Automatic Generation of Sigma-Protocols*

Endre Bangerter[1], Thomas Briner[2], Wilko Henecka[3], Stephan Krenn[4], Ahmad-Reza Sadeghi[3], and Thomas Schneider[3]

[1] Bern University of Applied Sciences, Biel-Bienne, Switzerland
endre.bangerter@bfh.ch
[2] Abraxas Informatik AG, Zürich, Switzerland
thomas.briner@gmail.com
[3] Horst Görtz Institute for IT Security, Ruhr-University Bochum, Germany
wilko.henecka@rub.de,{ahmad.sadeghi,thomas.schneider}@trust.rub.de
[4] Bern University of Applied Sciences, Biel-Bienne, Switzerland, and University of Fribourg, Switzerland
stephan.krenn@bfh.ch

Abstract. Efficient zero-knowledge proofs of knowledge (ZK-PoK) are basic building blocks of many cryptographic applications such as identification schemes, group signatures, and secure multi-party computation. Currently, first applications that essentially rely on ZK-PoKs are being deployed in the real world. The most prominent example is the Direct Anonymous Attestation (DAA) protocol, which was adopted by the Trusted Computing Group (TCG) and implemented as one of the functionalities of the cryptographic chip Trusted Platform Module (TPM).

Implementing systems using ZK-PoK turns out to be challenging, since ZK-PoK are significantly more complex than standard crypto primitives (e.g., encryption and signature schemes). As a result, the design-implementation cycles of ZK-PoK are time-consuming and error-prone.

To overcome this, we present a compiler with corresponding languages for the automatic generation of sound and efficient ZK-PoK based on Σ-protocols. The protocol designer using our compiler formulates the goal of a ZK-PoK proof in a high-level protocol specification language, which abstracts away unnecessary technicalities from the designer. The compiler then automatically generates the protocol implementation in Java code; alternatively, the compiler can output a description of the protocol in LaTeX which can be used for documentation or verification.

Keywords: Zero-Knowledge, Protocol Compiler, Language Design.

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 some secret values (proof of knowledge property), without the verifier

* This work was in part funded by the European Community's FP7 projects CACE and ECRYPT II under grant agreements no. 216499 and 216676.

learning anything about them beyond what was known before the protocol run (zero-knowledge property). There are fundamental results showing that all relations in NP have ZK-PoK [1,2,3]. The corresponding protocols are of theoretical relevance, but much too inefficient to be used in practical applications.

In contrast to these generic protocols for arbitrary NP statements we concentrate on a subset of practically relevant relations that can be proven with practically efficient protocols. Essentially, all efficient ZK-PoK protocols used in practice today are based on a class of three move protocols, called Σ-protocols.

Basic Σ-protocols allow to prove knowledge of a secret preimage under a homomorphism (e.g., a discrete exponentiation or an RSA function). There are numerous variations of these preimage proofs. For instance, "AND-proofs" allow to prove simultaneous knowledge of multiple preimages under different homomorphisms. Similarly there are "OR-proofs" and proofs to show that different preimages fulfill a set of linear relations.

ZK-PoK proof techniques based on Σ-protocols play an important role in applied cryptography. In fact, many practically oriented applications use such proofs as basic building blocks. Examples therefore include identification schemes [4], interactive verifiable computation [5], group signatures [6], secure watermark detection [7], and efficient secure multiparty computation [8].

While many of these applications typically only exist on a specification level, a direction of applied research has produced first real-world applications using ZK-PoKs. One prominent example is the Direct Anonymous Attestation (DAA) protocol [9], which was adopted by the Trusted Computing Group (TCG) – an industry consortium of many IT enterprises – as a privacy enhancing mechanism for remote authentication of computing platforms. Another example is the identity mixer anonymous credential system [10], which was released by IBM into the Eclipse Higgins project, an open source effort dedicated to developing software for "user-centric" identity management.

Up to now, the design and implementation of practical ZK-PoK protocols is done "by hand". The security proofs of these protocols consist of, loosely speaking, a handful of standard arguments and tricks which are repeated in different constellations over and over again. In fact, past experiences, e.g., during the development of the previous two examples have shown the following:

- Implementation cycles of ZK-PoK are time-consuming and error-prone.
- Minor changes in the protocol specification can result in substantial implementation work, i.e., it is hard to achieve resilience against design modifications.
- Protocols are often designed by cryptographers and implemented by software engineers. The former typically are not skilled in implementation matters and the latter have a hard time understanding details and subtleties of ZK-PoK protocols, which are sometimes rather complex. This can lead to a rupture between design and implementation, resulting in implementation errors.

Our Contributions. To overcome the mentioned challenges, we have designed and implemented a language and a corresponding compiler. Given a high-level

ZK-PoK protocol specification in our language, the compiler automatically generates the implementation of the corresponding Σ-protocol.

The design of the language is inspired by the widely used Camenisch-Stadler notation [11]. It allows to specify Σ-protocols and compositions (e.g. *AND*, *OR*) thereof, while it abstracts away details that are unnecessary at a protocol design level. Since the Camenisch-Stadler notation is informal and incomplete, our language contains additional elements, denoting, e.g., the algebraic setting in which the proofs are carried out.

ZK-PoK protocol specifications in this language are then translated by the compiler either into Java or LaTeX code. The group operations in the generated code are expressed in terms of abstract interfaces. This allows users of the code to plug their preferred libraries or favorite algebraic groups into the protocol code by implementing our abstract interfaces. The LaTeX code can be used for documenting the protocols and also for verification purposes. To the best of our knowledge, this is the first compiler suite to support automatic generation of sound ZK-PoK protocols.

The current version of the compiler allows to generate a large number of protocols found in the literature, including Pedersen Commitments/Verifiable Secret Sharing [12], Schnorr Authentication/Signatures [4], proof showing that a number is the product of two safe primes [5], Electronic Cash [13,14,15], Group Signatures [16], and Ring Signatures [17]. Also supported are ZK-PoKs of a plaintext corresponding to a ciphertext or relations between plaintexts under various asymmetric encryption schemes such as, RSA [18], Paillier [19], or Damgård-Jurik [20]; these homomorphic encryption schemes are widely used in e-voting and secure multiparty computation.

The existing theory and collection of ZK-PoK proof techniques using Σ-protocols is vast, and a satisfactory unified theory underlying these techniques is missing. In fact, for some of these techniques it is not clear whether and how they can be combined in a modular way. To design the input language and compiler on solid theoretical grounds, we have put together a unified framework of existing proof techniques. This framework is simple to understand, modular and encompasses a large number of existing ZK-PoK. The basis of the framework are simple proofs of knowledge of preimages under homomorphisms. For these basic proofs, we have incorporated the theory by Cramer [21] on special homomorphisms, which are essentially homomorphisms with a known order co-domain as well as RSA and Paillier-type of homomorphisms. Our framework then describes how the basic protocols can be composed to obtain "AND" and "AND-OR" proofs, and to prove linear relations among preimages.

Related Work. This paper describes ongoing work on the zero-knowledge compiler initiated by [22] which focused mainly on the implementation details of the compiler. The motivation for having a compiler framework for zero-knowledge protocols was described in [23]. In this paper we describe the underlying theoretical framework and how to use the fixed and slightly extended (e.g., native support of groups $\mathbb{Z}_n^*$) compiler based on a concrete running-example. An earlier draft of this paper was presented at the poster session of Eurocrypt 2009 [24].

An analysis of Σ-protocols for special homomorphisms can be found in [21], and the used composition rules are explained in [17]. A first framework for Boolean formulae containing linear relations was done by Brands [25] and extended in [26] to a larger class of predicates. The idea underlying our proofs for linear relations is the same as in [27]. A unified theory for exponentiation homomorphisms in arbitrary groups has recently been published [28] which we plan to incorporate into future versions of the compiler. Yet, this does not influence proofs for special homomorphisms, for which our compiler is currently designed.

In principle, zero-knowledge can be obtained from secure multiparty computation (SMPC) by evaluating the corresponding verification relation securely [2]. While this allows to prove arbitrary NP statements in zero-knowledge in communication and computation complexity which is linear in the circuit size, this approach is limited in practice by the circuit size (today's implementations of generic SMPC techniques can evaluate circuits with a few million gates only [8,29,30]). The Σ-protocols generated by our compiler are much more efficient but limited to a smaller, yet useful, class of statements that can be proven.

Provably secure protocols for two-party secure function evaluation (SFE) based on homomorphic encryption [31] respectively circuits [29,32,33] can be generated automatically. Similar to what our compiler does in the context of ZK-PoK protocols, these compilers allow to specify the function to be evaluated in a high-level language and automatically compile this into an executable protocol. In order to achieve security against malicious participants, cut-and-choose techniques together with efficient zero-knowledge proofs are added to prove that parties behave honestly [8,34]. Recently, highly efficient protocols combining subprotocols based on homomorphic encryption with such based on circuits were proposed. To secure the conversion between both domains against malicious players they make use of efficient ZK-PoK [35]. Our compiler can be used to generate these ZK-PoK protocols at the interfaces between different protocols.

A specification language at the implementation level of cryptographic primitives is Cryptography Aware Language and Compiler (CAO) [36]. This framework provides compiler support for efficient and secure implementation of cryptographic primitives resistant against software side-channels [37] and applications to elliptic curve cryptography [38]. In future versions of our compiler we plan to automatically generate implementations of our generated protocols also in CAO.

Overall, our compiler for automatic generation of sound ZK-PoK protocols can be positioned in between the (high-level) compilers for secure computation [31,29,33] and the (low-level) compilers to automatically generate implementations of cryptographic primitives [36].

Outline. In §2 we describe the theoretical framework of Σ-protocols underlying our compiler. In §3 we describe the compiler and its input language. Particularly, we give a detailed example showing how our compiler can be used to prove relations among messages encrypted with the Damgård-Jurik [20] cryptosystem.

2 General Framework Description

Our compiler can be used to generate protocols for honest-verifier zero-knowledge (HVZK) proofs of knowledge of preimages under homomorphisms. These proofs can be combined arbitrarily using the Boolean operators AND and OR, which allows proving knowledge of certain subsets of preimages. Further, homogeneous linear relations among the preimages can be proven. In this section we want to briefly recap the theory underlying the compiler as well as the techniques we've implemented. After giving some basic notation and definitions in §2.1, we will formally describe the class of proofs for which the compiler produces HVZK proofs of knowledge in form of Σ-protocols in §2.2 and review the techniques we implemented together with sufficient conditions guaranteeing soundness in §2.3. Finally in §2.4 we will conclude by showing how these results can be used to prove more complex relations among the preimages, such as multiplicative or polynomial ones.

2.1 Preliminaries

By $s \in_R S$ we denote a uniform random choice of element s from set S. The cardinality of S is denoted by $\#S$. A mapping $\phi : \mathcal{G} \rightarrow \mathcal{H}$ from an additive group $(\mathcal{G}, +)$ into a multiplicative group $(\mathcal{H}, \cdot)$ is called *homomorphism*, iff for all $a, b \in \mathcal{G}$ we have $\phi(a + b) = \phi(a) \cdot \phi(b)$. By Im ϕ we denote the *image of* ϕ, i.e., Im $\phi = \{z \in \mathcal{H} : \exists w \in \mathcal{G} : z = \phi(w)\}$, which is a subgroup of $\mathcal{H}$.

Next we briefly recap the notion of zero-knowledge proofs of knowledge, and that of Σ-protocols which our compiler uses to implement them.

Let R be a binary relation and let $(x, w) \in R$, where w is a witness and x an element of the associated language L_R. Informally, a *proof of knowledge* with *knowledge error* κ for R is a pair of interactive algorithms (P, V), such that every (potentially dishonest) prover P^* who on input x can make verifier V accept with probability more than $\kappa(x)$, has to know a w', such that $(x, w') \in R$; further, V always accepts for the honest prover P. A formal definition is given in [39].

All protocols generated by our compiler are *Σ-protocols*. Informally, a Σ-protocol is a protocol with 3 messages being exchanged: the prover sends a *commitment* t to V, who replies with a random *challenge* c from a predefined challenge set $\mathcal{C}$. Then P computes a *response* s, which V uses to decide whether to accept or to reject the proof. The protocol must satisfy three properties: First, the verifier always accepts for an honest prover. Second, having two tuples (t, c, s), (t, c', s') with $c \neq c'$ for which the verifier accepts, it's possible to efficiently compute a witness. Finally, the protocol is HVZK. It turns out that from the form of the protocol and the first two properties, the proof of knowledge property can be implied. For a more detailed discussion of Σ-protocols see, e.g., [21].

Notation of ZK-PoKs. Using the notation introduced in [11] to denote ZK-PoKs, a term like

$$\mathsf{ZPK}\Big[(\omega_1, \omega_2) : x_1 = \phi_1(\omega_1) \quad \wedge \quad x_2 = \phi_2(\omega_2) \quad \wedge \quad \omega_1 = a\omega_2\Big]$$

means "*proof of knowledge of* w_1, w_2 *such that* $x_1 = \phi_1(w_1)$, $x_2 = \phi_2(w_2)$ *and* $w_1 = aw_2$". We will stick to the common convention that knowledge of variables denoted by Greek letters has to be proven, whereas all other quantities are assumed to be known to both parties, i.e. P and V. Note that this notation specifies a *proof-goal* rather than a protocol: it describes what actually has to be proven, but there may be many differently efficient protocols for the same proof-goal.

2.2 Proof-Goals Supported by Our Compiler

The compiler described in §3 can be used to generate implementations for HVZK proofs of knowledge of preimages under homomorphisms. The proofs can be combined arbitrarily using the Boolean operators "AND" and "OR", which allows proving knowledge of sets respectively subsets of preimages. Also homogeneous linear relations among the preimages can be proven.

That is, the class of proof-goals that can be handled by our compiler consists of all expressions that can be expressed in one of the following two forms:

$$\mathsf{ZPK}\Big[(\omega_1, \ldots, \omega_m) : \bigvee \bigwedge y_i = \phi_i(\omega_i)\Big)\Big] \tag{1}$$

or

$$\mathsf{ZPK}\Big[(\omega_1, \ldots, \omega_m) : \bigwedge y_i = \phi_i(\omega_1, \ldots, \omega_m) \wedge HLR(\omega_1, \ldots, \omega_m)\Big] \tag{2}$$

Here, $HLR(w_1, \ldots, w_m)$ denotes a system of homogeneous linear relations among the preimages. That is, it consists of a set of equations of the following form:

$$w_i = \sum_{j>i} a_{ij} w_j \qquad \text{with} \qquad a_{ij} \in \mathbb{Z}.$$

We want to make some remarks on the specification on the proof-goals: first, in (1), the proof-goal does not necessarily have to be given in disjunctive normal form (DNF), but also as arbitrary monotone Boolean formula, i.e. a Boolean formula containing arbitrarily many $\wedge$ and $\vee$ with predicates of the form $y_j = \phi_j(\omega_j)$. Second, in (1) as well as in (2), linear relations can also be proven *implicitly*: for instance, it's easy to see that $\mathsf{ZPK}\big[(\omega_1, \omega_2) : y = \phi(\omega_1, \omega_2) \wedge \omega_1 = 2\omega_2\big]$ is equivalent to $\mathsf{ZPK}\big[(\omega) : y = \phi(2\omega, \omega)\big]$ by setting $w := w_2$. Finally, note that the group w_i lies in can decompose into a product of groups. That is, w_i can denote a vector $(w_{i1}, \ldots, w_{ik_i})$ of elements.

2.3 Implemented Techniques and Soundness Conditions

In this section we briefly describe which techniques we implemented in our compiler, and point out when our compiler makes use of them.

AND-proofs. An *AND-proof* allows to prove knowledge of multiple preimages, i.e., it is used to prove a semantic goal like (2) without linear relations. Such a proof can be realized by considering the product homomorphism of the ϕ_i, and proving knowledge of a preimage of this as follows:

- The compiler defines $\mathcal{G} := \mathcal{G}_1 \times \cdots \times \mathcal{G}_m$, and $\mathcal{H} := \mathcal{H}_1 \times \cdots \times \mathcal{H}_m$.
- It sets $\phi : \mathcal{G} \to \mathcal{H}$, $\phi(w_1, .., w_m) := (\phi_1(w_1, .., w_m), .., \phi_m(w_1, .., w_m))$.
- Further, it defines $w := (w_1, \ldots, w_m)$ and $x := (x_1, \ldots, x_m)$.
- Finally, it performs the following proof: $\mathsf{ZPK}\big[(\omega) : x = \phi(\omega)\big]$.

AND-OR-Proofs. An *AND-OR-proof* is capable of proving knowledge of preimages corresponding to one out of a family of given subsets of $\{x_1, \ldots, x_m\}$. That is, it can be used to proof expressions like (1). In this case, the proof goal is first translated into disjunctive normal form (DNF), and then each conjunctive term is proved using the technique described before. The OR-proof is then performed using the technique of [17] based on Shamir's secret sharing scheme [40].

Linear Relations. If linear constraints occur in (2), the compiler uses a technique which is very similar to that for "AND"-proofs [27]. It is based on the observation that the set of all elements in $\mathcal{G} := \mathcal{G}_1 \times \cdots \times \mathcal{G}_m$ satisfying the linear constraints in (2) is a subgroup of $\mathcal{G}$. Thus, by denoting this set by $\hat{\mathcal{G}}$ the same technique as for AND-proofs can be used with $\hat{\mathcal{G}}$ instead of $\mathcal{G}$.

We stress that because of the form of the equation system random choices in $\hat{\mathcal{G}}$ can be drawn efficiently by forward substitution.

Sufficient conditions to guarantee soundness. It is a well known result that all Σ-protocols for preimage proofs under homomorphisms with finite domain are HVZK proofs of knowledge for the challenge set $\mathcal{C} = \{0, 1\}$ [21] . Yet, this only guarantees a knowledge error of $\kappa = 1/2$ and many repetitions are necessary to reach a sufficiently small knowledge error in most applications.

It turns out that for certain homomorphisms we can obtain much more efficient proofs, since they allow to obtain a small knowledge error in a single protocol run. Consider an homomorphism ϕ, for which a non-zero multiple v of the order of Im ϕ is known: then we have that $x^v = 1 = \phi(0)$ for all $x \in \text{Im } \phi$. Especially, if $\text{ord}(\mathcal{H})$ is known, one can set $v := \text{ord}(\mathcal{H})$. Such homomorphisms are used in [4]. The authors of [41] use power homomorphisms $\phi : \mathbb{Z}_n^* \to \mathbb{Z}_n^*, x \mapsto x^e$ where n is an RSA modulus and $e \in \mathbb{Z}$. There we have $x^e = \phi(x)$ for all x. In both cases it's feasible to find a preimage of a power of x for each $x \in \text{Im } \phi$. This property is caught by the following definition:

Definition 1 (Special Homomorphism [21]). *A homomorphism ϕ is called* special*, if there is a probabilistic polynomial time algorithm that on input $\phi : \mathcal{G} \to \mathcal{H}$ and $x \in \text{Im } \phi$ outputs $(u, v) \in \mathcal{G} \times \mathbb{Z} \setminus \{0\}$, such that $x^v = \phi(u)$. For a fixed ϕ, the* special exponent *v being output has to be the same for all x.*

Building on this definition, we get the following theorem giving conditions for the Σ-protocols produced by our compiler to be sound:

Theorem 1. *The composition techniques described above result in HVZK proofs of knowledge with knowledge error $1/\#\mathcal{C}$ for (1) or (2), if the following conditions are satisfied:*

– *All ϕ_i, $i = 1, \ldots, m$ are special, and the special exponent v_i of ϕ_i satisfies $v_i \le \max(\mathcal{C})$.*
– *If the preimage of ϕ_j occurs in one of the homogeneous linear relations in (2), the special exponent of ϕ_j is a non-zero multiple of the order of* $\mathrm{Im}\ \phi_j$.

Proof (Sketch). The case of proving knowledge of only one preimage is handled in, e.g., [21,42], by using Shamir's trick. By observing that the product of special homomorphisms is again special with a special exponent equal to the product of the special exponents of its factors, the correctness of the AND-composition follows. With a similar argument, the soundness for the case of linear equations can be inferred [27]. Finally, the proof for proof goals containing ORs can be found in [17]. □

2.4 Proving More Complex Relations

Using our compiler even more complex proof goals than pure preimage proofs (optionally containing homogeneous linear relations) can be realized. On a high level, all proof goals having an equivalent representation as preimage proofs containing only homogeneous linear relations can be handled. Yet, this rewriting has to be manually by the user of our compiler. We thus illustrate on hand of two practically important classes of relations how this can be done.

Example 1 (Multiplicative Relations modulo ord(Im ϕ)*).* To prove knowledge of the discrete logarithms w_1, w_2, w_3 of x_1, x_2, x_3 in base g, satisfying $w_1 w_2 = w_3 \bmod \mathrm{ord}(\mathrm{Im}\ \phi)$ one can perform the following "AND"-proof with one implicit linear relation:

$$\mathsf{ZPK}\Big[(\omega_1, \omega_2) : x_1 = g^{\omega_1} \ \wedge\ x_2 = g^{\omega_2} \ \wedge\ x_3 = x_1^{\omega_2}\Big].$$

If P can convince V that he knows such w_1, w_2, it is clear that he knows the discrete logarithms of x_1 and x_2. Further, we can infer the following: $x_3 = x_1^{w_2} = (g^{w_1})^{w_2} = g^{w_1 w_2}$. Hence, P knows the discrete logarithm of x_3 in base g, and it is equal to $w_1 w_2$. That is what had to be proven.

Example 2 (Inhomogeneous Linear Relations). Inhomogeneous linear relations can easily be homogenized [25] by using the homomorphic property of ϕ: for instance, proving knowledge of w_1, w_2 such that $x_i = \phi(w_i)$, and $w_1 = w_2 + c$ for a fixed $c \in \mathcal{G}$ is equivalent to performing

$$\mathsf{ZPK}\Big[(\omega) : x_1 = \phi(\omega) \ \wedge\ x_2 \cdot \phi(c)^{-1} = \phi(\omega)\Big].$$

We remark that by combining these two techniques, arbitrary polynomial relations modulo the order of Im ϕ among the secret preimages can be proved. Finally, we note that proving that a certain relation is *not* satisfied, e.g., that two discrete logarithms are not equal, requires a little more effort, as no equivalent representations in form of pure preimage proofs are known for such proof goals. Thus, the source code of the last round of the verifier has to be edited, and a simple check for inequality of two values has to be added manually. For a description of techniques handling such proof goals see, e.g., [26].

In the next section we describe how our current compiler implements the described general framework and give a practical example.

3 Implementation of Our ZK-PoK Compiler

We have implemented a compiler that can automatically generate Σ-protocols according to the theoretical framework described in §2. The initial version of the compiler was started in [22,23]. In this work we describe how to use the compiler with a concrete example.[1] The compiler is used as follows (cf. Fig. 1):

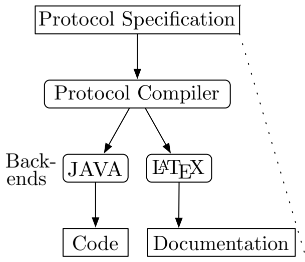
– The user formulates the *Protocol Specification* of the intended Σ-protocol in our high-level *input language*. This language abstracts away all implementation details, e.g., how to combine protocols, operations performed within algorithms, or messages to be exchanged. It allows to describe all expressions of the language discussed in §2 and is inspired by the Camenisch-Stadler notation [11], but augmented so that one can actually generate code. This is impossible directly from the Camenisch-Stadler notation as it does not contain information on the underlying algebraic structures. More details on the input language will be given later in §3.1.
– Then, the *Protocol Compiler* automatically transforms this protocol specification into the corresponding implementation of the protocol.
– This protocol implementation can be output as JAVA-code which can easily be incorporated into other applications that use the corresponding ZK-PoK protocol. Alternatively, a LaTeX documentation which shows the detailed steps (e.g., inputs, algorithms, operations, messages) of the protocol can be generated. The compiler was designed modularly to be easily extendible with other back-ends, e.g., to produce C-code for embedded platforms.

3.1 Input Language

Below, we describe the rationale underlying the input language and how to use it to formulate a proof goal based on the following running example:

Many protocols for secure computation use the semantically-secure, additively-homomorphic encryption scheme of Paillier [19] which was extended by Damgård and Jurik [20]. Recall, in this scheme encryption is performed as $E(m, r) = g^m \cdot r^n \bmod n^2$ with message $m \in \mathbb{Z}_n$, randomness $r \in_R \mathbb{Z}_n^*$, and public key n, where n is a RSA modulus and $g := n+1 \in \mathbb{Z}_{n^2}^*$. This scheme allows to add values under encryption, i.e., $E(a)E(b) = E(a + b)$, where the operations are performed in the ciphertext group $\mathbb{Z}_{n^2}^*$ respectively plaintext group $\mathbb{Z}_n$. This property allows to compute linear operations on ciphertexts (crypto-computing) and is used in many protocols such as [35,43,44] - just to name a few. The security against

[1] The compiler together with a formal syntactic definition of the input language as EBNF is available at `http://zkc.cace-project.eu`.



```
// Declarations
  Group Zn, Zm*;                              // L1
  GroupElement g,x_[1..2],rho_[0..3],mu;      // L2
  Homomorphism phi_[0..3];                    // L3
  IntegerConstant n;                          // L4
// Assignments
  AssignGroupMember(Zn,mu);                               // L5
  AssignGroupMember(Zm*,{g,x_[1..2],rho_[0..3]}); // L6
// Definitions
  DefineHomomorphism(phi_0, (rho_0) |-> (rho_0^n));           // L7
  DefineHomomorphism(phi_1, (rho_1) |-> (rho_1^n));           // L8
  DefineHomomorphism(phi_2, (mu,rho_2) |-> (g^mu * rho_2^n)); // L9
  DefineHomomorphism(phi_3, (mu,rho_3) |-> (g^mu * rho_3^n)); // L10
// Protocol Specification
  SpecifyProtocol [                                                    // L11
    Relation = ([(x_1)=phi_0(rho_0)] || [(x_1*g^(-1))=phi_1(rho_1)]) // L12
      || ([(x_1)=phi_2(mu,rho_2)] && [(x_2)=phi_3(mu,rho_3)]);      // L13
    Target = LATEX;                                                    // L14
  ]                                                                    // L15
```

Fig. 1. Architecture and Example for Protocol Specification in Input Language

honest-but-curious adversaries of such protocols follows from the semantic security of the encryption scheme, whereas for security against malicious adversaries each party usually needs to prove in zero-knowledge that it behaved correctly.

The following example is inspired by the application scenario described above. It does not correspond to a published protocol but is rather chosen to demonstrate many features of our compiler. One party proves in ZK that a generated ciphertext x_1 is either an encryption of 0 or 1 (this need arises for example in oblivious transfer protocols based on Paillier encryption [45]), or it encrypts the same plaintext μ encrypted as another ciphertext x_2 (this could be used to prove that the encrypted message is consistent with a previous encrypted message). More formally, this proof goal is written in Camenisch-Stadler notation [11] as

$$\mathsf{ZPK}\Big[(\mu,\rho_{0..3}) : (x_1 = E(0,\rho_0) \vee x_1 = E(1,\rho_1)) \\ \vee (x_1 = E(\mu,\rho_2) \wedge x_2 = E(\mu,\rho_3))\Big].$$

Plugging in the explicit definitions of the encryption function yields

$$\mathsf{ZPK}\Big[(\mu,\rho_{0..3}) : (x_1 = \rho_0^n \vee x_1 g^{-1} = \rho_1^n) \vee (x_1 = g^\mu \rho_2^n \wedge x_2 = g^\mu \rho_3^n)\Big]. \qquad (3)$$

However, the proof goal given in Camenisch-Stadler notation is not yet explicit enough for automatic generation of protocols as it is a semi-formal notation which does not contain the involved algebraic structures which is essential for the generation. For this, the input language of our compiler requires explicit `Declarations` of the involved algebraic objects (groups, elements, homomorphisms, constants), `Assignments` from group elements to the group they live in, as well as `Definitions` of homomorphisms which encapsulate functions with homomorphic properties as described next. In the following we refer to the line numbers (L...) of the example given in Fig. 1. These line numbers are comments which are separated with `//` in our input language.

Declarations (L1-L4): In the beginning the name of each group (L1), group element (L2), homomorphism (L3), and integer constant (L4) used in the protocol must be declared. As in L1, multiple elements can be separated with a comma. For convenience, multiple elements can be grouped together with array notation, e.g., in L2 where `x_[1..2]` is a shortcut for `x_1,x_2`. The integer constant `n` in L4 will later be set to the RSA modulus n in the implementation.

The compiler supports additive groups $(\mathbb{Z}_n, +)$ defined as `Zn` as well as multiplicative groups $(\mathbb{Z}_m^*, *)$ defined as `Zm*` (L1). The single letter following the capital `Z` is the name of the modulus which must be set to the corresponding value during runtime. In our example, `n` would be set to the RSA modulus n, whereas `m` would be set to n^2. Future versions of the compiler will allow to express such relations as arbitrary expressions already in the input language.

Assignments (L5-L6): Each group element declared before must be assigned to a group in this section, i.e. `mu` to `Zn` in L5. To assign multiple group elements to the same group, they can be put in curly braces (L6).

Definitions (L7-L10): As described in §2, efficient Σ-protocols can be generated to prove knowledge of preimages under homomorphisms. To allow automatic generation of such Σ-protocols, the user identifies the homomorphisms in the proof goal in equation (3) and writes it as

$$\mathsf{ZPK}\Big[(\mu, \rho_{0..3}) : (x_1 = \phi_0(\rho_0) \vee x_1 g^{-1} = \phi_1(\rho_1)) \\ \vee (x_1 = \phi_2(\mu, \rho_2) \wedge x_2 = \phi_3(\mu, \rho_3))\Big], \tag{4}$$

where e.g., $\phi_2 : (\mu, \rho_2) \mapsto g^\mu \rho_2^n$. This homomorphism is specified in our input language (L9), where the first parameter is the name of the homomorphism `phi_2` followed by the list of preimages `(mu,rho_2)` and finally the mapping from preimages to images as term `g^mu * rho_2^n`. The compiler automatically infers domain and co-domain of the homomorphism from the involved group elements which have been assigned to groups in the `Assignments` section. Using this information, the compiler checks that the group operations in the mapping are written correctly to avoid errors in the input specification. In additive groups, `+` denotes the group-operation, and `*` the multiplication with a scalar. In multiplicative groups (as `Zm*` in the example), `*` and `^` are handled analogously.

Protocol Specification (L11-L15): After having declared, assigned and defined all needed components, the protocol to be generated can be specified in the `SpecifyProtocol [...]` block (L11-L15):

For this, the relation to be proven - rewritten to use homomorphisms (4) - is formulated one-to-one in the input language (L12-L13). Boolean compositions are written as in the C language, i.e., AND composition as `&&` and OR composition as `||`. If this expression is not explicitly given in the disjunctive normal form (DNF) as in (1) the compiler transforms it automatically into this form.

Finally, a back-end of the compiler is chosen by specifying the output target. In the example, we chose the `LATEX` back-end in L14 to automatically generate the LaTeX documentation given in §A from the protocol specification in Fig. 1.

Alternatively, setting the target to `JAVA` would produce Java source code for the generated Σ-protocol. The Java code corresponds to the algorithms of the Σ-protocol for prover and verifier $(\mathsf{P}_1, \mathsf{P}_2, \mathsf{V})$ that can easily be integrated into user applications. Some parameters that can not yet be inferred by the compiler automatically (like the size of the challenge set) must be chosen by the user according to the theory described in §2 and provided as constructor arguments.

Yet, this does not cause much effort to the user: for instance, for every $x \in \operatorname{Im} \phi_2$ we have that $(0, x)$ satisfies $x^n = \phi_2(0, x)$, and thus ϕ_2 is special with special exponent n, cf. Def. 1. The same holds for ϕ_0, ϕ_1, ϕ_3. Hence, the maximum c^+ of the challenge set has only to be chosen smaller than any prime divisor of n. But as n is an RSA-modulus, all its divisors have some hundred bits, and c^+ should have about 80 bits in practical applications. Hence, choosing $c^+ := 2^{80}$ satisfies the conditions of Th. 1, and one gets an HVZK proof of knowledge.

Easy Extendability with Further Groups: While the two most common groups $(\mathbb{Z}_n, +)$ and $(\mathbb{Z}_m, *)$ are natively supported by our toolbox already, a user can easily add arbitrary self-defined groups. This allows to easily enhance the toolbox, e.g., with groups over elliptic curves that allow high performance and are ideally suited for constraint devices such as embedded systems. To extend the compiler with such a self-defined group, the user would declare an abstract group $(G, +)$ as `Group (G,+);` in the `Declarations` part of the input language. The compiler treats this group called G as an additive group which is also output into the LaTeX documentation. The JAVA back-end automatically generates an abstract class for this group which the user can instantiate with the corresponding implementation of the operations in the intended group.

Future Work. We are currently working on a new version of the compiler which supports efficient proofs in hidden-order groups and automatic transformation of the generated Σ-protocols into non-interactive zero-knowledge proofs (NIZK).

References

1. Goldreich, O., Micali, S., Wigderson, A.: 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 FOCS 1986
2. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: STOC 2007, pp. 21–30. ACM Press, New York (2007)
3. Kilian, J.: A note on efficient zero-knowledge proofs and arguments (extended abstract). In: STOC 1992, pp. 723–732. ACM Press, New York (1992)
4. Schnorr, C.: Efficient signature generation by smart cards. Journal Of Cryptology 4(3), 161–174 (1991)
5. 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)

6. Camenisch, J.: Group Signature Schemes and Payment Systems Based on the Discrete Logarithm Problem. PhD thesis, ETH Zurich, Konstanz (1998)
7. Adelsbach, A., Rohe, M., Sadeghi, A.-R.: Complementing zero-knowledge watermark detection: Proving properties of embedded information without revealing it. Multimedia Systems 11, 143–158 (2005)
8. Lindell, Y., Pinkas, B., Smart, N.: 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: 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: CCS 2002, pp. 21–30. ACM Press, New York (2002)
11. Camenisch, J., Stadler, M.: Efficient group signature schemes for large groups. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 410–424. Springer, Heidelberg (1997)
12. 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)
13. 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)
14. Chan, A., Frankel, Y., Tsiounis, Y.: Easy come - easy go divisible cash. Technical Report TR-0371-05-98-582, GTE (1998), updated version with corrections
15. Okamoto, T.: An efficient divisible electronic cash scheme. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 438–451. Springer, Heidelberg (1995)
16. Camenisch, J., Lysyanskaya, A.: Signature schemes and anonymous credentials from bilinear maps. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 56–72. Springer, Heidelberg (2004)
17. 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)
18. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. CACM 21(2), 120–126 (1978)
19. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999)
20. Damgård, I., Jurik, M.: A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In: Kim, K.-c. (ed.) PKC 2001. LNCS, vol. 1992, pp. 119–136. Springer, Heidelberg (2001)
21. Cramer, R.: Modular Design of Secure yet Practical Cryptographic Protocols. PhD thesis, CWI and University of Amsterdam (1996)
22. Briner, T.: Compiler for zero-knowledge proof-of-knowledge protocols. Master's thesis, ETH Zurich (2004)
23. Camenisch, J., Rohe, M., Sadeghi, A.-R.: Sokrates - a compiler framework for zero-knowledge protocols. In: WEWoRC 2005 (2005)
24. Bangerter, E., Camenisch, J., Krenn, S., Sadeghi, A.R., Schneider, T.: Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471 (2008); Poster session of EUROCRYPT (2009)
25. 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)

26. 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)
27. Camenisch, J., Stadler, M.: Proof systems for general statements about discrete logarithms. Technical Report 260, Institute for Theoretical Computer Science, ETH Zürich (1997)
28. Camenisch, J., Kiayias, A., Yung, M.: On the portability of generalized Schnorr proofs. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 425–442. Springer, Heidelberg (2010)
29. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay — a secure two-party computation system. In: USENIX Security 2004 (2004)
30. 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)
31. MacKenzie, P., Oprea, A., Reiter, M.K.: Automatic generation of two-party computations. In: ACM CCS 2003, pp. 210–219. ACM, New York (2003)
32. Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: ACM CCS 2008, pp. 257–266. ACM Press, New York (2008)
33. Paus, A., Sadeghi, A.-R., Schneider, T.: Practical secure evaluation of semi-private functions. In: Abdalla, M., Pointcheval, D., Fouque, P.-A., Vergnaud, D. (eds.) ACNS 2009. LNCS, vol. 5536, pp. 89–106. Springer, Heidelberg (2009)
34. Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 52–78. Springer, Heidelberg (2007)
35. Brickell, J., Porter, D.E., Shmatikov, V., Witchel, E.: Privacy-preserving remote diagnostics. In: ACM CCS 2007, pp. 498–507. ACM Press, New York (2007)
36. Barbosa, M., Noad, R., Page, D., Smart, N.: First steps toward a cryptography-aware language and compiler. Cryptology ePrint Archive, Report 2005/160 (2005)
37. Barbosa, M., Page, D.: On the automatic construction of indistinguishable operations. Cryptology ePrint Archive, Report 2005/174 (2005)
38. Barbosa, M., Moss, A., Page, D.: Compiler assisted elliptic curve cryptography. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part II. LNCS, vol. 4804, pp. 1785–1802. Springer, Heidelberg (2007)
39. 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)
40. Shamir, A.: How to share a secret. Communications of ACM 22, 612–613 (1979)
41. Guillou, L., Quisquater, J.: A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In: Günther, C.G. (ed.) EUROCRYPT 1988. LNCS, vol. 330, pp. 123–128. Springer, Heidelberg (1988)
42. Bangerter, E.: Efficient Zero-Knowledge Proofs of Knowledge for Homomorphisms. PhD thesis, Ruhr-University Bochum (2005)
43. Brickell, J., Shmatikov, V.: Privacy-preserving classifier learning. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 128–147. Springer, Heidelberg (2009)
44. Piva, A., Caini, M., Bianchi, T., Orlandi, C., Barni, M.: Enhancing privacy in remote data classification. In: New Approaches for Security, Privacy and Trust in Complex Environments, SEC 2008 (2008)
45. Lipmaa, H.: Verifiable homomorphic oblivious transfer and private equality test. In: Laih, C.-S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 416–433. Springer, Heidelberg (2003)

A Generated Output for Example in Fig. 1

A.1 Protocol Inputs

Homomorphisms defined in Input File

$\phi_0 : \mathbb{Z}_m^* \to \mathbb{Z}_m^*,\ \rho_0 \mapsto {\rho_0}^n$
$\phi_1 : \mathbb{Z}_m^* \to \mathbb{Z}_m^*,\ \rho_1 \mapsto {\rho_1}^n$
$\phi_2 : \mathbb{Z}_n \times \mathbb{Z}_m^* \to \mathbb{Z}_m^*,\ (\mu, \rho_2) \mapsto g^\mu \cdot {\rho_2}^n$
$\phi_3 : \mathbb{Z}_n \times \mathbb{Z}_m^* \to \mathbb{Z}_m^*,\ (\mu, \rho_3) \mapsto g^\mu \cdot {\rho_3}^n$

Homomorphisms used in Protocol

$\phi_0,\ \phi_1,\ \psi_2 = \phi_2 \times \phi_3$

Common Input

$\mathbb{Z}_m^*, \mathbb{Z}_n$
$\mathbb{Z} : c^+, n$
$\mathbb{Z}_m^* : g, x_1, x_2$

Preimage Input

$\mathbb{Z}_n : \mu$
$\mathbb{Z}_m^* : \rho_0, \rho_1, \rho_2, \rho_3$

Access Structure

$\Big((\rho_0)\Big) \vee \Big((\rho_1)\Big) \vee \Big((\mu, \rho_2) \wedge (\mu, \rho_3)\Big)$

Constraints on Preimages

$\mu_{\phi_3} = 1 \cdot \mu_{\phi_2}$

Relation

$\phi_0 : x_1 = {\rho_0}^n$
$\phi_1 : x_1 \cdot g^{-1} = {\rho_1}^n$
$\phi_2 : x_1 = g^\mu \cdot {\rho_2}^n$
$\phi_3 : x_2 = g^\mu \cdot {\rho_3}^n$

A.2 Protocol

```
Round 1, Prover:
if secret $\rho_0$ is known:
        $r_{0,0} \in_R \mathbb{Z}_m^*$
        $t_{0,0} := ({r_{0,0}}^n)$
else:
        $s_{0,0} \in_R \mathbb{Z}_m^*$
        $c_0 \in_R [0, c^+]$
        $t_{0,0} := ({s_{0,0}}^n) \cdot {x_1}^{c_0}$
if secret $\rho_1$ is known:
        $r_{1,0} \in_R \mathbb{Z}_m^*$
        $t_{1,0} := ({r_{1,0}}^n)$
else:
        $s_{1,0} \in_R \mathbb{Z}_m^*$
        $c_1 \in_R [0, c^+]$
        $t_{1,0} := ({s_{1,0}}^n) \cdot (x_1 \cdot g^{-1})^{c_1}$
if secret $(\mu, \rho_2, \mu, \rho_3)$ is known:
        $r_{2,0} \in_R \mathbb{Z}_n$, $r_{2,1} \in_R \mathbb{Z}_m^*$, $r_{2,3} \in_R \mathbb{Z}_m^*$
        $r_{2,2} := r_{2,0} \cdot 1$
        $t_{2,0} := (g^{r_{2,0}} \cdot {r_{2,1}}^n)$
        $t_{2,1} := (g^{r_{2,2}} \cdot {r_{2,3}}^n)$
else:
        $s_{2,0} \in_R \mathbb{Z}_n$, $s_{2,1} \in_R \mathbb{Z}_m^*$, $s_{2,3} \in_R \mathbb{Z}_m^*$, $s_{2,2} := s_{2,0} \cdot 1$
        $c_2 \in_R [0, c^+]$
        $t_{2,0} := (g^{s_{2,0}} \cdot {s_{2,1}}^n) \cdot {x_1}^{c_2}$
        $t_{2,1} := (g^{s_{2,2}} \cdot {s_{2,3}}^n) \cdot {x_2}^{c_2}$
```

$$\xrightarrow{\quad t_{0,0}, t_{1,0}, t_{2,0}, t_{2,1} \quad}$$

```
Round 2, Verifier:
$c \in_R [0, c^+]$
```

$$\xleftarrow{\quad c \quad}$$

```
Round 3, Prover:
$(c_0, c_1, c_2) :=$ complete$(c, \{c_0, c_1, c_2\})$
if secret $\rho_0$ is known:
        $s_{0,0} := r_{0,0} \cdot ((\rho_0)^{-1})^{c_0}$
if secret $\rho_1$ is known:
        $s_{1,0} := r_{1,0} \cdot ((\rho_1)^{-1})^{c_1}$
if secret $(\mu, \rho_2, \mu, \rho_3)$ is known:
        $(s_{2,0}, s_{2,1}) := (r_{2,0}, r_{2,1}) + (-(\mu, \rho_2)) \cdot c_2$
        $(s_{2,2}, s_{2,3}) := (r_{2,2}, r_{2,3}) + (-(\mu, \rho_3)) \cdot c_2$
```

$$\xrightarrow{\quad s_{0,0}, s_{1,0}, s_{2,0}, s_{2,1}, s_{2,2}, s_{2,3}, c_0, c_1, c_2 \quad}$$

```
Round 4, Verifier:
Check whether:
        isConsistent$(c, \{c_0, c_1, c_2\}) \overset{?}{=}$ true
        $s_{2,2} \overset{?}{=} 1 \cdot s_{2,0}$
        $t_{0,0} \overset{?}{=} ({s_{0,0}}^n) \cdot {x_1}^{c_0}$
        $t_{1,0} \overset{?}{=} ({s_{1,0}}^n) \cdot (x_1 \cdot g^{-1})^{c_1}$
        $t_{2,0} \overset{?}{=} (g^{s_{2,0}} \cdot {s_{2,1}}^n) \cdot {x_1}^{c_2}$
        $t_{2,1} \overset{?}{=} (g^{s_{2,2}} \cdot {s_{2,3}}^n) \cdot {x_2}^{c_2}$
```