \vee [\![S_2]\!]_{\sigma,\phi}
\end{array}
$$

Note: $\sigma : \mathsf{Var} \to \mathsf{Val}$ is a substitution assigning closed values to the variables in S and $\phi : \mathsf{Var} \to \mathsf{Val}$ is a value environment mapping variables to closed values (usually functions).

For example, the statement of the zero-knowledge proof in the DAA-signing protocol is $S_{daa} = (x_f = \mathsf{check}\langle T_{vki}\rangle\ y_{vki}\ x_{cert}\ x_f \wedge y_m = id\langle\mathsf{Un}\rangle\ y_m)$. Verification succeeds only if the check function returns x_f when the values of y_{vki}, x_{cert}, and x_f are passed as arguments. The type instantiation using T_{vki} does not affect the semantics, but it is necessary when deriving types. The second conjunct simply mentions the message that is authenticated by the proof and does not affect the semantics of the statement (id is the identity function). We introduce this conjunct only so that y_m appears in the statement. The verification function additionally checks that the value given by the prover to y_{vki} is the same as the one passed as argument by the verifier. We call the variable y_{vki} *matched*. This matching is done inside the verification function (instead of later by the verifier) in order to obtain a stronger type for this function.

Sorts and Types. The values of the variables mentioned in a statement are either made public or are kept secret. We further make a distinction between the public values that are matched by the verifier, and the ones that are obtained as the result of the verification. In the following we assume a function $sort_S$ that for each variable x occurring in the statement 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, public if x has a public value obtained by the verifier after checking the proof, secret if the value of x is not revealed by the proof. For all variables occurring in a statement S the user also needs to provide a type. In the following we assume a function T_S that assigns a type to any variable occurring in S.

In the simplified DAA example, the variables x_f and x_{cert} are secret ($sort_{S_{daa}}(x_f) = sort_{S_{daa}}(x_{cert}) = \mathsf{secret}$), while y_{vki} is matched against the signature verification key of the issuer, which the verifier already knows ($sort_{S_{daa}}(y_{vki}) = \mathsf{matched}$). The payload message y_m is revealed by the proof to anyone including the attacker, so $sort_{S_{daa}}(y_m) = \mathsf{public}$ and $T_{S_{daa}}(y_m) = \mathsf{Un}$. The TPM identifier x_f is given a secret and untainted type: $T_{S_{daa}}(x_f) = \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\} = T_{vki}$. 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

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

of type T_{vki}, so it is given type $\mathsf{Unsealing}_{\mathrm{sign}}\,\langle T_{vki}\rangle$. 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 give the certificate sort public, since the verifier could then distinguish if two consecutive requests come from the same user or not (as in the pseudonymous version of DAA). While we assume that if $sort_S(x) = \mathsf{public}$ or $sort_S(x) = \mathsf{matched}$ then $T_S(x)$ has kind public, the converse does not need to be true.

Additional Logical Formula. The user can additionally specify an arbitrary logical formula over (some of) the variables occurring in the statement. This formula depends on the logic of the protocol, and does not need to follow from the statement, but it has to hold true, in addition to the statement, in the typing environment of the prover. If the statement is strong enough to identify the prover as a honest (type-checked) protocol participant[7], then the additional formula can be safely transmitted to the typing environment of the verifier. For a statement S we denote this additional logical formula by C_S. In the DAA example we have that $C_{S_{daa}} = \mathsf{Send}(x_f, y_m)$, since this predicate does not follow from the statement but it holds true in the typing environment of the (honest) prover .

5.3 Automatic Code Generation

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

5.3.1 Typed Interface

The interface generated for a zero-knowledge statement contains just one function: mkZK. When passed a unit this function returns the three functions implementing the zero-knowledge oracle. This setup is necessary since the three functions share hidden state.

$$\mathsf{mkZK}_S : \mathsf{unit} \to (\mathsf{Create}_S * \mathsf{Verify}_S * \mathsf{Public})$$

The types of the three functions are as follows[8]:

$$\begin{array}{lcl}
\mathsf{Create}_S & = & \tau_S \to \mathsf{Un} \\
 & & \quad \text{where } \tau_S = \mathsf{Un} \vee \sum_{x \in \mathsf{vars}(S)} x : T_S(x).\{C_S\} \\
\mathsf{Public} & = & \mathsf{Un} \to \mathsf{Un} \\
\mathsf{Verify}_S & = & \mathsf{Un} \to (\mathsf{Un} \ \wedge \prod_{y \in \mathsf{matched}(S)} y : T_S(y). \\
 & & \quad \sum_{y \in \mathsf{public}(S)} y : T_S(y).\{\underset{\widetilde{x}=\mathsf{secret}(S)}{\exists \widetilde{x}.} C_S \wedge F(S,E)\} \quad)
\end{array}$$

The function used to create zero-knowledge proofs has type Create_S. It takes as argument all the witnesses of the proof as a tuple, 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 witnesses have the types provided by the user. Additionally, we check that the formula C_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: $\tau_{S_{daa}} = \mathsf{Un} \vee (y_{vki} : \mathsf{Seal}_{\mathrm{sign}}\,\langle T_{vki}\rangle * y_m : \mathsf{Un} * x_f : T_{vki} * x_{cert} : \mathsf{Un} * \{\mathsf{Send}(x_f, y_m)\})$, where $T_{vki} = \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\}$.

[7] Signature proofs of knowledge have this property [20, 39].

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

Table 9 The formula conveyed by a statement $F(S, E)$

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

The function used to read the public witnesses of a zero-knowledge proof has type Public. This function takes as input the sealed zero-knowledge proof of type Un and returns the tuple of public witnesses, also at type Un.

The function used for verifying zero-knowledge proofs has type Verify_S. 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 $\exists \widetilde{x}.C_S \wedge F(S, E)$ holds, where the public and matched variables can appear and the secret variables are existentially quantified. If the prover is a protocol participant then the first conjunct C_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 C_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. This is explained in more detail in Section 5.3.2. The second conjunct, $F(S, E)$, 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 [9]), we use a transformation function F that computes the formula conveyed by the statement. This transformation is straightforward with the exception of function applications, where we use the formulas guaranteed by the dependently-typed cryptographic functions. The formal definition is given in Table 9.

For instance, in the DAA example, we have that

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

The second conjunct is equal to $(y_m = y_m)$, since the identity function is typed to $x : \mathsf{Un} \rightarrow \{y : \mathsf{Un} \mid x = y\}$ in our standard library. As explained in Section 4, we have that $E_{std} \vdash \mathsf{check}\langle T_{vki} \rangle : xvk : \mathsf{Unsealing}_{\mathrm{sign}}\langle T_{vki} \rangle \rightarrow z : \mathsf{Un} \rightarrow x : (T_{vki} \vee \mathsf{Un}) \rightarrow \{y : T_{vki} \mid y = x \wedge \exists SK.SKPair(xvk, SK) \wedge Signed(SK, x, z)\}$. So for the first conjunct after applying the corresponding substitutions we obtain: $(x_f = x_f) \wedge (\exists sk.SKPair(y_{vki}, sk) \wedge Signed(sk, x_f, x_{cert})) \wedge \mathsf{OkTPM}(x_f)$. The predicate $\mathsf{OkTPM}(x_f)$ was obtained from the nested refinement type T_{vki}, according to the definition of *forms* from Section 3. Finally, after removing the trivial equalities we obtain that:

$$F(S_{daa}, E_{std}) = (\exists sk.SKPair(y_{vki}, sk)\ \wedge Signed(sk, x_f, x_{cert})) \wedge \mathsf{OkTPM}(x_f).$$

5.3.2 Implementation

The generated mkZK_S function creates a fresh seal k of type $\tau_S = \mathsf{Un} \vee \sum_{x \in \mathsf{vars}(S)} x : T_S(x).\{C_S\}$. The union type is necessary since the witnesses 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_S and ver_S that return the function for extracting the public witnesses and the zero-knowledge verification function, respectively.

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

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

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.

$pub_S = \lambda k_{unsealing} : \mathsf{Un} \to \tau_S.\lambda z : \mathsf{Un}.$
$\quad \mathsf{let}\ z' = k_{unsealing}\ z\ \mathsf{in}$
$\quad \mathsf{case}\ z'' = z'\ \mathsf{in}$
$\quad \mathsf{let}\ (\widetilde{yz}, \widetilde{x}) = z''\ \mathsf{in}\ (\widetilde{yz})$

The case construct is necessary since τ_S is a union type. In case z' has type Un then the declared return type Un is trivial to justify. In case z' has type $\sum_{x \in \mathsf{vars}(S)} x : T_S(x).\{C_S\}$ we rely on the earlier assumption that all public and matched variables have a public type, in order to give the returned tuple $(\widetilde{y})$ type Un.

The type and the implementation of the ver_S function are more involved. The function inputs the unsealing function $k_{unsealing}$ of type $\mathsf{Un} \to \tau_S$, a candidate zero-knowledge proof z of type Un, and values for the matched variables. Since the type Verify_S contains an intersection type (Un is one of the branches and this makes the type Verify_S public) we use a for construct to introduce this intersection type. If the proof is verified by the attacker we can assume that for all $y' \in \mathsf{matched}(S)$ we have $y' : \mathsf{Un}$ and need to type the return value to Un. On the other hand, if the proof is verified by a protocol participant we can assume that for all $y' \in \mathsf{matched}(S)$ we have $y' : T_S(y')$, and need to give the returned value type $\sum_{y \in \mathsf{public}(S)} y : T_S(y).\{\exists_{\widetilde{x}=\mathsf{secret}(S)} \widetilde{x}.\ C_S \wedge F(S,E)\}$. 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 C_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.

$ver_S = \lambda k_{unsealing} : \mathsf{Un} \to \tau_S.\lambda z : \mathsf{Un}.$
$\quad \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{\mathsf{Un}};\ \widetilde{T_S(y)}\ \mathsf{do}$
$\quad \lambda y'_1 : \alpha_1 \ldots . \lambda y'_n : \alpha_n.$
$\quad\quad (*1*)\ \ \mathsf{let}\ z' = k_{unsealing}\ z\ \mathsf{in}$

Table 10 Converting Decision Trees to Expressions

$$\exp(\mathsf{true}, \sigma) = (\sigma(z_1), \ldots, \sigma(z_n)), \text{ where } \widetilde{z} = \mathsf{public}(S)$$
$$\exp(\mathsf{false}, \sigma) = \mathit{failwith}\ ()$$
$$\exp(\mathsf{if}\ x = f\langle\widetilde{T}\rangle\ x_1 \ldots x_n\ \mathsf{then}\ D_1\ \mathsf{else}\ D_2, \sigma) =$$
$$\mathsf{if}\ \sigma(x) = f\langle\widetilde{T}\rangle\ \sigma(x_1) \ldots \sigma(x_n)\ \mathsf{as}\ y\ \mathsf{then}$$
$$\exp(D_1, \sigma\{y/x\})\ \mathsf{else}\ \exp(D_2, \sigma)$$

Note: Variables y, y_1, and y_2 are always freshly chosen.

$(*2*)$ case $z'' = z'$ in
$(*3*)$ let $(\widetilde{y}, \widetilde{z}, \widetilde{x}) = z''$ in
$(*4*)$ if $(\widetilde{y}) = (\widetilde{y'})$ as $(\widetilde{y''})$ then
$(*5*)$ "$\exp(\mathrm{prime}(S), \{\widetilde{y''}/\widetilde{y}\})$"
else *failwith* ()

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 [21, 48] for the details.

Finally, the decision tree $\mathrm{prime}(S\,[R_1, \ldots, R_n])$ is converted into an RCF expression using a function called exp (Table 10). Other than the decision tree, this function takes as argument a substitution σ that records which is the variable with the strongest type that corresponds to each witness. Initially this substitution is $\{\widetilde{y'}/\widetilde{y}\}$, i.e., it maps the matched variables $\widetilde{y}$ to the values $\widetilde{y'}$ taken as arguments (remember that since $\widetilde{y}$ and $\widetilde{y'}$ were tested for equality in the previous step and $\widetilde{y'}$ have the stronger types). After checking each atomic statement the conversion introduces new variables that stand for some of the witnesses and updates the substitution accordingly. The conversion works as follows. The leaves of the decision tree marked with true are converted into expressions that return the tuple $(\sigma(x_1), \ldots, \sigma(x_n))$, 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 σ to match x to y. The else branch is generated by recursively calling $\exp(D_2, \sigma)$.

In the DAA example the decision tree has a very simple (linear) structure:

if $x_f = \mathsf{check}\langle T_{vki}\rangle\ y_{vki}\ x_{cert}\ x_f$ then
  if $y_m = id\langle\mathsf{Un}\rangle\ y_m)$ then true
else false else false.

This decision tree is used to generate the following verification function:

$ver_{S_{daa}} = \lambda k_{unsealing} : \mathsf{Un} \to \tau_{S_{daa}}.\lambda z : \mathsf{Un}.$
  for α in Un; $\mathsf{Unsealing}_{\mathrm{sign}}\,\langle T_{vki}\rangle$ do $\lambda y'_{vki} : \alpha.$

```
let z' = k_unsealing z in
case z'' = z' in
let (y_vki, y_m, x_f, x_cert) = z'' in
if (y_vki) = (y'_vki) as (y''_vki) then
   if x_f = check⟨T_vki⟩ y''_vki x_cert x_f as x'_f then
      if y_m = id⟨Un⟩ y_m as y'_m then
         (y'_m, ())
      else failwith ()
   else failwith ()
else failwith ()
```

5.3.3 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.

Type-checking the DAA example succeeds, and we illustrate this on the $ver_{S_{daa}}$ function. The for construct requires us to type-check the body of the function twice. The first time the type of the argument y'_{vki} has type Un and we need to give the result $(y'_m, ())$ type Un. This is immediate since y'_m has type $\mathsf{Un} \wedge \mathsf{Un}$ which is equivalent to Un. The second time we type-check the body of the function, the argument y'_{vki} is given type $\mathsf{Unsealing}_{\mathrm{sign}}\langle T_{vki}\rangle$. The type of z' is $\tau_{S_{daa}}$, i.e., a union type where one of the types is Un, so we have again two sub-cases to consider. Intuitively, if z'' is of type Un then the proof comes from the attacker. The application $\mathsf{check}\langle T_{vki}\rangle\ y''_{vki}\ x_{cert}\ x_f$ is typed to $T_{vki} = \{z_f : \mathsf{Private} \mid \mathsf{OkTPM}(z_f)\}$. On the other hand, x_f is obtained by splitting z'' that has type Un, so it also has type Un. However, the types Un and T_{vki} are not disjoint only if false is entailed in the typing environment. This causes the second if to add false to the typing environment, and therefore the return value to have any type, including the one specified in the interface. Intuitively, this means that if the proof comes from the adversary the second if will never succeed since the attacker does not know the secret TPM identifier, so it is safe to return anything on that branch. Finally, we need to consider the case when z'' is of type $(y_{vki} : \mathsf{Seal}_{\mathrm{sign}}\langle T_{vki}\rangle * y_m : \mathsf{Un} * x_f : T_{vki} * x_{cert} : \mathsf{Un} * \{\mathsf{Send}(x_f, y_m)\})$. After splitting z'' the predicate $\mathsf{Send}(x_f, y_m)$ is entailed by the environment, and, since x_f has type T_{vki}, also $\mathsf{OkTPM}(x_f)$ is entailed. The third if adds the equality $y_m = y'_m$, so that also $\mathsf{Send}(x_f, y'_m)$ holds. Since, $\exists x_f.\ \mathsf{Send}(x_f, y'_m) \wedge \mathsf{OkTPM}(x_f)$ is logically entailed we can justify the return type of the function also in this case.

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 C_S is not justified by the statement and the types of the witnesses. In this case ver_S cannot match its defined type.

6 Implementation

We have implemented a complete tool-chain for RCF: 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 performs some amount of type inference, and supports an extended syntax with respect to the one in the paper, including: interfaces, 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 the proof obligations. We use the TPTP syntax for the generated proof obligations, which is a standard supported by most first-order theorem provers [51] (in our experiments we used the E equational prover [47]). We have tested the type-checker on the samples in Appendix **??** and on other simple examples; 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, the code generator for zero-knowledge, 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 2000 lines of code, while the whole tool-chain has over 4500 lines of code. All the tools are available at [10].

7 Conclusions and Future Work

We have presented a general technique for verifying implementations of cryptographic protocols based on zero-knowledge proofs. We developed a tool that automatically generates the symbolic implementation of non-interactive zero-knowledge proofs, starting from a high-level specification of their statements. The security of RCF protocol implementations is verified by a type system combining refinement, union, and intersection types. The analysis is fully automated, efficient, and compositional.

As future work, we plan to investigate the automated generation of concrete cryptographic implementation of zero-knowledge proofs, and thus to complement the generation of symbolic implementations as 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 [24].

Acknowledgments. We thank Cédric Fournet and Andrew D. Gordon for many constructive discussions. Cătălin Hriţcu is supported by a fellowship from Microsoft Research and the International Max Planck Research School for Computer Science.

References

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

[2] 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.

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

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

[5] M. Abadi, B. Blanchet, and C. Fournet. Automated verification of selected equivalences for security protocols. In *Proc. 20th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 331–340. IEEE Computer Society Press, 2005.

[6] 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.

[7] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. 4th ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

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

[9] M. 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.

[10] M. Backes, C. Hriţcu, M. Maffei, and T. Tarrach. Type-checking implementations of protocols based on zero-knowledge proofs. Implementation available at `http://mail-infsec.cs.uni-sb.de:8000/projects/F5/`.

[11] 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.

[12] E. Bangerter, J. Camenisch, S. Krenn, A. Sadeghi, and T. Schneider. Automatic generation of sound zero-knowledge protocols. Accepted at Eurocrypt 2009. IACR Cryptology ePrint Archive: Report 2008/471, Nov. 2008. `http://eprint.iacr.org/`.

[13] 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.

[14] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. Technical Report MSR-TR-2008-118, Microsoft Research, 2008.

[15] 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.

[16] 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.

[17] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 139–152. IEEE Computer Society Press, 2006.

[18] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE Computer Society Press, 2001.

[19] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1998.

[20] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proc. 11th ACM Conference on Computer and Communications Security*, pages 132–145. ACM Press, 2004.

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

[22] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Formal analysis of Kerberos 5. *Theoretical Computer Science*, 367(1):57–87, 2006.

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

[24] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: A secure voting system. In *Proc. 29th IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society Press, 2008.

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

[26] A. Datta, A. Derek, J. Mitchell, and A. Roy. Protocol composition logic (pcl). *Electronic Notes on Theoretical Computer Science*, 172:311–358, 2007.

[27] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13(3):423–482, 2005.

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

[29] S. Doghmi, J. Guttman, and F. Thayer. Searching for shapes in cryptographic protocols. In *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, pages 523–538. Springer-Verlag, 2007.

[30] N. Durgin, J. Mitchell, and D. Pavlovic. A compositional logic for proving security properties of protocols. *Journal of Computer Security*, 11(4):677–721, 2004.

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

[32] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF)*, pages 31–45. IEEE Computer Society Press, 2007.
[33] T. Freeman and F. Pfenning. Refinement types for ML. In *In Programming Language Design and Implementation (PLDI'91)*, pages 268–277. ACM Press, 1991.
[34] J.-Y. Girard. The System F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.
[35] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *6th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2005)*, pages 363–379. Springer, 2005.
[36] A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
[37] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.
[38] G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 18–30. IEEE Computer Society Press, 1997.
[39] L. Lu, J. Han, L. Hu, J. Huai, Y. Liu, and L. M. Ni. Pseudo trust: Zero-knowledge based authentication in anonymous peer-to-peer protocols. In *Proc. 2007 IEEE International Parallel and Distributed Processing Symposium*, page 94. IEEE Computer Society Press, 2007.
[40] J. Morris. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.
[41] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 12(21):993–999, 1978.
[42] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.
[43] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. pages 227–274, 1998.
[44] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
[45] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.
[46] S. Schneider. Security properties and CSP. In *Proc. 17th IEEE Symposium on Security & Privacy*, pages 174–187, 1996.
[47] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
[48] G. Smolka and C. E. Brown. Introduction to computational logic. Lecture Notes, Saarland University, July 2008. Available at `http://www.ps.uni-sb.de/courses/cl-ss08/script/icl.pdf`.
[49] E. Sumii and B. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375(1-3):169–192, 2007.
[50] E. Sumii and B. C. Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4):521–554, 2003.
[51] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
[52] F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. 19th IEEE Symposium on Security & Privacy*, pages 160–171, 1998.
[53] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, 1996.
[54] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. 26th Symposium on Principles of Programming Languages (POPL '99)*, pages 214–227. ACM Press, 1999.

A RCF

A.1 Syntax

Table 11 repeats the syntax of RCF. Additional to the calculus considered in the body of the paper we are considering kind-bounded polymorphism, i.e., bounded polymorphism where the (upper or lower) bound is type Un.

Table 11 Syntax of RCF values and expressions

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

Notation: We use square brackets to denote optional parts.

A.2 Operational Semantics

The operational semantics is standard (Table 12), and (conservatively) extends the one in [13, 14] to the newly introduced constructs. The semantics of the for construct is similar to the one in [25].

A.3 Authorization Logic

We use the same generic definition of an authorization logic, and the same authorization logic instance as in [13, 14]: classical first order-logic with equality where the RCF values are syntactically[9] embedded (FOL/F). In principle we could also use intuitionistic first-order logic as the authorization logic.

[9]Equality between embedded functions is only up to α-conversion.

Table 12 Extensions to the reduction relation of RCF

$$(\Lambda\alpha[::k].A)\langle T\rangle \to A\{T/\alpha\} \qquad \text{(Red Inst)}$$

$$\mathsf{if}\ M = M\ \mathsf{then}\ A\ \mathsf{else}\ B \to A \qquad \text{(Red Eq)}$$

$$\mathsf{if}\ M = M\ \mathsf{as}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B \to A\{M/x\} \qquad \text{(Red Cast Eq)}$$

$$\mathsf{if}\ M = N\ [\mathsf{as}\ x]\ \mathsf{then}\ A\ \mathsf{else}\ B \to B, \mathsf{if}\ M \neq N \qquad \text{(Red Not Eq)}$$

$$\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T};\widetilde{U}\ \mathsf{do}\ A \to \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T};\widetilde{U}\ \mathsf{do}\ A', \mathsf{if}\ A \to A' \qquad \text{(Red For Ctxt)}$$

$$(\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T};\widetilde{U}\ \mathsf{do}\ M)\ N \to \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T};\widetilde{U}\ \mathsf{do}\ (M\ N), \mathsf{if}\ \alpha \notin \mathit{free}(N) \qquad \text{(Red For Appl)}$$

$$\mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T};\widetilde{U}\ \mathsf{do}\ M \to M, \quad \text{if } \{\widetilde{\alpha}\} \cap \mathit{free}(M) = \emptyset \qquad \text{(Red For Remove)}$$

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

B Type System

Table 13 Syntax of Types

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

Notations: Let Un denote the type unit, $\{C\}$ denote the "OK" type $\{x : \mathsf{Un} \mid C\}$, where $x \notin fv(C)$, $\bot$ denote $\{\mathsf{false}\}$, and $\mathsf{Private}$ denote $\bot \to \bot$ (where by $T \to U$ we mean $x : T \to U$ with $x \notin free(U)$). Let $\widetilde{x} : \widetilde{T}$ denote $x_1 : T_1, \ldots, x_n : T_n$ for some n.

Table 14 Judgments

$E \vdash \diamond$	E is syntactically well-formed
$E \vdash T$	in E, type T is syntactically well-formed
$E \models C$	formula C is derivable from E
$E \vdash T :: k$	in E, type T has kind k
$E \vdash T <: U$	in E, type T is a subtype of type U
$E \vdash A : T$	in E, expression A has type T

Table 15 Syntax of Typing Environments

$\mu ::=$		environment entry
	$\alpha[:: k]$	[bounded] type variables
	$\alpha <: \alpha'$	subtyping for type variables ($\alpha \neq \alpha'$)
	$a \updownarrow T$	channel name
	$x : T$	variable
$E ::=$	$\mu_1, \ldots, \mu_n$	environment

Table 16 Well-formed Environment

$$\textsc{Env Empty}\quad \emptyset \vdash \diamond \qquad\qquad \textsc{Type}\quad \frac{E \vdash \diamond \qquad \mathit{free}(T) \subseteq \mathit{dom}(E)}{E \vdash T}$$

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

$$\textsc{Derive}\quad \frac{E \vdash \diamond \qquad \mathit{free}(C) \subseteq \mathit{dom}(E) \qquad \mathit{forms}(E) \models C}{E \models C}$$

Definition:

$$\begin{aligned}
&\mathit{dom}(\alpha[:: k]) = \{\alpha\} \\
&\mathit{dom}(\alpha <: \alpha') = \{\alpha, \alpha'\} \\
&\mathit{dom}(a \updownarrow T) = \{a\} \\
&\mathit{dom}(x : T) = \{x\} \\
&\mathit{dom}(\mu_1, \ldots, \mu_n) = \mathit{dom}(\mu_1) \cup \ldots \cup \mathit{dom}(\mu_n)
\end{aligned}$$

Definition:

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

Definition: $\mathit{tvars}(E) = \{\alpha \mid \alpha \in \mathit{dom}(E)\}$

Table 17 Kinding ($k \in \{\mathsf{pub}, \mathsf{tnt}\}$) $E \vdash T :: k$

Let $\overline{k}$ satisfy $\overline{\mathsf{pub}} = \mathsf{tnt}$ and $\overline{\mathsf{tnt}} = \mathsf{pub}$

$$\textsc{Kind Var}\ \frac{\alpha \in dom(E) \qquad E \models \mathsf{false}}{E \vdash \alpha :: k} \qquad \textsc{Kind Var Bounded}\ \frac{E \vdash \diamond \qquad (\alpha :: k) \in E}{E \vdash \alpha :: k} \qquad \textsc{Kind Unit}\ \frac{E \vdash \diamond}{E \vdash \mathsf{unit} :: k}$$

$$\textsc{Kind Fun}\ \frac{E \vdash T :: \overline{k} \qquad E, x : T \vdash U :: k}{E \vdash (x : T \to U) :: k} \qquad \textsc{Kind Pair}\ \frac{E \vdash T :: k \qquad E, x : T \vdash U :: k}{E \vdash (x : T * U) :: k} \qquad \textsc{Kind Rec}\ \frac{E, \alpha :: k \vdash T :: k}{E \vdash (\mu\alpha.T) :: k}$$

$$\textsc{Kind Refine Pub}\ \frac{E \vdash \{x : T \mid C\} \qquad E \vdash T :: \mathsf{pub}}{E \vdash \{x : T \mid C\} :: \mathsf{pub}} \qquad \textsc{Kind Refine Empty Pub}\ \frac{E \vdash \{x : T \mid C\} \qquad E, x : T \models \neg C}{E \vdash \{x : T \mid C\} :: \mathsf{pub}}$$

$$\textsc{Kind Refine Tnt}\ \frac{E \vdash T :: \mathsf{tnt} \qquad E, x : T \models C}{E \vdash \{x : T \mid C\} :: \mathsf{tnt}} \qquad \textsc{Kind Top Tnt}\ \frac{E \vdash \diamond}{E \vdash \top :: \mathsf{tnt}} \qquad \textsc{Kind Top Pub}\ \frac{E \models \mathsf{false}}{E \vdash \top :: \mathsf{pub}} \qquad \textsc{Kind And Pub1}\ \frac{E \vdash T :: \mathsf{pub}}{E \vdash T \wedge U :: \mathsf{pub}}$$

$$\textsc{Kind And Pub2}\ \frac{E \vdash U :: \mathsf{pub}}{E \vdash T \wedge U :: \mathsf{pub}} \qquad \textsc{Kind And Tnt}\ \frac{E \vdash T :: \mathsf{tnt} \qquad E \vdash U :: \mathsf{tnt}}{E \vdash T \wedge U :: \mathsf{tnt}} \qquad \textsc{Kind Or Pub}\ \frac{E \vdash T :: \mathsf{pub} \qquad E \vdash U :: \mathsf{pub}}{E \vdash T \vee U :: \mathsf{pub}}$$

$$\textsc{Kind Or Tnt1}\ \frac{E \vdash T :: \mathsf{tnt}}{E \vdash T \vee U :: \mathsf{tnt}} \qquad \textsc{Kind Or Tnt2}\ \frac{E \vdash U :: \mathsf{tnt}}{E \vdash T \vee U :: \mathsf{tnt}} \qquad \textsc{Kind Forall}\ \frac{E, \alpha \vdash T :: k}{E \vdash \forall\alpha.T :: k} \qquad \textsc{Kind Forall Bounded}\ \frac{E, \alpha :: k' \vdash T :: k}{E \vdash \forall\alpha :: k'.T :: k}$$

Derived rules:

$$\textsc{Kind Ok Public}\ \frac{E \vdash \{C\}}{E \vdash \{C\} :: \mathsf{pub}} \qquad \textsc{Kind Ok Tainted}\ \frac{E \vdash \{C\} \qquad E \models C}{E \vdash \{C\} :: \mathsf{tnt}}$$

Table 18 Subtyping $E \vdash T <: U$

$$\textsc{Sub Refl}\ \frac{E \vdash T \qquad \mathit{tvars}(E) \cap \mathit{free}(T) = \emptyset}{E \vdash T <: T} \qquad \textsc{Sub Public Tainted}\ \frac{E \vdash T :: \mathsf{pub} \qquad E \vdash U :: \mathsf{tnt}}{E \vdash T <: U}$$

$$\textsc{Sub Fun}\ \frac{E \vdash T' <: T \qquad E, x : T' \vdash U <: U'}{E \vdash (x : T \to U) <: (x : T' \to U')} \qquad \textsc{Sub Pair}\ \frac{E \vdash T <: T' \qquad E, x : T \vdash U <: U'}{E \vdash (x : T * U) <: (x : T' * U')}$$

$$\textsc{Sub Var}\ \frac{E \vdash \diamond \qquad \alpha <: \alpha' \in E}{E \vdash \alpha <: \alpha'} \qquad \textsc{Sub Rec}\ \frac{E, \alpha <: \alpha' \vdash T <: T' \qquad \alpha \notin \mathit{free}(T') \qquad \alpha' \notin \mathit{free}(T')}{E \vdash \mu\alpha.T <: \mu\alpha'.T}$$

$$\textsc{Sub Refine Left}\ \frac{E \vdash \{x : T \mid C\} \qquad E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'} \qquad \textsc{Sub Refine Empty}\ \frac{E \vdash \{x : T \mid C\} \qquad E, x : T \models \neg C}{E \vdash \{x : T \mid C\} <: T'}$$

$$\textsc{Sub Refine Right}\ \frac{E \vdash T <: T' \qquad E, x : T \models C}{E \vdash T <: \{x : T' \mid C\}} \qquad \textsc{Sub Top}\ \frac{E \vdash T}{E \vdash T <: \top} \qquad \textsc{Sub And LB1}\ \frac{E \vdash U \qquad E \vdash T <: T'}{E \vdash T \wedge U <: T'}$$

$$\textsc{Sub And LB2}\ \frac{E \vdash T \qquad E \vdash U <: T'}{E \vdash T \wedge U <: T'} \qquad \textsc{Sub And Greatest}\ \frac{E \vdash T <: T_1 \qquad E \vdash T <: T_2}{E \vdash T <: T_1 \wedge T_2} \qquad \textsc{Sub Or Smallest}\ \frac{E \vdash T_1 <: T \qquad E \vdash T_2 <: T}{E \vdash T_1 \vee T_2 <: T}$$

$$\textsc{Sub Or UB1}\ \frac{E \vdash U \qquad E \vdash T' <: T}{E \vdash T' <: T \vee U} \qquad \textsc{Sub Or UB2}\ \frac{E \vdash T \qquad E \vdash T' < U}{E \vdash T' <: T \vee U} \qquad \textsc{Sub Forall}\ \frac{E, \alpha \vdash T <: U}{E \vdash \forall\alpha.T <: \forall\alpha.U}$$

$$\textsc{Sub Forall Bounded}\ \frac{E, \alpha :: k \vdash T <: U}{E \vdash \forall\alpha :: k.T <: \forall\alpha :: k.U}$$

Derived rules:

$$\textsc{Sub Refine}\ \frac{E \vdash T <: T' \qquad E, x : \{x : T \mid C\} \models C'}{E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}} \qquad \textsc{Sub Ok}\ \frac{E, \{C\} \models C'}{E \vdash \{C\} <: \{C'\}}$$

Table 19 More subtyping rules $E \vdash T <: U$

$$\textsc{Sub Dist Int-Arr}\quad E \vdash (x : T \to U_1) \wedge (x : T \to U_2) <: (x : T \to U_1 \wedge U_2)$$

Note: The other direction of these rules can be already obtained as a derived rule.

Table 20 Rules for Values $E \vdash M : T$

$$\textsc{Val Var}\ \frac{E \vdash \diamond \qquad (x : T) \in E}{E \vdash x : T} \qquad \textsc{Val Unit}\ \frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}} \qquad \textsc{Val Fun}\ \frac{E, x : T \vdash A : U}{E \vdash \lambda x : T.A : (x : T \to U)}$$

$$\textsc{Val Pair}\ \frac{E \vdash M : T \qquad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (x : T * U)} \qquad \textsc{Val Fold}\ \frac{E \vdash M : T\{\mu\alpha.T/\alpha\} \qquad E \vdash \mu\alpha.T}{E \vdash \mathsf{fold}\ M : \mu\alpha.T}$$

$$\textsc{Val Refine}\ \frac{E \vdash M : T \qquad E \models C\{M/x\}}{E \vdash M : \{x : T \mid C\}} \qquad \textsc{Val Poly}\ \frac{E, \alpha \vdash A : T}{E \vdash \Lambda\alpha.A : \forall\alpha.T} \qquad \textsc{Val Poly Bounded}\ \frac{E, \alpha :: k \vdash A : T}{E \vdash \Lambda\alpha :: k.A : \forall\alpha :: k.T}$$

Derived rules:

$$\textsc{Val Ok}\ \frac{E \models C}{E \vdash () : \{C\}}$$

Table 21 Formula Extraction $\overline{A}$

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

$$\overline{A} = \mathsf{true}, \text{ otherwise}$$

Table 22 Logical Characterization of Type Disjointness $\mathsf{non\text{-}disj}(T_1, T_2)$

$$\mathsf{non\text{-}disj}(\mathsf{Un}, \mathsf{Private}) = \mathsf{non\text{-}disj}(\mathsf{Private}, \mathsf{Un}) = \mathsf{false}$$
$$\mathsf{non\text{-}disj}(\mathsf{Un}, \{x : \mathsf{Private} \mid C\}) = \mathsf{non\text{-}disj}(\{x : \mathsf{Private} \mid C\}, \mathsf{Un}) = \mathsf{false}$$
$$\mathsf{non\text{-}disj}(T, U) = \mathsf{true}, \text{ otherwise}$$

Property: If there exists N so that $E \vdash N : T$ and $E \vdash N : U$ then $E \models \mathsf{non\text{-}disj}(T, U)$.

Table 23 Rules for Expressions $E \vdash A : T$

$$\textsc{Exp Subsum}\ \frac{E \vdash A : T \qquad E \vdash T <: T'}{E \vdash A : T'} \qquad \textsc{Exp Appl}\ \frac{E \vdash M : (x : T \to U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\}}$$

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

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

$$\textsc{Exp Unfold}\ \frac{E \vdash M : \mu\alpha.T}{E \vdash \mathsf{unfold}\ M : T\{\mu\alpha.T/\alpha\}} \qquad \textsc{Exp Assume}\ \frac{E \vdash \diamond \qquad \mathit{free}(C) \subseteq \mathit{dom}(E)}{E \vdash \mathsf{assume}\ C : \{\_ : \mathsf{unit} \mid C\}} \qquad \textsc{Exp Assert}\ \frac{E \models C}{E \vdash \mathsf{assert}\ C : \mathsf{unit}}$$

$$\textsc{Exp Let}\ \frac{E \vdash A : T \qquad E, x : T \vdash B : U \qquad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A\ \mathsf{in}\ B : U} \qquad \textsc{Exp Res}\ \frac{E, a \updownarrow T \vdash A : U \qquad a \notin fn(U)}{E \vdash (\nu a \updownarrow T)A : U}$$

$$\textsc{Exp Send}\ \frac{E \vdash M : T \qquad (a \updownarrow T) \in E}{E \vdash a!M : \mathsf{unit}} \qquad \textsc{Exp Recv}\ \frac{E \vdash \diamond \qquad (a \updownarrow T) \in E}{E \vdash a? : T}$$

$$\textsc{Exp Fork}\ \frac{E, \_ : \{\overline{A_2}\} \vdash A_1 : T \qquad E, \_ : \{\overline{A_1}\} \vdash A_2 : U}{E \vdash (A_1 \Rsh A_2) : U} \qquad \textsc{Exp Inst}\ \frac{E \vdash A : \forall\alpha.U \qquad E \vdash T}{E \vdash A\langle T\rangle : U\{T/\alpha\}}$$

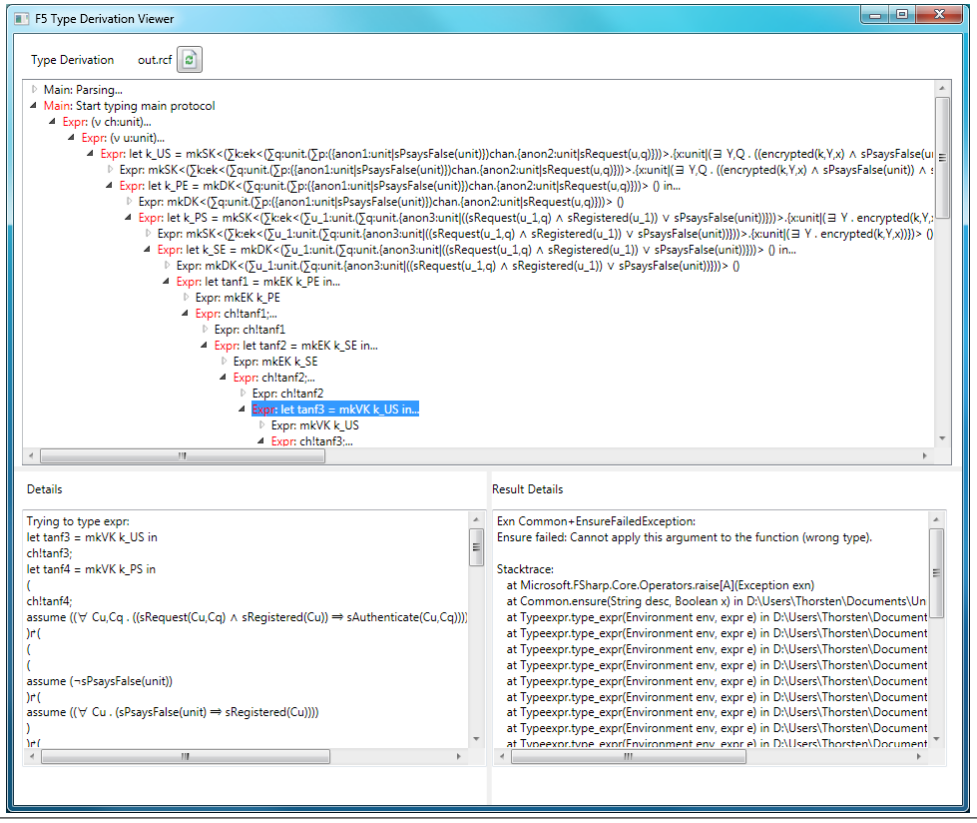
$$\textsc{Exp Inst Bounded}\ \frac{E \vdash A : \forall\alpha :: k.U \qquad E \vdash T :: k}{E \vdash A\langle T\rangle : U\{T/\alpha\}} \qquad \textsc{Exp Case}\ \frac{E \vdash M : T_1 \vee T_2 \qquad E, x : T_1 \vdash A : U \qquad E, x : T_2 \vdash A : U}{E \vdash \mathsf{case}\ x = M\ \mathsf{in}\ A : U}$$

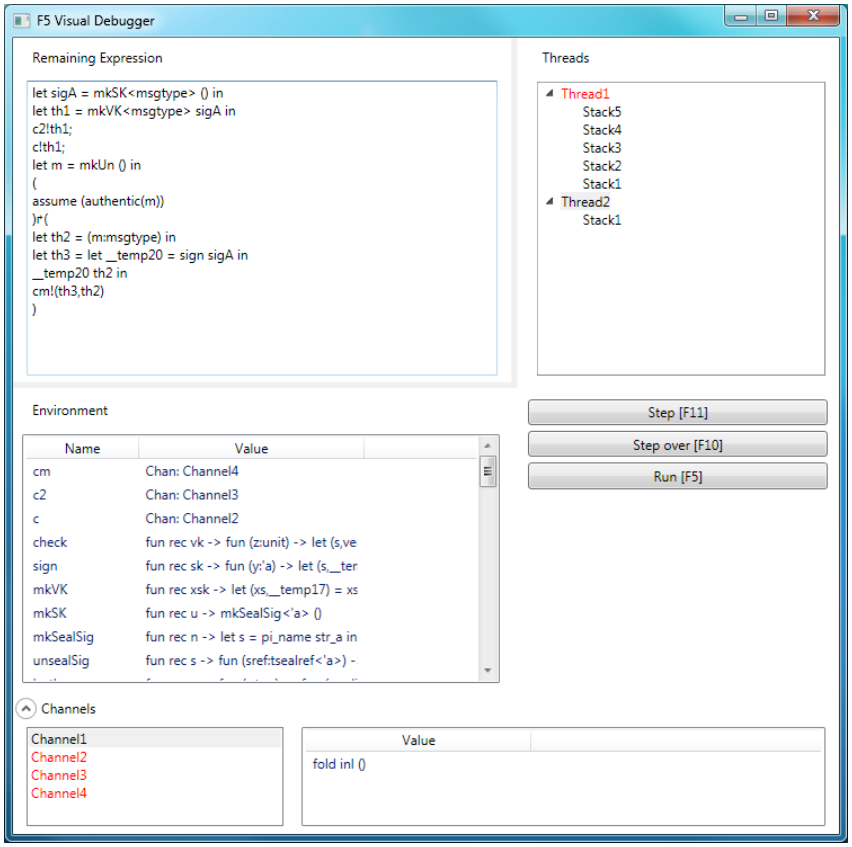
$$\textsc{Exp For}\ \frac{i \in \{1, 2\} \qquad E \vdash A\{\widetilde{T^i}/\widetilde{\alpha}\} : U}{E \vdash \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T^1}; \widetilde{T^2}\ \mathsf{do}\ A : U} \qquad \textsc{Exp And}\ \frac{E \vdash A : T \qquad E \vdash A : U}{E \vdash A : T \wedge U}$$

Derived rule:

$$\frac{E \vdash A\{\widetilde{T}/\widetilde{\alpha}\} : V_1 \qquad E \vdash A\{\widetilde{U}/\widetilde{\alpha}\} : V_2}{E \vdash \mathsf{for}\ \widetilde{\alpha}\ \mathsf{in}\ \widetilde{T}; \widetilde{U}\ \mathsf{do}\ A : V_1 \wedge V_2}$$

C Tools


F5 Type Derivation Viewer
Type Derivation
out.rcf
Main: Parsing...
Main: Start typing main protocol
Expr: (v ch:unit)...
Expr: (v u:unit)...
Expr: let k_US = mkSK<(∑k:ek<(∑q:unit.(∑p:({anon1:unit|sPsaysFalse(unit)})chan.{anon2:unit|sRequest(u,q)}))>.{x:unit|(∃ Y,Q . ((encrypted(k,Y,x) ∧ sPsaysFalse(unit)) ∧
Expr: mkSK<(∑k:ek<(∑q:unit.(∑p:({anon1:unit|sPsaysFalse(unit)})chan.{anon2:unit|sRequest(u,q)}))>.{x:unit|(∃ Y,Q . ((encrypted(k,Y,x) ∧ sPsaysFalse(unit)) ∧
Expr: let k_PE = mkDK<(∑q:unit.(∑p:({anon1:unit|sPsaysFalse(unit)})chan.{anon2:unit|sRequest(u,q)}))> () in...
Expr: mkDK<(∑q:unit.(∑p:({anon1:unit|sPsaysFalse(unit)})chan.{anon2:unit|sRequest(u,q)}))> ()
Expr: let k_PS = mkSK<(∑k:ek<(∑u_1:unit.(∑q:unit.{anon3:unit|((sRequest(u_1,q) ∧ sRegistered(u_1)) ∨ sPsaysFalse(unit))}))>.{x:unit|(∃ Y . encrypted(k,Y,
Expr: mkSK<(∑k:ek<(∑u_1:unit.(∑q:unit.{anon3:unit|((sRequest(u_1,q) ∧ sRegistered(u_1)) ∨ sPsaysFalse(unit))}))>.{x:unit|(∃ Y . encrypted(k,Y,x))})> ()
Expr: let k_SE = mkDK<(∑u_1:unit.(∑q:unit.{anon3:unit|((sRequest(u_1,q) ∧ sRegistered(u_1)) ∨ sPsaysFalse(unit))}))> () in...
Expr: mkDK<(∑u_1:unit.(∑q:unit.{anon3:unit|((sRequest(u_1,q) ∧ sRegistered(u_1)) ∨ sPsaysFalse(unit))}))> ()
Expr: let tanf1 = mkEK k_PE in...
Expr: mkEK k_PE
Expr: ch!tanf1;...
Expr: ch!tanf1
Expr: let tanf2 = mkEK k_SE in...
Expr: mkEK k_SE
Expr: ch!tanf2;...
Expr: ch!tanf2
Expr: let tanf3 = mkVK k_US in...
Expr: mkVK k_US
Expr: ch!tanf3;...
Details
Trying to type expr:
let tanf3 = mkVK k_US in
ch!tanf3;
let tanf4 = mkVK k_PS in
(
ch!tanf4;
assume ((∀ Cu,Cq . ((sRequest(Cu,Cq) ∧ sRegistered(Cu)) ⇒ sAuthenticate(Cu,Cq))))
)r(
(
(
assume (¬sPsaysFalse(unit))
)r(
assume ((∀ Cu . (sPsaysFalse(unit) ⇒ sRegistered(Cu))))
)
)r(
Result Details
Exn Common+EnsureFailedException:
Ensure failed: Cannot apply this argument to the function (wrong type).
Stacktrace:
at Microsoft.FSharp.Core.Operators.raise[A](Exception exn)
at Common.ensure(String desc, Boolean x) in D:\Users\Thorsten\Documents\Un
at Typeexpr.type_expr(Environment env, expr e) in D:\Users\Thorsten\Document
at Typeexpr.type_expr(Environment env, expr e) in D:\Users\Thorsten\Document
at Typeexpr.type_expr(Environment env, expr e) in D:\Users\Thorsten\Document
at Typeexpr.type_expr(Environment env, expr e) in D:\Users\Thorsten\Document
at Typeexpr.type_expr(Environment env, expr e) in D:\Users\Thorsten\Document
at Typeexpr.type_expr(Environment env, expr e) in D:\Users\Thorsten\Document
at Typeexpr.type_expr(Environment env, expr e) in D:\Users\Thorsten\Document


F5 Visual Debugger
Remaining Expression
let sigA = mkSK<msgtype> () in
let th1 = mkVK<msgtype> sigA in
c2!th1;
c!th1;
let m = mkUn () in
(
assume (authentic(m))
)r(
let th2 = (m:msgtype) in
let th3 = let __temp20 = sign sigA in
__temp20 th2 in
cm!(th3,th2)
)
Threads
Thread1
Stack5
Stack4
Stack3
Stack2
Stack1
Thread2
Stack1
Environment
Name
Value
cm
Chan: Channel4
c2
Chan: Channel3
c
Chan: Channel2
check
fun rec vk -> fun (z:unit) -> let (s,ve
sign
fun rec sk -> fun (y:'a) -> let (s,__ter
mkVK
fun rec xsk -> let (xs,__temp17) = xs
mkSK
fun rec u -> mkSealSig<'a> ()
mkSealSig
fun rec n -> let s = pi_name str_a in
unsealSig
fun rec s -> fun (sref:tsealref<'a>) -
Step [F11]
Step over [F10]
Run [F5]
Channels
Channel1
Channel2
Channel3
Channel4
Value
fold inl ()