A Certifying Compiler for Zero-Knowledge Proofs of Knowledge Based on Σ-Protocols*

(Full Version)

José Bacelar Almeida[1], Endre Bangerter[2], Manuel Barbosa[1], Stephan Krenn[3], Ahmad-Reza Sadeghi[4], Thomas Schneider[4]

[1] Universidade do Minho, Portugal
{jba,mbb}@di.uminho.pt
[2] Bern University of Applied Sciences, Biel-Bienne, Switzerland
endre.bangerter@jdiv.org
[3] Bern University of Applied Sciences, Biel-Bienne, Switzerland, and University of Fribourg, Switzerland
stephan.krenn@bfh.ch
[4] Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
{ahmad.sadeghi,thomas.schneider}@trust.rub.de

Abstract. Zero-knowledge proofs of knowledge (ZK-PoK) are important building blocks for numerous cryptographic applications. Although ZK-PoK have very useful properties, their real world deployment is typically hindered by their significant complexity compared to other (non-interactive) crypto primitives. Moreover, their design and implementation is time-consuming and error-prone.

We contribute to overcoming these challenges as follows: We present a comprehensive specification language and a certifying compiler for ZK-PoK protocols based on Σ-protocols and composition techniques known in literature. The compiler allows the fully automatic translation of an abstract description of a proof goal into an executable implementation. Moreover, the compiler overcomes various restrictions of previous approaches, e.g., it supports the important class of exponentiation homomorphisms with hidden-order co-domain, needed for privacy-preserving applications such as idemix. Finally, our compiler is certifying, in the sense that it automatically produces a formal proof of security (soundness) of the compiled protocol (currently covering special homomorphisms) using the Isabelle/HOL theorem prover.

Key words: Zero-Knowledge, Protocol Compiler, Formal Verification

1 Introduction

A zero-knowledge proof of knowledge (ZK-PoK) is a two-party protocol between a prover and a verifier, which allows the prover to convince the verifier that he knows a secret value that satisfies a given relation (*proof of knowledge* or *soundness property*), without the verifier being able to learn anything about the secret (*zero-knowledge property*). For a formal definition we refer to [17]. Fundamental results show that there are ZK-PoK for all languages in NP [47]. The corresponding protocols are of theoretical relevance, but are much too inefficient to be used in practice.

* This work was in part funded by the European Community's Seventh Framework Programme (FP7) under grant agreement no. 216499. An extended abstract of this work is given in [2].

Beside these generic protocols of mainly theoretical interest there are various protocols which are efficient enough for real world use. Essentially, all these practically relevant ZK-PoK protocols are based on the so called Σ-protocols. What is typically being proved using a basic Σ-protocol is the knowledge of a preimage under a homomorphism (e.g., a secret discrete logarithm). These preimage proofs can then be combined to considerably more complex protocols. In fact, many systems in applied cryptography use such proofs as building blocks. Examples include voting schemes [50,52], biometric authentication [18,54], group signatures [29], interactive verifiable computation [33], e-cash [24,31] and secure multiparty computation [56].

While many of these applications only exist on specification level, a direction of applied research has produced first systems using ZK-PoKs that are deployed in the real world. The probably most prominent example is *Direct Anonymous Attestation* (DAA) [27], a privacy enhancing mechanism for remote authentication of computing platforms, which was adopted by the Trusted Computing Group, an industry consortium of many IT enterprises. Another example is the *idemix* anonymous credential system [30], which IBM released into the Eclipse Higgins project, an open source effort dedicated to developing software for user-centric identity management.

Up to now, design, implementation and verification of the formal cryptographic security properties (i.e., zero-knowledge and soundness) as well as code security properties (e.g., security against buffer overflows, side channel vulnerabilities, etc.) is done "by hand". In fact, past experiences, e.g., made when realizing DAA and idemix, have shown that this is a time consuming and error prone task. This is certainly due to the fact that ZK-PoK are considerably more complex than other non-interactive crypto primitives such as encryption schemes.

In particular, the soundness property needs to be proved for each ZK-PoK protocol from scratch. The proofs are often not inherently complex, but nevertheless require an intricate knowledge of the techniques being used. This is obviously a major hurdle in the real world adoption of ZK-PoK, since even experts in the field are not immune to protocol design errors. In fact, minor flaws in protocol designs [3,11,45] (which can be fixed easily once detected) can lead to serious security flaws [36,41,55].

In this paper we describe languages, a compiler and integrated tools that support and partially automate the design, implementation and formal verification of ZK-PoK based on Σ-protocols. The goal of our research is to overcome the difficulties mentioned concerning the design and implementation of ZK-PoK, and thus to bring ZK-PoK to practice by making them accessible to crypto and security engineers.

1.1 Our Contributions

In a nutshell, we present a toolbox that takes an abstract description of the proof goal[5] of a ZK-PoK as input, and produces a provably sound implementation of a suitable protocol in the C language.

[5] By proof goal, we refer to *what a prover wants to demonstrate in zero-knowledge*. For instance, the proof goal can be to prove knowledge of a discrete logarithm.

More precisely, we extend previous directions with the following functionalities of practical and theoretical relevance:

- We present a comprehensive protocol specification language and compiler which support most relevant Σ-protocols and composition techniques found in the literature, including basic protocols for proving knowledge in arbitrary groups, `AND` and `OR` compositions, and techniques for proving linear relations among secret pre-images (e.g., equality of two discrete logarithms). A comprising summary of these techniques can be found in [70].
  Examples of protocols that can be automatically generated by our compiler include [1,18,24,27,29,32,30,31,33,50,52,54,56,57,66,68].
- Our compiler also absorbs certain design-level decisions. For instance, it automatically chooses certain security parameters and intervals used in the protocols to assert the statistical zero-knowledge property of discrete log proofs in hidden order groups. It thus eliminates the potential of security vulnerabilities resulting from inconsistent parameter choices. Further, the compiler has capabilities to automatically rewrite the proof goal to reduce the complexity of the generated protocol.
- Last but not least, our compiler partially alleviates the implementor from the responsibility to establish a theoretical security guarantee for the protocol, by producing a formal proof of the theoretical soundness[6] property. Technically, the compiler produces a *certificate* that the protocol generated by the compiler fulfills its specification. The validity of the certificate is then formally verified by the Isabelle/HOL formal theorem prover [65]. That is, our tool can be seen as a *certifying* compiler. This formal verification component currently only supports a subset of the protocols for which our compiler can generate code. Yet, it already covers a considerable class of applications, such as [24,66,68].

Related Work. Compiler based (semi-)automatic generation of cryptographic protocols has attracted considerable research interest recently, for instance in the field of multi-party computations [42,58,59].

A first prototype ZK-PoK compiler was developed in [28,34] and extended within the CACE project [10,9]. Yet, this compiler offers no optimization or verification functionalities and can only handle a subset of the proof goals supported by our compiler whose architecture was presented in [13].

Very recently, Meiklejohn et al. [60] presented another ZK-PoK compiler for specific applications such as e-cash. To maximize efficiency, their tool generates protocols which exploit precomputations, a feature which is not yet supported by our compiler. However, our compiler provides a substantially broader class of proofs goals such as Or-compositions, and homomorphisms such as RSA [67]. Further, formal verification is left as an "interesting area of study" in [60].

[6] The soundness property is arguably the most relevant security property for many practical applications of ZK-PoK, as it essentially establishes that it is infeasible to prove an invalid knowledge claim. However, our tool is currently being expanded to cover other relevant security properties, namely the zero-knowledge property.

Symbolic models that are suitable for expressing and formally analyzing protocols that rely on ZK protocols as building blocks were presented in [5,16]. In [5] the first mechanized analysis framework for such protocols was proposed by extending the automatic tool ProVerif [23]. The work in [4] proposed an alternative solution to the same problem based on a type-based mechanism. Our work does not overlap with these contributions, and can be seen as complementary. The previous frameworks assume that the underlying ZK-PoK components are secure under adequate security models in order to prove the security of higher level protocols. We work at a lower level and focus on proving that specific ZK-PoK protocols generated by our compiler satisfy the standard computational security model for this primitive. Recent results in establishing the computational soundness of ZK-PoK-aware symbolic analysis can be found in [6]. Currently, we do not establish a connection between the security properties offered by the ZK-PoK protocols produced by our compiler and the level of security required to enable the application of computational soundness results.

We follow a recent alternative approach to obtaining computational security guarantees through formal methods: directly transposing provable security arguments to mechanized proof tools. This allows to deal directly with the intricacies of security proofs, but the potential for mechanization is yet to match that of symbolic analysis. In this aspect, our work shares some of its objectives with parallel work by Barthe et al. [15] describing the formalization of the theory of ZK-PoK in the Coq-based CertiCrypt tool [14]. This formalization includes proofs for the general theorems that establish the completeness, soundness and special honest-verifier ZK properties for Σ-protocols based on homomorphisms. Proving that a concrete protocol satisfies this set of properties can then be achieved by instantiating these general proofs with concrete homomorphisms. Although not completely automatic, this requires relatively small interaction with the user. In this work we provide further evidence that the construction of computational security proofs over mechanized proof tools can be fully automatic. The catch is that our verification component is highly specialized for (a specific class of) ZK-PoK and relies on in-depth knowledge on how the protocol was constructed.

Our work is also related to the formal security analysis of cryptographic protocol implementations. A tool for the analysis of cryptographic code written in C is proposed in [48]. In [19,20], approaches for extracting models from protocol implementations written in F#, and automatically verifying these models by compilation to symbolic models (resp. computational models) in ProVerif [21] (resp. CryptoVerif [22]), can be found. As above, the latter works target higher level protocols such as TLS that use cryptographic primitives as underlying components. Furthermore, the static cryptographic library that implements these primitives must be trusted by assumption. Our work can be seen as a first step towards a tool to automatically extend such a *trusted computing base* when ZK-PoK protocols for different goals are required.

Structure of this Document. In §2 we recap the theoretical framework used by our compiler, which we present in §3. Finally, the formal verification infrastructure is explained in §4.

2 Preliminaries

We first recap some basic notation and theory underlying ZK-PoK .

2.1 Notation

By $s \in_R \mathcal{S}$ we denote the uniform random choice of an element s from set $\mathcal{S}$. The order of a group $\mathcal{G}$ is denoted by $\operatorname{ord}\mathcal{G}$. Finally, $\operatorname{minDiv}(a)$ is the smallest prime dividing an integer a.

We use the notation from [35] for specifying ZK-PoK. A term like

$$\mathsf{ZPK}\left[(\chi_1, \chi_2) : y_1 = \phi_1(\chi_1) \quad \wedge \quad y_2 = \phi_2(\chi_2) \quad \wedge \quad \chi_1 = a\chi_2\right]$$

means "*zero-knowledge proof of knowledge of values* x_1, x_2 *such that* $y_1 = \phi_1(x_1)$, $y_2 = \phi_2(x_2)$, *and* $x_1 = ax_2$". Variables of which knowledge is proved are denoted by Greek letters, whereas all other quantities (known to both parties) are denoted by Latin letters. Note that this notation specifies a *proof goal* rather than a protocol: it describes what has to be proved, but there may be various, differently efficient protocols to do so.

We call a term like $y = \phi(x)$ in the proof goal an *atomic predicate*. A *predicate* is the composition of atomic predicates and predicates using arbitrary many (potentially none) boolean junctors And ($\wedge$) and Or ($\vee$).

2.2 Σ-Protocols as ZK-PoK Protocols

Most practical ZK-PoK are based on *Σ-protocols*. Given probabilistic polynomial time algorithms $\mathsf{P}_1, \mathsf{P}_2, \mathsf{V}$, they have the following form: to prove knowledge of a secret x satisfying a relation with some public y, the prover first sends a *commitment* $t := \mathsf{P}_1(x, y)$ to the verifier, who then draws a random *challenge* c from a predefined *challenge set* $\mathcal{C}$. Receiving c, the prover computes a *response* $s := \mathsf{P}_2(x, y, c)$. Now, if $\mathsf{V}(t, c, s, y) = \mathsf{true}$, the verifier accepts the proof, otherwise it rejects. Whenever the verifier accepts, we call (t, c, s) an *accepting protocol transcript*.

Formally, for the protocol to be a proof of knowledge with knowledge error κ, there must be an algorithm E' satisfying the following: whenever a (potentially malicious) prover can make the verifier accept with probability $\varepsilon > \kappa$, E' can extract x from the prover in a number of steps proportional to $(\varepsilon - \kappa)^{-1}$ [17]. For Σ-protocols, this boils down to the existence of an efficient *knowledge extractor* E, which takes as inputs two accepting protocol transcripts $(t, c', s'), (t, c'', s'')$ with $c' \neq c''$, and y, and outputs a value x' satisfying the relation [37,40].

A Σ-protocol satisfies the ZK property, if there is an efficient *simulator* S, taking c, y as inputs, and outputting tuples that are indistinguishable from real accepting protocol transcripts with challenge c [37,40].

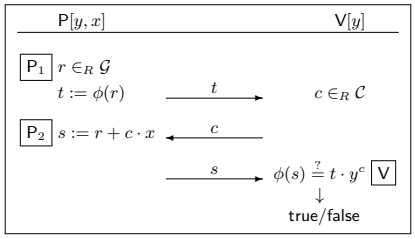

Fig. 1: The Σ^ϕ-protocol for performing $\mathsf{ZPK}[(\chi) : y = \phi(\chi)]$.

2.3 Proving Atomic Predicates

We next summarize the basic techniques for proving atomic predicates.

The Σ^ϕ-Protocol. The Σ^ϕ-protocol allows to efficiently prove knowledge of preimages under homomorphisms with a finite domain [68,51]. For instance, it can be used to prove knowledge of the content of ciphertexts under the Cramer/Shoup [39] or the RSA [67] encryption schemes, and many others [63,64,43,53,46]. Also, it can be used for all homomorphisms mapping into a group over elliptic curves.

The protocol flow, as well as inputs and outputs of both parties, are shown in Fig. 1. The Σ^ϕ-protocol is a ZK-PoK with knowledge error $\kappa = 1/c_{\mathsf{max}}$ for suitably chosen challenge set $\mathcal{C} = \{0, \ldots, c_{\mathsf{max}} - 1\}$. Yet, while $c_{\mathsf{max}} = 2$ can safely be used for any homomorphism ϕ, the maximal value of c_{max} heavily depends on ϕ. We thus briefly describe the theory needed for choosing $\mathcal{C}$ correctly.

Although finding a preimage x for a given $y = \phi(x)$ is usually hard for homomorphisms ϕ used in cryptography, it is often easy to find the preimage of a known power of y. Let, for example, the order q of the domain of ϕ be known: given $y = \phi(x)$ we can efficiently compute a preimage of y^q, as we have $y^q = 1 = \phi(0)$. Similarly, for homomorphisms $\phi : \mathcal{H} \times \mathcal{G} \to \mathcal{H} : (a, b) \mapsto a^e \cdot \psi(b)$ (as used in [67,63,64,43]) a preimage of y^e is given by $(y, 0)$. It turns out that this *special property* is crucial for reaching high efficiency in the Σ^ϕ-protocol.

Definition 1 (Special Homomorphisms [37]). *A homomorphism $\phi : \mathcal{G} \to \mathcal{H}$ is* special*, if for any image $y \in \mathcal{H}$ a pair $(u, v) \in \mathcal{G} \times \mathbb{Z} \setminus \{0\}$ satisfying $\phi(u) = y^v$ can efficiently be computed, where the value v has to be the same for all y. We call (u, v) a* pseudo preimage *of y under ϕ, and v the* special exponent *of ϕ.*

Theorem 2 (Knowledge Error of the Σ^ϕ-Protocol [37]). *Let ϕ be a homomorphism with finite domain. Then the Σ^ϕ-protocol using $\mathcal{C} = \{0, \ldots, c_{\mathsf{max}} - 1\}$ is a ZK-PoK with knowledge error $\kappa = 1/c_{\mathsf{max}}$, if either $c_{\mathsf{max}} = 2$, or ϕ is special with special exponent v and $c_{\mathsf{max}} \leq \mathrm{minDiv}(v)$.*

The Σ^{GSP}- and the Σ^{exp}-Protocols. The practically important class of exponentiation homomorphisms with hidden-order codomain (e.g., $\phi : \mathbb{Z} \to \mathbb{Z}_n^* : a \mapsto g^a$, where n is an RSA modulus, and g generates the quadratic residues modulo n) cannot be treated with the Σ^ϕ-protocol.

Two Σ-protocols for such homomorphisms can be found in the literature. The Σ^{GSP}-protocol generalizes the Σ^{ϕ}-protocol to the case of infinite domains (i.e., $\mathcal{G} = \mathbb{Z}$), and can be used very efficiently if assumptions on the homomorphism ϕ are made [45,41]. On the other hand, the so-called Σ^{exp}-protocol presented in [10,8] takes away these assumptions, by adding an auxiliary construction based on a common reference string and some computational overhead. Depending on the proof goal and certain implementation issues, either of these two protocols can be more efficient. We refer to [12] for a detailed security and efficiency analysis.

2.4 Operations on Σ-Protocols

Next, we briefly summarize some techniques, which allow one to use Σ-protocols in a more general way than for proving atomic predicates only.

Reducing the knowledge error. The knowledge error of a Σ-protocol can be reduced from κ to κ^r by repeating the protocol r times in parallel. The verifier accepts the proof, if and only if it accepted all instances [17]. In this way, arbitrarily small knowledge errors can be achieved.

Boolean composition. In practice, it is often necessary to prove knowledge of multiple, or one out of a set of, secret values in one step. This can be achieved by performing so-called And- respectively Or-*compositions*. While the former requires the prover to know the secrets for *all* combined predicates to convince the verifier, he only needs to know *at least one* of them for the latter. In this case, the verifier will not be able to learn which secrets are actually known to the prover [70].

For a Boolean And, the only difference to running the proofs for the combined predicates independently in parallel is, that the verifier only sends *one* challenge c, which is then used in all combined predicates.

Combining n predicates by a Boolean Or is a bit more involved. By allowing the prover to choose the challenges c_i for *all but one* predicate, he can simulate accepting protocol transcripts for those predicates he does not know the secret for. The remaining challenge must then be chosen such that $\sum_{i=1}^{n} c_i \equiv c \mod c_{\mathsf{max}}$. To ensure this, the prover adds $c_1, \ldots, c_n$ to its response, which is now given by $((s_1, c_1), \ldots, (s_n, c_n))$, where s_i is the response of the i-th predicate. In addition to running all verification algorithms, the verifier also checks that the c_i add up to the challenge c.

Threshold composition. For instance, for a contract to become valid it may be required that at least k out of n board members of a company sign the document. Yet, the contracting party should not learn the identity of the signers. Performing such a ZK-PoK by using nested And- and Or-compositions becomes very inefficient if n is large. A much more efficient way for performing such *n-out-of-k threshold compositions* is to apply the technique from [38], instantiated with Shamir's secret sharing scheme [69].

Non-interactivity. Σ-protocols can be made non-interactive by applying the Fiat-Shamir heuristic [44]. The idea is that the prover obtains a random challenge c by hashing its commitment. Additionally to its verification algorithm V, the verifier then also checks whether c was computed correctly. In this way, only a single message has

to be sent in the protocol, and the proof can easily be converted into a signature proof of knowledge.

Algebraic relations among preimages. By re-adjusting the atomic predicates of a proof goal, virtually any algebraic relations among the preimages can be proven. For examples we refer to [70,57,25,26].

3 Compiler

In this section we describe our ZK-PoK compiler that automatically generates provably sound implementations and documentation for specific classes of ZK-PoK protocols from a high-level specification of the intended protocol. The modularly constructed compiler (cf. §3.1) is easy to use and can generate code and documentation for many practical ZK-PoK protocols using arbitrary homomorphisms by applying the built-in techniques presented in §2 and several automatic optimizations (cf. §3.2). Moreover, it is integrated with a tool that formally verifies the soundness of generated protocols for special homomorphisms (cf. §4).

3.1 Architecture

The architecture of our ZK-PoK compiler suite which is built from multiple components is shown in Fig. 2. This allows to easily extend the compiler via new plugins and backends. Furthermore, the single components are designed modularly themselves, such that, e.g., the mathematical libraries used in the C Backend can be exchanged with minor effort.

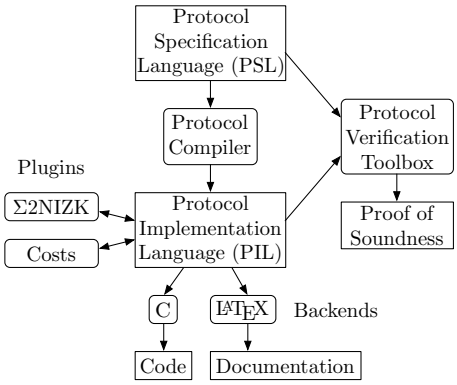

Fig. 2: Architecture of our ZK-PoK compiler suite.

Protocol Specification. The user formulates the specification of the intended protocol in our *Protocol Specification Language* (PSL). This language is based on the Camenisch-Stadler notation [35], and extends it to eliminate any equivocation. As a result it allows to unambiguously specify complex Σ-protocols. On a high level, PSL allows to specify the inputs and algebraic setting of the proof goal, the types of Σ-protocols to be used,

and their compositions. In particular, PSL supports all Σ-protocols presented in §2.3 that can be arbitrarily composed using the composition techniques described in §2.4. We give more details on PSL later in §3.2.

Protocol Compiler. The Protocol Compiler translates the protocol specification into the corresponding protocol implementation formulated in our *Protocol Implementation Language* (PIL). This language can be thought of as a kind of pseudo-code describing the protocol, i.e., the sequence of operations computed by both parties (including group operations, random choices, checks, etc.) and the messages sent between them. Further details on PIL are given in App. A.

Backends. Backends allow to transform the protocol implementation into various output languages. The C Backend generates source code in the C programming language for prover and verifier. By providing the GNU multi-precision arithmetic library [49] this source code can be compiled into executable code. The LaTeX Backend generates a human-readable documentation of the protocol. An example output generated by the LaTeX backend is given in App. B.

Protocol Verification Toolbox. This formal verification component of our compiler takes as input the protocol specification (PSL) and implementation (PIL) files from a compilation run, and extracts from them the relevant information to construct the corresponding proof of soundness. The proof is performed automatically using the theorem prover Isabelle/HOL [61,62] which generates the formal proof and a human-readable documentation on the soundness guarantees of the generated protocol that can be used for product certification purposes. More details are given in §4.

Plugins. The protocol implementation of the generated Σ-protocols can be transformed with plugins. The Σ2NIZK plugin produces a non-interactive ZK-PoK (NIZK) by applying the Fiat-Shamir transformation [44] (cf. §2.4). The functionality of this plugin could easily be extended to signature proofs of knowledge. The Costs plugin determines the abstract costs of the generated protocol, i.e., the communication complexity and the number of operations that need to be performed in each group. This allows to compare the complexity of different protocols on an abstract level and in future releases to automatically select the most efficient one.

3.2 Protocol Specification Language and Optimizations

Next, we describe the optimizations performed by our compiler and the rationale underlying PSL. We show how to use the compiler for the following representative application example: In the context of a group-oriented application we want to prove the following informal statement:

> *"One of two legitimate users has committed to message m without revealing m or the identity of the user who committed."*

For computing a commitment c to message m with randomness r we use the Pedersen commitment scheme [66] of the form $c = g^m h^r$, where g and h are generators of the group $\mathcal{H}$ of known prime order q (e.g., $\mathbb{Z}_p^*$ where p is prime and q divides $p-1$, or an elliptic

curve group) and the committer does not know $\log_g h$. To authenticate legitimate users we use Diffie-Hellman keys: each user randomly picks a sufficiently large secret key sk_i, computes the public key $\mathrm{pk}_i = g^{\mathrm{sk}_i}$ and publishes pk_i. To ease presentation, we use the same group $\mathcal{H}$ for commitments and keys of users, but the compiler could use different groups as well.

Now, given the commitment c and the public keys $\mathrm{pk}_1, \mathrm{pk}_2$ of the two legitimate users, the informal statement translates into this proof goal:

$$\mathsf{ZPK}\Big[(\mu, \rho, \sigma_1, \sigma_2) : c = g^{\mu} h^{\rho} \wedge \big(\mathrm{pk}_1 = g^{\sigma_1} \vee \mathrm{pk}_2 = g^{\sigma_2}\big)\Big],$$

where $(\mu, \rho, \sigma_1) = (m, r, \mathrm{sk}_1)$ or $(\mu, \rho, \sigma_2) = (m, r, \mathrm{sk}_2)$ are possible sets of secrets that allow to prove the relation. With homomorphisms $\psi : (a, b) \mapsto g^a h^b$ and $\phi : (a) \mapsto g^a$ we rewrite this as

$$\mathsf{ZPK}\Big[(\mu, \rho, \sigma_1, \sigma_2) : \underbrace{c = \psi(\mu, \rho)}_{P_0} \wedge \big(\underbrace{\mathrm{pk}_1 = \phi(\sigma_1)}_{P_1} \vee \underbrace{\mathrm{pk}_2 = \phi(\sigma_2)}_{P_2}\big)\Big],$$

where the atomic predicates are P_0, P_1, and P_2. This proof goal together with the underlying algebraic setting can be expressed in PSL as shown in Fig. 3 and described next. Each PSL file consists of the following sections:

```
Declarations { Prime(1024) p;
               Prime(160)  q;
               G=Zmod+(q)  m, r, sk_1, sk_2;
               H=Zmod*(p)  g@{order=q}, h@{order=q}, c@{order=q},
                           pk_1@{order=q}, pk_2@{order=q};  }
Inputs { Public        := p,q,g,h,c,pk_1,pk_2;
         ProverPrivate := m,r,sk_1,sk_2; }
Properties { KnowledgeError := 80;
             ProtocolComposition := P_0 And (P_1 Or P_2);  }
GlobalHomomorphisms { Homomorphism (phi : G -> H : (a) |-> (g^a)); }
// Predicates
SigmaPhi P_0 { Homomorphism (psi : G^2 -> H : (a,b) |-> (g^a * h^b));
               ChallengeLength := 80;  Relation ((c) = psi(m,r));     }
SigmaPhi P_1 { ChallengeLength := 80;  Relation ((pk_1) = phi(sk_1)); }
SigmaPhi P_2 { ChallengeLength := 80;  Relation ((pk_2) = phi(sk_2)); }
```

Fig. 3: PSL Example.

Declarations. All variables used in the protocol must first be declared in this section. PSL supports several data types with a given bit-length such as signed integers (`Int`) or primes (`Prime`). Also intervals (`[a,b]`) and predefined multiplicative and additive groups are supported, e.g., `Zmod*(p)` denotes $(\mathbb{Z}_p^*, *)$ and `Zmod+(q)` denotes $(\mathbb{Z}_q, +)$. In this section, an identifier can be assigned to a group and constants can be predefined. The compiler also supports abstract groups, which can be instantiated with one's favorite group (e.g., such over elliptic curves). The order of elements can be annotated for verification purposes, e.g., as `g@{order=q}`.

Inputs. Here, the inputs of the protocol are assigned to both parties by specifying which ones are publicly known to both and which are private inputs of the prover. All inputs must have been declared beforehand.

Properties. This section specifies the properties of the protocol to be generated. For instance, `KnowledgeError := 80` specifies an intended knowledge error κ of 2^{-80}. The proof goal can be specified by combining the Σ-protocols of the atomic predicates by arbitrarily nested Boolean And and Or operators. Furthermore, the compiler supports n-out-of-k-threshold compositions [38] based on Shamir secret sharing [69] as described in §2.4.

Optimizations. The compiler automatically applies transformations to the proof goal in order to reduce the complexity of the generated protocol. For instance, an expression like `P_1 Or P_2 Or (P_1 And P_2)` is simplified to `P_1 Or P_2` which halves the complexity of the resulting protocol. By introspecting the predicates, further optimizations could be implemented easily.

GlobalHomomorphisms. Homomorphisms that appear in multiple atomic predicates can be defined as global homomorphisms in this optional section. The description of a homomorphism in PSL is a natural translation from the mathematical notation for homomorphisms consisting of name, domain, co-domain, and the mapping function.

Predicates. Finally, the atomic predicates used in the proof goal composition are specified. Each predicate is proved with a Σ-protocol: one of `SigmaPhi`, `SigmaGSP` or `SigmaExp`. For each Σ-protocol, the relation between public images and private preimages must be defined using local or global homomorphisms. `ChallengeLength` specifies the maximum challenge length that can be used to prove this atomic predicate with the given Σ-protocol (cf. §2.3 for details). Note that this value depends, e.g., on the size of the special exponent of the homomorphism, and thus, in general, cannot be automatically determined by the compiler, as the factorization of the special exponent might not be available.

Optimizations. The compiler automatically determines the number of repetitions for each atomic predicate to achieve the intended knowledge error. For proofs in hidden order groups using the Σ^{exp}-protocol the compiler automatically chooses the size of the auxiliary modulus as described in [12] - the automatic choice of the most efficient protocol Σ^{exp} or Σ^{GSP} described therein is currently being implemented. In future work, the automatic choice of parameter sizes could be automatically inferred from a higher-level specification of the intended proof goal.

4 Verification

The Protocol Verification Toolbox (PVT) of our compiler suite (cf. Fig. 2) automatically produces a formal proof for the soundness property of the compiled protocol. In other words it formally validates the guarantee obtained by a verifier executing the compiled protocol: "The prover indeed knows a witness satisfying the proof goal."

Overview. The internal operation of the PVT is sketched in Fig. 4; the phases (1) to (6) are explained in the following. As inputs, two files are given: the protocol specification

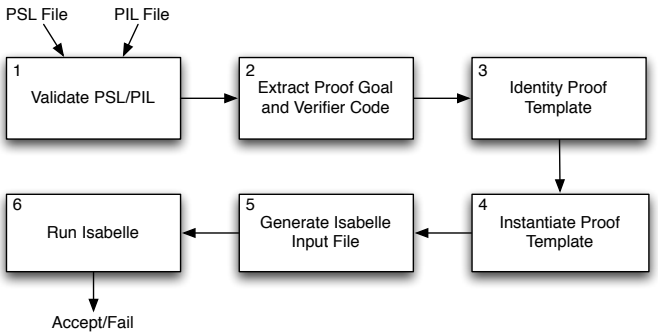

Fig. 4: Internal operation of the Protocol Verification Toolbox (PVT).

(a PSL file) that was fed as input to the compiler, and the protocol implementation description that was produced by the compiler (a PIL file). The PVT first checks (1) the syntactic correctness of the files and their semantic consistency, e.g., it verifies that the PSL and PIL files operate on the same groups, and other similar validations. Then, the information required for the construction of the soundness proof is extracted (2). This information essentially consists of the proof goal description from the PSL file and the code for the verifier in the implementation file. In particular, the former includes the definition of the concrete homomorphisms being used in the protocol, and information about the algebraic properties of elements, homomorphisms, etc.[7].

The reason for the verification toolbox only considering the verifier code is that by definition [17] the soundness of the protocol essentially concerns providing guarantees for the verifier, regardless of whether the prover is honestly executing the protocol or not. Looking at the description of Σ-protocols in §2 and the example PIL file included in App. C, one can see that the verifier code typically is very simple. The exception is the final algebraic verification that is performed on the last response from the prover, which determines whether the proof of knowledge should be accepted. The theoretical soundness proof that we construct essentially establishes that this algebraic check is correct with respect to the proof goal, i.e., that it assures the verifier that the prover must know a valid witness. The soundness proof is then generated in three steps:

a) Firstly, an adequate proof template is selected from those built into the tool (3). If no adequate template exists for this particular protocol, then the user is notified and the process terminates.
b) The proof template is instantiated with the concrete parameters corresponding to the input protocol (4) and translated into an output file (5) compatible with the Isabelle/HOL proof assistant: a theory file.
c) Finally, the proof assistant is executed on the theory file produced before (6). If the proof assistant successfully finishes, then we have a formal proof of the theoretical soundness of the protocol. Furthermore, the Isabelle/HOL framework permits generating a human-readable version of the proof that can be used for product documentation.

[7] This justifies the verification-specific annotations in the PSL file, as described in §3.

The process is fully automatic and achieving this was a major challenge to our design. As can be seen in Fig. 4, our tool uses Isabelle/HOL [62] as a back-end (6). In order to achieve automatic validation of the generated proofs, it was necessary to construct a library of general lemmata and theorems in HOL that capture, not only the properties of the algebraic constructions that are used in ZK-PoK protocols, but also the generic provable security stepping stones required to establish the theoretical soundness property. We therefore intensively employed and extended the Isabelle/HOL Algebra Library [7], which contains a wide range of formalizations of mathematical constructs. By relying on a set of existing libraries such as this, development time was greatly shortened, and we were able to create a proof environment in which we can express proof goals in a notation that is very close to the standard mathematical notation adopted in cryptography papers. More information about Isabelle/HOL can be found in [61,62].

Remark. No verification is carried out at this moment of the executable code that is generated from the PIL file, but this is a program correctness problem rather than a theoretical security problem. It thus must be addressed using a different techniques that we do not cover in this paper.

We next detail the most important aspects of our approach.

Proof strategy. Proving the soundness property of the ZK-PoK protocols produced by the compiler essentially means proving that the success probability of a malicious prover in cheating the verifier is bounded by the intended knowledge error. As described in §2.2, this involves proving the existence of (or simply to construct) an efficient knowledge extractor.

Our verification component is currently capable of dealing with the Σ^{ϕ}-protocol, which means handling proof goals involving special homomorphisms (cf. Definition 1) for which it is possible to efficiently find pseudo-preimages. As all special homomorphisms used in cryptography fall into one of the two classes described when introducing special homomorphisms in §2, the verification toolbox has the ability to automatically find a pseudo-preimage for any concrete homomorphism that it encounters without human interaction.

A central stepping stone in formally proving the existence of an efficient knowledge extractor is the following lemma (which actually proves Theorem 2) that we have formalized in HOL.

Lemma 3 (Shamir's Trick [8]). *Let (u_1, v_1) and (u_2, v_2) be pseudo-preimages of y under homomorphism ϕ. If v_1 and v_2 are co-prime, then there exists a polynomial time algorithm that computes a preimage x of y under ϕ. This algorithm consists of the Extended Euclidean Algorithm to obtain $a, b \in \mathbb{Z}$ such that $av_1 + bv_2 = 1$, and then calculating $x = au_1 + bu_2$.*

In fact, given a special homomorphism and two accepting protocol transcripts for a ZK-PoK of an atomic predicate, we prove the existence of a knowledge extractor by ensuring that we may instantiate Lemma 3.

The compiler also supports composition with Boolean And and Or. If multiple predicates are combined by And, the verification tool defines as proof goal the existence of a knowledge extractor for each and all of them separately, i.e., one needs to show that the

witness for each predicate can be extracted independently from the other predicates. In case of Or proofs (i.e., knowledge of one out of a set of preimages), the proof strategy looks as follows. First, for each atomic predicate, an Isabelle theorem proves the existence of a knowledge extractor. In a second step, it is then shown that the assumptions of at least one of these theorems are satisfied (i.e., that at least for one predicate we actually have different challenges).

Isabelle/HOL formalization. The HOL theory file produced by the Protocol Verification Toolbox is typical, in the sense that it contains a set of auxiliary lemmata that are subsequently used as simplification rules, and a final lemma with the goal to be proved. The purpose of the auxiliary lemmata is to decompose the final goal into simpler and easy to prove subgoals. They allow a systematic proof strategy that, because it is modularized, can handle proof goals of arbitrary complexity. Concretely, the proof goal for a simple preimage ZK-PoK such as those associated with Diffie-Hellman keys ($\mathrm{pk} = g^{\mathrm{sk}}$) used in the example in §3 looks like the following theorem formulation:

Theorem (Proof Goal). *Let G and H be commutative groups, where G represents the group of integers. Take as hypothesis the algebraic definition of the exponentiation homomorphism $\phi : G \to H$, quantified for all values of G, i.e., fix $g \in H$ with order q and assume $\forall a \in G.\ \phi(a) = g^a$.*

Take a prime $q > 2$ and $c_{\mathsf{max}} \in \mathbb{Z}$ such that $0 < c_{\mathsf{max}} < q$, take $t, \mathrm{pk} \in H$ such that the order of pk *is q, take $s', s'' \in G$ and $c', c'' \in \mathbb{Z}$ such that $0 < c', c'' < c_{\mathsf{max}}$ and $c' \neq c''$, and assume $\phi(s') = t \cdot \mathrm{pk}^{c'} \ \wedge\ \phi(s'') = t \cdot \mathrm{pk}^{c''}$.*

Then there exist $a, b \in \mathbb{Z}$ such that $\phi(au + b\Delta s) = \mathrm{pk} \quad \wedge \quad av + b\Delta c = 1$, where $\Delta s := s' - s''$ and $\Delta c := c' - c''$, and $(u, v) = (0, q) \in G \times \mathbb{Z}$ is a pseudo-preimage of pk *under ϕ.*

Instrumental in constructing the proof goal and auxiliary lemmata that permit carrying out the formal proof are the verifier's verification equations extracted from the PIL file. Indeed, the part of the proof goal that describes the two transcripts of the protocol (t, c', s') and (t, c'', s'') is constructed by translating this verification equation into Isabelle/HOL. For example, the following statement from the PIL file:

```
Verify((_t*(pk^_c)) == (g^_s));
```

will be translated into the Isabelle/HOL formalization

$$t \otimes_H (\mathrm{pk}(\wedge_H)c') = g(\wedge_H)s';\ t \otimes_H (\mathrm{pk}(\wedge_H)c'') = g(\wedge_H)s'';\ c' \neq c'';$$

where $\otimes_H$ and $(\wedge_H)$ represent the multiplicative and exponentiation operations in H, respectively. A typical proof is then structured as follows.

A first lemma with these equations as hypothesis allows the system to make a simple algebraic manipulation, (formally) proving the following:

$$(t \otimes_H (\mathrm{pk}(\wedge_H)c')) \otimes_H \mathrm{inv}_H(t \otimes_H (\mathrm{pk}(\wedge_H)c'')) = g(\wedge_H)s' \otimes_H \mathrm{inv}_H(g(\wedge_H)s'')$$

where inv_H represents the inversion operation for H. The subsequent lemmata continue simplifying this equation, until we obtain:

$$\text{pk}(\wedge_H)(c' - c'') = g(\wedge_H)(s' - s'').$$

By introducing the homomorphism $\phi : G \to H$ we are able to show

$$\text{pk}(\wedge_H)(\Delta c) = \phi(\Delta s)$$

where $\Delta c = c' - c''$ and $\Delta s = s' - s''$. We thus obtained the pseudo-preimage $(\Delta s, \Delta c)$ from the two accepting protocol transcripts. The second pseudo-preimage, which is needed for Lemma 3, is found by analyzing the proof goal extracted from the PSL file, which in our example was:

```
Relation((pk) = phi(sk)).
```

Recall that we have embedded in our tool the domain specific knowledge to generate pseudo-preimages for the class of protocols that we formally verify, so that we introduce another explicit pseudo-preimage as an hypothesis in our proof, e.g. $(0, q)$, and prove that it satisfies the pseudo-preimage definition. At this point we can instantiate the formalization of Lemma 3, and complete the proof for the above theorem, which implies the existence of a knowledge extractor.

Proof goals for more complex Σ-protocols involving And and Or composition of simple preimage ZK-PoK are formalized as described in the previous subsection and in line with the theoretic background introduced in §2. For And combinations, the proof goal simply contains the conjunction of the independent proof goals for each of the simple preimage proofs provided as atomic predicates. For Or combinations, the proof goal assumes the existence of two transcripts for the composed protocol

$$((t^1, \ldots, t^n), c', ((s_1^1, c_1^1), \ldots, (s_1^n, c_1^n)) \quad \text{with} \quad \sum_{i=1}^{n} c_1^i \equiv c' \mod c_{\mathsf{max}}$$

and analogously for c'', such that $c' \neq c''$. It then states that, there exists an $i \in \{1, \ldots, n\}$ for which we can construct a proof of existence of a knowledge extractor such as that described above. The assumptions regarding the consistency of the previous summations are, again, a direct consequence of the verifier code as can be seen in the example in App. C.

References

1. A. Adelsbach and A.-R. Sadeghi. Zero-knowledge watermark detection and proof of ownership. In *Information Hiding*, volume 2137 of *LNCS*, pages 273–288. Springer, 2001.
2. J. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on Σ-protocols. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *European Symposium on Research in Computer Security – ESORICS 2010*, volume 6345 of *LNCS*, pages 151–167. Springer, 2010.

3. G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *CRYPTO 00*, volume 1880 of *LNCS*, pages 255–270. Springer, 2000.
4. M. Backes, C. Hritcu, and M. Maffei. Type-checking zero-knowledge. In *ACM CCS 08*, pages 357–370. ACM, 2008.
5. M. Backes, M. Maffei, and D. Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *IEEE Symposium on Security and Privacy – SP 08*, pages 202–215. IEEE, May 2008.
6. M. Backes and D. Unruh. Computational soundness of symbolic zero-knowledge proofs against active attackers. In *IEEE Computer Security Foundations Symposium - CSF 08*, pages 255–269, June 2008. Preprint on IACR ePrint 2008/152.
7. C. Ballarin, F. Kammüller, and L. Paulson. The Isabelle/HOL Algebra Library. `http://isabelle.in.tum.de/library/HOL/HOL-Algebra/document.pdf`, 2008.
8. E. Bangerter. *Efficient Zero-Knowledge Proofs of Knowledge for Homomorphisms*. PhD thesis, Ruhr-University Bochum, 2005.
9. E. Bangerter, T. Briner, W. Heneka, S. Krenn, A.-R. Sadeghi, and T. Schneider. Automatic generation of Σ-protocols. In *EuroPKI 09 (to appear)*, 2009.
10. E. Bangerter, J. Camenisch, S. Krenn, A.-R. Sadeghi, and T. Schneider. Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471, 2008. Poster Session of EUROCRYPT 09.
11. E. Bangerter, J. Camenisch, and U. Maurer. Efficient proofs of knowledge of discrete logarithms and representations in groups with hidden order. In *PKC 05*, volume 3386 of *LNCS*, pages 154–171. Springer, 2005.
12. E. Bangerter, A. Grünert, and S. Krenn. On the (in)practicability of zero-knowledge proofs of knowledge using hidden-order groups. Technical report, Bern University of Applied Sciences (CH), University of Fribourg (CH), and University of London (GB), 2010.
13. E. Bangerter, S. Krenn, A.-R. Sadeghi, T. Schneider, and J.-K. Tsay. On the design and implementation of efficient zero-knowledge proofs of knowledge. In *Software Performance Enhancements for Encryption and Decryption and Cryptographic Compilers – SPEED-CC 09*, October 12-13, 2009.
14. G. Barthe, B. Grégoire, and S. Béguelin. Formal certification of code-based cryptographic proofs. In *ACM SIGPLAN-SIGACT POPL 09*, pages 90–101, 2009.
15. G. Barthe, D. Hedin, S. Zanella Béguelin, B. Grégoire, and S. Heraud. A machine-checked formalization of Σ-protocols. In *23rd IEEE Computer Security Foundations Symposium – CSF 2010*. IEEE, 2010.
16. Anguraj Baskar, R. Ramanujam, and S. P. Suresh. A Dolev-Yao Model for Zero Knowledge. In *Advances in Computer Science (ASIAN 09). Information Security and Privacy*, volume 5913 of *LNCS*, pages 137 – 146. Springer, 2009.
17. M. Bellare and O. Goldreich. On defining proofs of knowledge. In *CRYPTO 92*, volume 740 of *LNCS*, pages 390–420. Springer, 1993.
18. A. Bhargav-Spantzel, A. C. Squicciarini, S. Modi, M. Young, E. Bertino, and S. J. Elliott. Privacy preserving multi-factor authentication with biometrics. *Journal of Computer Security*, 15(5):529–560, 2007.
19. K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Cryptographically verified implementations for TLS. In *ACM CCS 08*, pages 459–468. ACM, 2008.
20. K. Bhargavan, C. Fournet, A.D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM Trans. Program. Lang. Syst.*, 31(1):1–61, 2008.
21. B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Workshop on Computer Security Foundations – CSFW 01*, page 82. IEEE, 2001.
22. B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy – SP 06*, pages 140–154. IEEE, 2006.
23. B. Blanchet. ProVerif: Cryptographic protocol verifier in the formal model, 2010.
24. S. Brands. Untraceable off-line cash in wallet with observers. In *CRYPTO 93*, volume 773 of *LNCS*, pages 302–318. Springer, 1994.
25. S. Brands. Rapid demonstration of linear relations connected by boolean operators. In *EUROCRYPT 97*, volume 1233 of *LNCS*, pages 318–333. Springer, 1997.

26. E. Bresson and J. Stern. Proofs of knowledge for non-monotone discrete-log formulae and applications. In *ISC 02*, volume 2433 of *LNCS*, pages 272–288. Springer, 2002.
27. E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *ACM CCS 04*, pages 132–145. ACM Press, 2004.
28. T. Briner. Compiler for zero-knowledge proof-of-knowledge protocols. Master's thesis, ETH Zurich, 2004.
29. J. Camenisch. *Group Signature Schemes and Payment Systems Based on the Discrete Logarithm Problem*. PhD thesis, ETH Zurich, Konstanz, 1998.
30. J. Camenisch and E. V. Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM CCS 02*, pages 21–30. ACM Press, 2002.
31. J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Balancing accountability and privacy using e-cash (extended abstract). In *SCN 06*, volume 4116 of *LNCS*, pages 141–155. Springer, 2006.
32. J. Camenisch and A. Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *EUROCRYPT 01*, volume 2045 of *LNCS*, pages 93–118. Springer, 2001.
33. J. Camenisch and M. Michels. Proving in zero-knowledge that a number is the product of two safe primes. In *EUROCRYPT 99*, volume 1592 of *LNCS*, pages 107–122. Springer, 1999.
34. J. Camenisch, M. Rohe, and A.-R. Sadeghi. Sokrates - a compiler framework for zero-knowledge protocols. In *WEWoRC 05*, 2005.
35. J. Camenisch and M. Stadler. Efficient group signature schemes for large groups (extended abstract). In *CRYPTO 97*, volume 1294 of *LNCS*, pages 410–424. Springer, 1997.
36. Z. Cao. Analysis of one popular group signature scheme. In *ASIACRYPT 06*, volume 4284 of *LNCS*, pages 460–466. Springer, 2006.
37. R. Cramer. *Modular Design of Secure yet Practical Cryptographic Protocols*. PhD thesis, CWI and University of Amsterdam, 1997.
38. R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO 94*, volume 839 of *LNCS*, pages 174–187. Springer, 1994.
39. R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO 98*, volume 1462 of *LNCS*, pages 13–25. Springer, 1998.
40. I. Damgård. On Σ-protocols, 2004. Lecture on Cryptologic Protocol Theory; Faculty of Science, University of Aarhus.
41. I. Damgård and E. Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *ASIACRYPT 02*, volume 2501 of *LNCS*, pages 77–85. Springer, 2002.
42. I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *PKC 09*, volume 5443 of *LNCS*, pages 160–179. Springer, 2009.
43. I. Damgård and M. Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *PKC 01*, volume 1992 of *LNCS*, pages 119–136. Springer, 2001.
44. A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In *CRYPTO 86*, volume 263 of *LNCS*, pages 186–194. Springer, 1987.
45. E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *CRYPTO 97*, volume 1294 of *LNCS*, pages 16–30. Springer, 1997.
46. T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO 84*, volume 196 of *LNCS*, pages 10–18. Springer, 1985.
47. O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(1):691–729, 1991. Preliminary version in 27th FOCS, 1986.
48. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI 05*, volume 3385 of *LNCS*, pages 363–379. Springer, 2005.
49. T. Granlund. The GNU MP Bignum Library. `http://gmplib.org/`, 2010.
50. J. Groth. Non-interactive zero-knowledge arguments for voting. In *ACNS 05*, volume 3531 of *LNCS*, pages 467–482. Springer, 2005.
51. L. Guillou and J.-J. Quisquater. A "paradoxical" identity-based signature scheme resulting from zero-knowledge. In *CRYPTO 88*, volume 403 of *LNCS*, pages 216–231. Springer, 1990.
52. W. Han, K. Chen, and D. Zheng. Receipt-freeness for Groth e-voting schemes. *Journal of Information Science and Engineering*, 25(2):517–530, 2009.

53. G. Hanaoka and K. Kurosawa. Efficient chosen ciphertext secure public key encryption under the computational Diffie-Hellman assumption. In *ASIACRYPT 08*, volume 5350 of *LNCS*, pages 308–325. Springer, 2008.
54. H. Kikuchi, K. Nagai, W. Ogata, and M. Nishigaki. Privacy-preserving similarity evaluation and application to remote biometrics authentication. *Soft Computing*, 14(5):529–536, 2010.
55. S. Kunz-Jacques, G. Martinet, G. Poupard, and J. Stern. Cryptanalysis of an efficient proof of knowledge of discrete logarithm. In *PKC 06*, volume 3958 of *LNCS*, pages 27–43. Springer, 2006.
56. Y. Lindell, B. Pinkas, and N. P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *SCN 08*, volume 5229 of *LNCS*, pages 2–20. Springer, 2008.
57. H. Lipmaa. On diophantine complexity and statistical zeroknowledge arguments. In *ASIACRYPT 03*, volume 2894 of *LNCS*, pages 398–415. Springer, 2003.
58. P. MacKenzie, A. Oprea, and M. K. Reiter. Automatic generation of two-party computations. In *ACM CCS 03*, pages 210–219. ACM, 2003.
59. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX Security 04*, 2004.
60. S. Meiklejohn, C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash. In *USENIX 10 (to appear)*, 2010.
61. T. Nipkow and L. Paulson. Isabelle web site. `http://isabelle.in.tun.de`, 2010.
62. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002.
63. T. Okamoto and S. Uchiyama. A new public-key cryptosystem as secure as factoring. In *EUROCRYPT 98*, volume 1403 of *LNCS*, pages 308–318. Springer, 1998.
64. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT 99*, volume 1592 of *LNCS*, pages 223–238. Springer, 1999.
65. L. Paulson. *Isabelle: a Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.
66. T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO 91*, volume 576 of *LNCS*, pages 129–140. Springer, 1992.
67. R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
68. C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.
69. A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
70. N. P. Smart, editor. *Final Report on Unified Theoretical Framework of Efficient Zero-Knowledge Proofs of Knowledge*. CACE project deliverable, 2009.

A Protocol Implementation Language

From the PSL file, the compiler automatically generates a description of the protocol in the *Protocol Implementation Language* (PIL). This language describes the algorithms for prover and verifier in detail, including the sequence of operations that are performed, and the messages exchanged between the two parties (for an example see the PIL output generated from the PSL input of Fig. 3 in App. C). This PIL file is used for the automatic generation of the LaTeX documentation (cf. App. B), and source code in the C programming language using the respective backends. The PIL description is also fed as an input to the protocol verification toolbox described in §4.

B Generated LaTeX Output for Example of §3.2

1 Declarations and Inputs

1.1 Common Declarations

$p \in \mathbb{P}_{1024}$
$q \in \mathbb{P}_{160}$
$k_{sec} = 80 \in \mathbb{Z}$
$h, g, c, pk_2, pk_1 \in \mathbb{Z}_p^*$

1.2 Private Declarations – Prover

1.2.1 Prover's Inputs

$m, sk_1, r, sk_2 \in \mathbb{Z}_q$

1.2.2 Global Variables

$\mathbf{c_2}, \mathbf{c_1} \in \{0,1\}^{80}$
$\mathbf{r_2}, \mathbf{r_3}, \mathbf{r_1}, \mathbf{r_4}, \mathbf{s_3}, \mathbf{s_2}, \mathbf{s_1}, \mathbf{s_4} \in \mathbb{Z}_q$

1.3 Private Declarations – Verifier

1.3.1 Global Variables

$\mathbf{c} \in \{0,1\}^{80}$
$\mathbf{t_1}, \mathbf{t_2}, \mathbf{t_3} \in \mathbb{Z}_p^*$

2 Protocol Rounds

2.1 Prover – Round0

$h^q \stackrel{?}{=} 1$
$g^q \stackrel{?}{=} 1$
$c^q \stackrel{?}{=} 1$
${pk_2}^q \stackrel{?}{=} 1$
${pk_1}^q \stackrel{?}{=} 1$

2.2 Verifier – Round0

$h^q \stackrel{?}{=} 1$
$g^q \stackrel{?}{=} 1$
$c^q \stackrel{?}{=} 1$
${pk_2}^q \stackrel{?}{=} 1$
${pk_1}^q \stackrel{?}{=} 1$

2.3 Prover – Round1

$\mathbf{r_1} \in_R \mathbb{Z}_q$
$\mathbf{r_2} \in_R \mathbb{Z}_q$
$\mathbf{t_1} := g^{\mathbf{r_1}} \cdot h^{\mathbf{r_2}}$

If known: sk_1 **do:**
| $\mathbf{r_3} \in_R \mathbb{Z}_q$
| $\mathbf{t_2} := g^{\mathbf{r_3}}$
Else
| $\mathbf{c_1} \in_R \{0,1\}^{80}$
| $\mathbf{s_3} \in_R \mathbb{Z}_q$
| $\mathbf{t_2} := g^{\mathbf{s_3}} \cdot pk_1{}^{((-(\mathbf{c_1})))}$
End if

If known: sk_2 **do:**
| $\mathbf{r_4} \in_R \mathbb{Z}_q$
| $\mathbf{t_3} := g^{\mathbf{r_4}}$
Else
| $\mathbf{c_2} \in_R \{0,1\}^{80}$
| $\mathbf{s_4} \in_R \mathbb{Z}_q$
| $\mathbf{t_3} := g^{\mathbf{s_4}} \cdot pk_2{}^{((-(\mathbf{c_2})))}$
End if

2.4 Verifier – Round1

$\mathbf{c} \in_R \{0,1\}^{80}$

$\mathbf{c}$;

2.5 Prover – Round2

$\mathbf{s_1} := \mathbf{r_1} + m \cdot \mathbf{c}$
$\mathbf{s_2} := \mathbf{r_2} + r \cdot \mathbf{c}$
Complete $(\mathbf{c_1}, \mathbf{c_2})$ such that
$\mathbf{c_1} + \mathbf{c_2} \equiv \mathbf{c} \bmod 2^{80}$

If known: sk_1 **do:**
| $\mathbf{s_3} := \mathbf{r_3} + sk_1 \cdot \mathbf{c_1}$
End if

If known: sk_2 **do:**
| $\mathbf{s_4} := \mathbf{r_4} + sk_2 \cdot \mathbf{c_2}$
End if

$\mathbf{s_1}$; $\mathbf{s_2}$; $\mathbf{s_3}$; $\mathbf{s_4}$; $\mathbf{c_1}$;

2.6 Verifier – Round2

Local Round Variables:
$\mathbf{c_2} \in \{0,1\}^{80}$

$\mathbf{s_1} \stackrel{?}{\in} \mathbb{Z}_q$
$\mathbf{s_2} \stackrel{?}{\in} \mathbb{Z}_q$
$\mathbf{t_1} \cdot c^{\mathbf{c}} \stackrel{?}{=} g^{\mathbf{s_1}} \cdot h^{\mathbf{s_2}}$

Complete $(\mathbf{c_1}, \mathbf{c_2})$ such that
$\mathbf{c_1} + \mathbf{c_2} \equiv \mathbf{c} \bmod 2^{80}$
$\mathbf{s_3} \stackrel{?}{\in} \mathbb{Z}_q$
$\mathbf{t_2} \cdot pk_1^{\mathbf{c_1}} \stackrel{?}{=} g^{\mathbf{s_3}}$

$\mathbf{s_4} \stackrel{?}{\in} \mathbb{Z}_q$
$\mathbf{t_3} \cdot pk_2^{\mathbf{c_2}} \stackrel{?}{=} g^{\mathbf{s_4}}$

C Generated PIL Code for Example of §3.2

```
ExecutionOrder := (Prover.Round0, Verifier.Round0, Prover.Round1, Verifier.Round1,
    Prover.Round2, Verifier.Round2);
Common (
    Prime(1024) p;
    Prime(160) q;
    H=Zmod*(p) pk_1, g, c, h, pk_2
) {}

Prover(G=Zmod+(q) sk_2, sk_1, m, r) {
    _C=Int(80) _c_2, _c_1;
    G _r_2, _r_3, _r_1, _r_4, _s_3, _s_2, _s_1, _s_4;
    Def (Void): Round0(Void) {
        Verify((pk_1^q) == 1);
        Verify((g^q) == 1);
        Verify((c^q) == 1);
        Verify((h^q) == 1);
        Verify((pk_2^q) == 1);
    }

    Def (H _t_1, _t_2, _t_3): Round1(Void) {
        _r_1 := Random(G);
        _r_2 := Random(G);
        _t_1 := ((g^_r_1)*(h^_r_2));
        IfKnown(sk_1){
            _r_3 := Random(G);
            _t_2 := (g^_r_3);
        } Else {
            _c_1 := Random(_C);
            _s_3 := Random(G);
            _t_2 := ((g^_s_3)*(pk_1^(-(_c_1))));
        }
        IfKnown(sk_2){
            _r_4 := Random(G);
            _t_3 := (g^_r_4);
        } Else {
            _c_2 := Random(_C);
            _s_4 := Random(G);
            _t_3 := ((g^_s_4)*(pk_2^(-(_c_2))));
        }
    }
```

```
    Def (_s_1; _s_2; _s_3; _s_4; _c_1): Round2(_C _c) {
        _s_1 := (_r_1+(m*_c));
        _s_2 := (_r_2+(r*_c));
        Complete((_c_1,_c_2),_c,OR);
        IfKnown(sk_1){
            _s_3 := (_r_3+(sk_1*_c_1));
        }
        IfKnown(sk_2){
            _s_4 := (_r_4+(sk_2*_c_2));
        }
    }
}

Verifier() {
    _C=Int(80) _c;
    H _t_1, _t_2, _t_3;
    Def (Void): Round0(Void) {
        Verify((pk_1^q) == 1);
        Verify((g^q) == 1);
        Verify((c^q) == 1);
        Verify((h^q) == 1);
        Verify((pk_2^q) == 1);
    }

    Def (_c): Round1(_t_1; _t_2; _t_3) {
        _c := Random(_C);
    }

    Def (Void): Round2(G=Zmod+(q) _s_1, _s_2, _s_3, _s_4; _C _c_1) {
        _C _c_2;
        CheckMembership(_s_1, G);
        CheckMembership(_s_2, G);
        Verify((_t_1*(c^_c)) == ((g^_s_1)*(h^_s_2)));
        Complete((_c_1,_c_2),_c,OR);
        CheckMembership(_s_3, G);
        Verify((_t_2*(pk_1^_c_1)) == (g^_s_3));
        CheckMembership(_s_4, G);
        Verify((_t_3*(pk_2^_c_2)) == (g^_s_4));
    }
}
```