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

José Bacelar Almeida[1], Endre Bangerter[2], Manuel Barbosa[1], Stephan Krenn[3], Ahmad-Reza Sadeghi[4], and 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 a high potential impact, their real world deployment is typically hindered by their significant complexity compared to other (non-interactive) crypto primitives. Moreover, their design and implementation are time-consuming and error-prone.

We contribute to overcoming these challenges as follows: We present a comprehensive specification language and a compiler for ZK-PoK protocols based on Σ-protocols. 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 DAA. Finally, our compiler is certifying, in the sense that it automatically produces a formal proof of the soundness of the compiled protocol for a large class of protocols using the Isabelle/HOL theorem prover.

Keywords: 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

* This work was in part funded by the European Community's Seventh Framework Programme (FP7) under grant agreement no. 216499. The compiler can be found at http://zkc.cace-project.eu, and a full version of this paper is given in [1].

soundness), without the verifier being able to learn anything about the secret (*zero-knowledge*). For a formal definition we refer to [2].

Almost all practically efficient ZK-PoK are based on so-called Σ-protocols, and allow one to prove 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 [3], biometric authentication [4], group signatures [5], interactive verifiable computation [6], e-cash [7], and secure multiparty computation [8].

While many of these applications only exist at the specification level, directed applied research efforts have already brought innovative systems using ZK-PoKs to the real world. The probably most prominent example is *Direct Anonymous Attestation* (DAA) [9], a privacy enhancing mechanism for remote authentication of computing platforms. Another example is the *idemix* anonymous credential system [10] for user-centric identity management.

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

In particular, the soundness property needs to be proved for each ZK-PoK protocol from scratch. The proofs often are not inherently complex, but still require an intricate knowledge of the techniques being used. This is a major hurdle for 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 can lead to serious security flaws, see for example [11] reporting a flaw in [12].

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

Our Contributions. In a nutshell, we present a toolbox that takes an abstract description of the proof goal[1] of a ZK-PoK as input, and produces a C implementation of a provably sound protocol. More precisely, we extend previous directions with multiple functionalities of practical and theoretical relevance:

– We present a compiler for ZK-PoK based on Σ-protocols. The compiler (and its input language) support all relevant basic Σ-protocols and composition techniques found in the literature. The user can specify preimage proofs for arbitrary group homomorphisms, and combine them by logical `AND` and `OR` operators. Also, algebraic relations among the secrets can be proved.

[1] 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.

Examples of protocols that can be automatically generated by our compiler include [3,4,5,6,7,8,9,10,13,14,15,16].

- The compiler absorbs certain design-level decisions, for example by automatically choosing security parameters and intervals used in the protocols to assert the statistical ZK property for 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 its soundness[2]. 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 [17]. Our tool can therefore be seen as a *certifying* compiler. The formal verification component currently supports a subset of the protocols for which our compiler can generate code, but it already covers a considerable class of applications, including [7,13,14].

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 [18,19,20].

A first prototype ZK-PoK compiler was developed in [21,22] and extended within the CACE project [23,24]. 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 [25].

Very recently, Meiklejohn et al. [26] 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 [27]. Further, formal verification is left as an "interesting area of study" in [26].

Symbolic models that are suitable for expressing and formally analyzing protocols that rely on ZK protocols as building blocks were presented in [28,29]. In [28] the first mechanized analysis framework for such protocols was proposed by extending the automatic tool ProVerif [30]. The work in [31] 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

[2] 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.

security model for this primitive. Recent results in establishing the computational soundness of ZK-PoK-aware symbolic analysis can be found in [32]. 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. [33] describing the formalization of the theory of ZK-PoK in the Coq-based CertiCrypt tool [34]. 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 [35]. In [36,37], 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 [38] (resp. CryptoVerif [39]), 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 then present in §3. Finally, the formal verification infrastructure is explained in §4.

2 Preliminaries

Before recapitulating the basics of Σ-protocols we introduce some notation. By $s \in_R \mathcal{S}$ we denote the uniform random choice of s from set $\mathcal{S}$. The order of a group $\mathcal{G}$ is denoted by $\mathrm{ord}(\mathcal{G})$, and the smallest prime dividing $a \in \mathbb{Z}$ by $\mathrm{minDiv}(a)$. We use the notation from [40] for specifying ZK-PoK. A term like

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

means "*zero-knowledge proof of knowledge of values* χ_1, χ_2 *such that* $y_1 = \phi_1(\chi_1)$, $y_2 = \phi_2(\chi_2)$, *and* $\chi_1 = a\chi_2$" with homomorphisms ϕ_1, ϕ_2. Variables of which knowledge is proved are denoted by Greek letters, whereas publicly known quantities are denoted by Latin letters. Note that this notation only specifies a *proof goal*: it describes what has to be proved, but there may be various, differently efficient protocols to do so. We call a term like $y = \phi(\chi)$ in the proof goal an *atom*. A *predicate* is the composition of atoms and predicates using arbitrary many (potentially none) boolean operators And ($\wedge$) and Or ($\vee$).

2.1 Σ-Protocols as ZK-PoK Protocols

Most practical ZK-PoK are based on *Σ-protocols*. Given efficient algorithms $\mathsf{P}_1, \mathsf{P}_2, \mathsf{V}$, these have the following form: to prove knowledge of a secret χ satisfying a relation with some public y, the prover first sends a *commitment* $t := \mathsf{P}_1(\chi, y)$ to the verifier, who then draws a random *challenge* c from a predefined *challenge set* $\mathcal{C}$. Receiving c, the prover computes a *response* $s := \mathsf{P}_2(\chi, y, c)$. 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 transcript*.

Formally, for the protocol to be a proof of knowledge with knowledge error κ, the verifier must always accept for an honest prover. Furthermore, 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 χ from the prover in a number of steps proportional to $(\varepsilon - \kappa)^{-1}$ [2]. 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 χ' satisfying the relation [41,42].

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 [41,42].

2.2 Proving Atoms

We next summarize the basic techniques for proving atoms.

The Σ^ϕ-Protocol. The Σ^ϕ-protocol allows to efficiently prove knowledge of preimages under homomorphisms with a finite domain [13,43]. For instance, it can be used to prove knowledge of the content of ciphertexts under the Cramer/Shoup [44] or the RSA [27] encryption schemes. 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. Depending on the homomorphism ϕ, the knowledge error κ of the Σ^ϕ-protocol varies significantly. We thus recall a definition by Cramer [41].

A homomorphisms $\phi : \mathcal{G} \rightarrow \mathcal{H}$ is called *special*, if there is a $v \in \mathbb{Z} \setminus \{0\}$, such that for every $y \in \mathcal{H}$ a preimage u of y^v can efficiently be computed. The pair (u, v) is then called a *pseudo-preimage* of y. For instance, all homomorphisms with known-order codomain are special: if $\mathrm{ord}(\mathcal{H}) = q$, a pseudo-preimage of $y \in \mathcal{H}$ is given by $(0, q)$, as we always have $\phi(0) = 1 = y^q$.

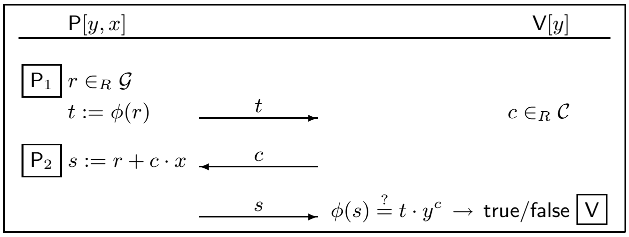
$\mathsf{P}[y, x]$
$\mathsf{V}[y]$
$\boxed{\mathsf{P}_1}$ $r \in_R \mathcal{G}$
$t := \phi(r)$
t
$c \in_R \mathcal{C}$
$\boxed{\mathsf{P}_2}$ $s := r + c \cdot x$
c
s
$\phi(s) \overset{?}{=} t \cdot y^c \rightarrow$ true/false $\boxed{\mathsf{V}}$

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

We now state the knowledge error that can be achieved by the Σ^{ϕ}-protocol:

Theorem 1 ([41]). *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 $\kappa = 1/c_{\mathsf{max}}$, 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 (including, for example, $\phi : \mathbb{Z} \rightarrow \mathbb{Z}_n^* : a \mapsto g^a$, 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 [45,46]. On the other hand, the so-called Σ^{exp}-protocol presented in [23,47] takes away these assumptions, by adding an auxiliary construction based on a common reference string and some computational overhead.

2.3 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 atoms 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. In this way, arbitrarily small knowledge errors can be achieved [2].

Composition techniques. 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- or Or-*compositions*, respectively. 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 [48].

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 slightly more involved. The prover is allowed to simulate all-but-one accepting protocol transcripts (for the predicates it does not know the secrets for) by allowing it to choose the according

c_i. The remaining challenge is then fixed such that $\sum_{i=1}^{n} c_i \equiv c \mod c_{\mathsf{max}}$. To ensure this, the response 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.

In principle, any Boolean formula can be proved by combining these two techniques. Yet, when proving knowledge of k out of n secrets this 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 [49], instantiated with Shamir's secret sharing scheme [50].

Non-interactivity. By computing the challenge as hash of the commitment t with a random oracle, a Σ-protocol can be made non-interactive [51]. Besides reducing the round- and communication complexity of the proofs, this technique allows conversion into signature proofs of knowledge as well.

Algebraic relations among preimages. By re-adjusting the atoms of a proof goal, virtually any algebraic relations among the preimages can be proven. For examples we refer to [16,45,46,48,52,53].

3 Compiler

In this section we present a compiler, which is easy to use and that can generate code and documentation for the ZK-PoK protocols used in most systems found in the literature. Its modular design allows one to easily extend its functionality with minor effort only. Finally, an integrated tool is capable of formally verifying the soundness of the generated protocols for special homomorphisms.

3.1 Architecture

Our compiler is built from multiple components as shown in Fig. 2. These components are again designed modularly themselves. Thus, enhancing the compiler on a high level (e.g., adding backends) and on a low level (e.g., changing libraries used in the backends) can be done without affecting unrelated components.

Protocol Specification. We call the input language of our compiler *Protocol Specification Language* (or *PSL*). It is based on the standard notation for ZK-PoK introduced in [40], but takes away any ambiguity by also containing group specifications, etc. On a high level, PSL allows one to specify the protocol inputs, the algebraic setting, the types of Σ-protocols to be used, and how they should be composed (cf. §2.2 and §2.3). For a more detailed discussion of PSL see §3.2.

Protocol Compiler. The Protocol Compiler translates a PSL file into a protocol implementation formulated in our *Protocol Implementation Language* (*PIL*). This language can be thought of as a kind of pseudo-code describing the sequence of operations computed by both parties (including group operations, random choices, checks, etc.) and the messages exchanged.

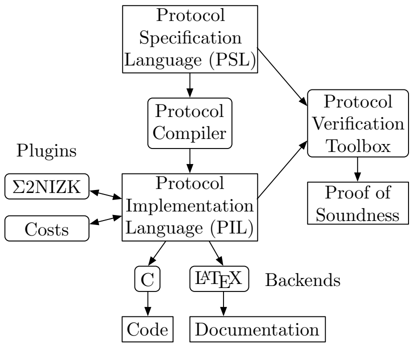
Protocol Specification Language (PSL)
Protocol Compiler
Protocol Verification Toolbox
Plugins
Σ2NIZK
Costs
Protocol Implementation Language (PIL)
Proof of Soundness
C
LaTeX
Backends
Code
Documentation

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

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 which can be compiled and linked together with the GNU multi-precision arithmetic library [54] into executable code. The LaTeX Backend generates a human-readable documentation of the protocol.

Protocol Verification Toolbox. This component takes as inputs a PSL file together with the corresponding PIL file obtained from a compilation run. It first extracts the necessary information for constructing a formal proof of the soundness property of the generated protocol. The theorem prover Isabelle/HOL [55,56] is then used to automatically check the generated formal proof. We give more details on this toolbox in §4.

Plugins. Currently, two plugins are available in our compiler. The Σ2NIZK-plugin transforms the generated protocol in a non-interactive version thereof by applying the technique from [51], cf. §2.3. Its functionality could easily be extended to signature proofs of knowledge. The Costs plugin determines the abstract costs of the protocol by computing the communication complexity and the number of group operations needed in any of the involved groups. This enables a comparison of the efficiency of different protocols already on an abstract level.

3.2 Protocol Specification Language and Optimizations

Next, we illustrate usage of our compiler (and, in particular, PSL) using the following informal statement from a group-oriented application as example:

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

For the example we assume that Pedersen commitments [14] were used to commit to m. Such commitments are of the form $c = g^m h^r$ for randomly chosen r. Here, $\langle g \rangle = \langle h \rangle = \mathcal{H}$ where $\mathcal{H}$ is a subgroup of prime order q of $\mathbb{Z}_p^*$ for some prime p. Further $\log_g h$ must not be known to the committing party.

```
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

To authenticate 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. For simplicity, we use the same group $\mathcal{H}$ for commitments and keys of users, but the compiler could use different groups as well.

Now, given $c, \mathrm{pk}_1, \mathrm{pk}_2$, the informal statement translates into this proof goal:

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

With homomorphisms $\psi : (a, b) \mapsto g^a h^b$ and $\phi : (a) \mapsto g^a$ we rewrite this as

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

where the atoms 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 blocks:

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

Inputs. Here, the inputs of both parties have to be specified. Only inputs that have been declared beforehand can be assigned.

Properties. This section specifies the properties of the protocol to be generated. For instance, `KnowledgeError := 80` specifies an intended knowledge error of $\kappa = 2^{-80}$. The proof goal can be specified by arbitrarily nesting atoms using Boolean And and Or operators. Furthermore, the compiler supports k-out-of-n-threshold compositions [49] based on Shamir secret sharing [50] (cf. §2.3).

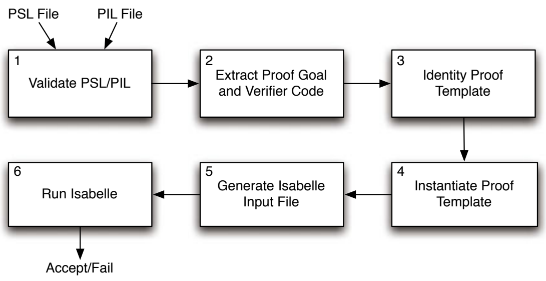
PSL File
PIL File
1 Validate PSL/PIL
2 Extract Proof Goal and Verifier Code
3 Identity Proof Template
4 Instantiate Proof Template
5 Generate Isabelle Input File
6 Run Isabelle
Accept/Fail

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

GlobalHomomorphisms. Homomorphisms that appear in multiple atoms can be defined globally in this optional section. Describing homomorphisms in PSL is a natural translation from their mathematical notation consisting of name, domain, co-domain, and the mapping function.

Predicates. Finally, the atoms used in `ProtocolComposition` are specified. Each predicate is proved with a Σ-protocol: one of `SigmaPhi`, `SigmaGSP` or `SigmaExp`. For each Σ-protocol, the relation between public and private values must be defined using local or global homomorphisms. `ChallengeLength` specifies the maximum challenge length that can be used to prove this atom with the given Σ-protocol (cf. §2.2 for details). Note that, in general, this value cannot be automatically determined by the compiler. It may depend, for example, on the size of the special exponent of the homomorphism, whose factorization might not be available. The compiler then automatically infers the required number of repetitions of each atom from this specification.

Optimizations. The compiler automatically transforms the proof goal in order to reduce its complexity. For instance, `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.

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. That is, it formally validates the guarantee obtained by a verifier executing the protocol: “The prover indeed knows a witness for 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 (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 group elements and homomorphisms[3].

The reason for the verification toolbox only considering the verifier code is that by definition [2] 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, 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 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) An adequate proof template is selected from those built into the tool (3). If no adequate template exists, 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) The proof assistant is executed on the theory file (6). If the proof assistant successfully finishes, then we have a formal proof of the theoretical soundness of the protocol. Isabelle/HOL also permits generating a human-readable version of the proof that can be used for product documentation.

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 [56] 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 employed and extended the Isabelle/HOL Algebra Library [57], 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 [55,56].

Remark. No verification is carried out of the executable code generated from the PIL file. This is a program correctness problem rather than a theoretical security problem, and must be addressed using different techniques not covered here.

We next detail the most important aspects of our approach.

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

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 §2.1, 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. §2.2) for which it is possible to efficiently find pseudo-preimages. As all special homomorphisms used in cryptography fall into one of two easily recognizable classes, the verification toolbox is able 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 1) that we have formalized in HOL.

Lemma 2 (Shamir's Trick [47]). *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 using the Extended Euclidean Algorithm to obtain $a, b \in \mathbb{Z}$ such that $av_1 + bv_2 = 1$, and then calculating $x = au_1 + bu_2$.*

Given a special homomorphism and two accepting protocol transcripts for a ZK-PoK of an atom, we prove the existence of a knowledge extractor by ensuring that we are able instantiate Lemma 2.

The PVT also supports Boolean And and Or composition. 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: 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 atom, 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 $\phi : G \rightarrow H$, quantified for all values of G: fix $g \in H$ with order q and assume $\forall a \in G.\ \phi(a) = g^a$. Furthermore, 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 $\phi(au + b\Delta s) = \mathrm{pk} \wedge av + b\Delta c = 1$, 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 for 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 PIL-statement:

```
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 that

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

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

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

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

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

where $\Delta c = c' - c''$ and $\Delta s = s' - s''$. We thus obtained $(\Delta s, \Delta c)$ as a first pseudo-preimage. The second one needed in Lemma 2 is found by analyzing the proof goal in the PSL file, which in our case was `Relation((pk) = phi(sk))`.

As we have embedded in our tool the domain specific knowledge to generate pseudo-preimages for the class of protocols that we formally verify, we can 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 2, 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 compositions 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 atoms. 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 for some $i \in \{1,\ldots,n\}$ 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 stated in §2.3.

Acknowledgments. We gratefully acknowledge Wilko Henecka and Andreas Grünert for their support on the implementation of the compiler. Also, we would like to thank Stefania Barzan for implementing a preliminary version of the PVT.

References

1. Almeida, J., Bangerter, E., Barbosa, M., Krenn, S., Sadeghi, A.R., Schneider, T.: A certifying compiler for zero-knowledge proofs of knowledge based on Σ-protocols. Cryptology ePrint Archive, Report 2010/339 (2010)
2. Bellare, M., Goldreich, O.: On defining proofs of knowledge. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 390–420. Springer, Heidelberg (1993)
3. Han, W., Chen, K., Zheng, D.: Receipt-freeness for Groth e-voting schemes. Journal of Information Science and Engineering 25, 517–530 (2009)
4. Kikuchi, H., Nagai, K., Ogata, W., Nishigaki, M.: Privacy-preserving similarity evaluation and application to remote biometrics authentication. Soft Computing 14, 529–536 (2010)
5. Camenisch, J.: Group Signature Schemes and Payment Systems Based on the Discrete Logarithm Problem. PhD thesis, ETH Zurich, Konstanz (1998)
6. Camenisch, J., Michels, M.: Proving in zero-knowledge that a number is the product of two safe primes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 107–122. Springer, Heidelberg (1999)
7. Brands, S.: Untraceable off-line cash in wallet with observers. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 302–318. Springer, Heidelberg (1994)
8. Lindell, Y., Pinkas, B., Smart, N.P.: Implementing two-party computation efficiently with security against malicious adversaries. In: Ostrovsky, R., De Prisco, R., Visconti, I. (eds.) SCN 2008. LNCS, vol. 5229, pp. 2–20. Springer, Heidelberg (2008)
9. Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. In: ACM CCS 2004, pp. 132–145. ACM Press, New York (2004)
10. Camenisch, J., Herreweghen, E.V.: Design and implementation of the idemix anonymous credential system. In: ACM CCS 2002, pp. 21–30. ACM Press, New York (2002)
11. Kunz-Jacques, S., Martinet, G., Poupard, G., Stern, J.: Cryptanalysis of an efficient proof of knowledge of discrete logarithm. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T.G. (eds.) PKC 2006. LNCS, vol. 3958, pp. 27–43. Springer, Heidelberg (2006)
12. Bangerter, E., Camenisch, J., Maurer, U.: Efficient proofs of knowledge of discrete logarithms and representations in groups with hidden order. In: Vaudenay, S. (ed.) PKC 2005. LNCS, vol. 3386, pp. 154–171. Springer, Heidelberg (2005)

13. Schnorr, C.: Efficient signature generation by smart cards. Journal of Cryptology 4, 161–174 (1991)
14. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 129–140. Springer, Heidelberg (1992)
15. Camenisch, J., Lysyanskaya, A.: An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 93–118. Springer, Heidelberg (2001)
16. Lipmaa, H.: On diophantine complexity and statistical zeroknowledge arguments. In: Laih, C.-S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 398–415. Springer, Heidelberg (2003)
17. Paulson, L.: Isabelle: a Generic Theorem Prover. Volume 828 of LNCS. Springer (1994)
18. MacKenzie, P., Oprea, A., Reiter, M.K.: Automatic generation of two-party computations. In: ACM CCS 2003, pp. 210–219. ACM, New York (2003)
19. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay — a secure two-party computation system. In: USENIX Security 2004 (2004)
20. Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous multiparty computation: Theory and implementation. In: Jarecki, S., Tsudik, G. (eds.) Public Key Cryptography – PKC 2009. LNCS, vol. 5443, pp. 160–179. Springer, Heidelberg (2009)
21. Briner, T.: Compiler for zero-knowledge proof-of-knowledge protocols. Master's thesis, ETH Zurich (2004)
22. Camenisch, J., Rohe, M., Sadeghi, A.R.: Sokrates - a compiler framework for zero-knowledge protocols. In: WEWoRC 2005 (2005)
23. Bangerter, E., Camenisch, J., Krenn, S., Sadeghi, A.R., Schneider, T.: Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471, Poster Session of EUROCRYPT 2009 (2008)
24. Bangerter, E., Briner, T., Heneka, W., Krenn, S., Sadeghi, A.R., Schneider, T.: Automatic generation of Σ-protocols. In: EuroPKI 2009 (to appear, 2009)
25. Bangerter, E., Krenn, S., Sadeghi, A.R., Schneider, T., Tsay, J.K.: 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 2009, October 12-13 (2009)
26. Meiklejohn, S., Erway, C., Küpçü, A., Hinkle, T., Lysyanskaya, A.: ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash. In: USENIX 10 (to appear, 2010)
27. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21, 120–126 (1978)
28. Backes, M., Maffei, M., Unruh, D.: Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In: IEEE Symposium on Security and Privacy – SP 2008, pp. 202–215. IEEE, Los Alamitos (2008)
29. Baskar, A., Ramanujam, R., Suresh, S.P.: A dolev-yao model for zero knowledge. In: Datta, A. (ed.) ASIAN 2009. LNCS, vol. 5913, pp. 137–146. Springer, Heidelberg (2009)
30. Blanchet, B.: ProVerif: Cryptographic protocol verifier in the formal model (2010)
31. Backes, M., Hritcu, C., Maffei, M.: Type-checking zero-knowledge. In: ACM CCS 2008, pp. 357–370. ACM, New York (2008)

32. Backes, M., Unruh, D.: Computational soundness of symbolic zero-knowledge proofs against active attackers. In: IEEE Computer Security Foundations Symposium - CSF 2008, 255–269 Preprint on IACR ePrint 2008/152 (2008)
33. Barthe, G., Hedin, D., Zanella Béguelin, S., Grégoire, B., Heraud, S.: A machine-checked formalization of Σ-protocols. In: 23rd IEEE Computer Security Foundations Symposium, CSF 2010, IEEE, Los Alamitos (2010)
34. Barthe, G., Grégoire, B., Béguelin, S.: Formal certification of code-based cryptographic proofs. In: ACM SIGPLAN-SIGACT POPL 2009, pp. 90–101 (2009)
35. Goubault-Larrecq, J., Parrennes, F.: Cryptographic protocol analysis on real C code. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 363–379. Springer, Heidelberg (2005)
36. Bhargavan, K., Fournet, C., Gordon, A., Tse, S.: Verified interoperable implementations of security protocols. ACM Trans. Program. Lang. Syst. 31(1), 1–61 (2008)
37. Bhargavan, K., Fournet, C., Corin, R., Zalinescu, E.: Cryptographically verified implementations for TLS. In: ACM CCS 2008, pp. 459–468. ACM, New York (2008)
38. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Workshop on Computer Security Foundations – CSFW 2001, p. 82. IEEE, Los Alamitos (2001)
39. Blanchet, B.: A computationally sound mechanized prover for security protocols. In: IEEE Symposium on Security and Privacy – SP 2006, pp. 140–154. IEEE, Los Alamitos (2006)
40. Camenisch, J., Stadler, M.: Efficient group signature schemes for large groups (extended abstract). In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 410–424. Springer, Heidelberg (1997)
41. Cramer, R.: Modular Design of Secure yet Practical Cryptographic Protocols. PhD thesis, CWI and University of Amsterdam (1997)
42. Damgård, I.: On Σ-protocols, Lecture on Cryptologic Protocol Theory, Faculty of Science, University of Aarhus (2004)
43. Guillou, L., Quisquater, J.J.: A "paradoxical" identity-based signature scheme resulting from zero-knowledge. In: Goldwasser, S. (ed.) CRYPTO 1988. LNCS, vol. 403, pp. 216–231. Springer, Heidelberg (1990)
44. Cramer, R., Shoup, V.: A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 13–25. Springer, Heidelberg (1998)
45. Fujisaki, E., Okamoto, T.: Statistical zero knowledge protocols to prove modular polynomial relations. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 16–30. Springer, Heidelberg (1997)
46. Damgård, I., Fujisaki, E.: A statistically-hiding integer commitment scheme based on groups with hidden order. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 77–85. Springer, Heidelberg (2002)
47. Bangerter, E.: Efficient Zero-Knowledge Proofs of Knowledge for Homomorphisms. PhD thesis, Ruhr-University Bochum (2005)
48. Smart, N.P. (ed.): Final Report on Unified Theoretical Framework of Efficient Zero-Knowledge Proofs of Knowledge. CACE project deliverable (2009)
49. Cramer, R., Damgård, I., Schoenmakers, B.: Proofs of partial knowledge and simplified design of witness hiding protocols. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 174–187. Springer, Heidelberg (1994)
50. Shamir, A.: How to share a secret. Communications of the ACM 22, 612–613 (1979)
51. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987)

52. Brands, S.: Rapid demonstration of linear relations connected by boolean operators. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 318–333. Springer, Heidelberg (1997)
53. Bresson, E., Stern, J.: Proofs of knowledge for non-monotone discrete-log formulae and applications. In: Chan, A.H., Gligor, V.D. (eds.) ISC 2002. LNCS, vol. 2433, pp. 272–288. Springer, Heidelberg (2002)
54. Granlund, T.: The GNU MP Bignum Library (2010), `http://gmplib.org/`
55. Nipkow, T., Paulson, L.: Isabelle (2010), `http://isabelle.in.tun.de`
56. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
57. Ballarin, C., Kammüller, F., Paulson, L.: The Isabelle/HOL Algebra Library (2008), `http://isabelle.in.tum.de/library/HOL/HOL-Algebra/document.pdf`