cPLC – A Cryptographic Programming Language and Compiler

Endre Bangerter, Stephan Krenn
Bern University of Applied Sciences
Engineering and Information Technology
CH-2501 Biel, Switzerland
Email: {endre.bangerter,stephan.krenn}@bfh.ch

Martial Seifriz, Ulrich Ultes-Nitsche
University of Fribourg
Department of Informatics
CH-1700 Fribourg, Switzerland
Email: {martial.seifriz,ulrich.ultes-nitsche}@unifr.ch

***Abstract*—Cryptographic two-party protocols are used ubiquitously in everyday life. While some of these protocols are easy to understand and implement (e.g., key exchange or transmission of encrypted data), many of them are much more complex (e.g., e-banking and e-voting applications, or anonymous authentication and credential systems).**

For a software engineer without appropriate cryptographic skills the implementation of such protocols is often difficult, time consuming and error-prone. For this reason, a number of compilers supporting programmers have been published in recent years. However, they are either designed for very specific cryptographic primitives (e.g., zero-knowledge proofs of knowledge), or they only offer a very low level of abstraction and thus again demand substantial mathematical and cryptographic skills from the programmer. Finally, some of the existing compilers do not produce executable code, but only metacode which has to be instantiated with mathematical libraries, encryption routines, etc. before it can actually be used.

In this paper we present a cryptographically aware compiler which is equally useful to cryptographers who want to benchmark protocols designed on paper, and to programmers who want to implement complex security sensitive protocols without having to understand all subtleties. Our tool offers a high level of abstraction and outputs well-structured and documented Java code. We believe that our compiler can contribute to shortening the development cycles of cryptographic applications and to reducing their error-proneness.

***Index Terms*—Cryptographic compiler; software engineering; security protocols;**

I. Introduction

During the last three decades a high number of devices running software has permeated our society. Even if many of them are invisible to most of the world, the statement that “our civilization runs on software” from Bjarne Stroustrup, inventor of the C++ programming language, is true: besides software running on laptops or desktop computers, it is also used in less obvious ways to control everyday devices such as televisions, airplanes or cash dispensers. Given this ubiquity of software and the difficulty of obtaining sound implementations for a given task, it is not surprising that implementation errors cause costs in the amount of billions of dollars every year [2].

This work was in part funded by the European Community’s Seventh Framework Programme (FP7) under grant agreement no. 216499 and the Swiss Hasler Foundation under projects no. 09037 and 10069. For a full version of this document we refer to [1].

Things get even worse when aiming for realizing security sensitive tasks such as e-banking or e-voting, or transferring sensitive data via e-mail or instant messaging over the Internet, where even small errors can lead to serious security flaws revealing private data. Prominent examples include, e.g., the widespread OpenSSL library [3], where several potential threats have been discovered [4], [5], or GPG [6], an open-source implementation of the OpenPGP standard, where misoptimizations lead to severe security flaws [7]. One main reason why such problems can emerge is the skill gap between cryptographers and programmers. While the former do not always have sufficient programming skills to obtain efficient implementations of protocols defined on paper, the latter often do not have enough mathematical and cryptographic knowledge to understand all subtleties of complex protocols. This makes the implementation of cryptographic applications even more time-consuming and error-prone than that of other systems. Furthermore, because of this skill gap many potentially interesting cryptographic applications found in the literature are left unimplemented.

During the last years several attempts to overcoming these problems have been made, including well-funded projects financed by the EU- or the US governments [8], [9]. One strategy all these projects have in common is to automatize the implementation process of cryptographic schemes, and to offer extensive compiler support to programmers.

A. Our Contribution

In this paper we present cPLC, a cryptographic Programming Language and Compiler, for generating implementations of two-party cryptographic protocols. The main features of our compiler can be summarized as follows:

- The input language of our compiler is strongly inspired by the standard notation for specifying protocols used in the cryptographic community. This allows a user to translate abstract specifications to input files of our compiler in a straightforward manner.
- Our compiler natively supports most groups used in applied cryptography, including residue groups and groups over elliptic curves. This is in contrast to most previous compilers, where (similar to C), e.g., all modular operations have to be implemented explicitly, which results

in illegible terms for complex expressions. Also, features such as automatic type inference allow one to introduce variables on the fly without any needs to declare them in advance. This is again in complete analogy to the standard way of defining protocols by hand.

- Furthermore, our compiler supports standard procedures (e.g., AES, SHA2), mathematical algorithms (e.g., primality tests, random number generation) as well as complex cryptographic primitives (e.g., zero-knowledge proofs of knowledge) as basic building blocks, which takes away the need for corresponding skills from the software engineer.
- Our compiler comes along with a graphical user interface (GUI), which supports the user in compiling input files, setting up communication details, etc., and which also allows for efficiently benchmarking the specified protocol.
- The output of our compiler is fully functional Java code, which can directly be compiled to binaries.

Overall, we believe that our compiler is the first such tool which can be used by cryptographically untrained software engineers to obtain sound implementations of arbitrary two-party protocols, as well as by cryptographers who want to efficiently implement their protocols designed on paper.

B. Related Work

A number of tools for specific cryptographic primitives exist. For instance, a first prototype of a compiler for zero-knowledge proofs of knowledge was started by Briner [10] and enhanced by Bangerter et al. [11]. Subsequently, the tool was redeveloped and extended [12], [13]. Independently from and parallel to this compiler, Meiklejohn et al. [14] also developed a compiler for zero-knowledge proofs. Both tools offer a high-level front end for specifying the intended protocol, but are not general enough to specify arbitrary cryptographic protocols. Our compiler uses that of Almeida et al. [13] as a subroutine for realizing zero-knowledge proofs.

In the field of secure two-party computation [15] a number of tools are available, e.g., Fairplay [16], VIFF [17] and TASTY [18]. The generated protocols in either of these tools are based on Yao's garbled circuits [19] or on homomorphic encryption, e.g., [20]. However, protocols solely based on these techniques are often too inefficient for practical use. Furthermore, none of these compilers offers an input language which is abstract enough to be used by non-experts.

Other generic cryptographic compilers have been suggested by Lucks et al. [21], Kiyomoto et al. [22], Lewis and Martin [23] and Schröpfer et al. [24]. The input language of some of these tools [22], [23], [24] is very low-level and does not offer the level of abstraction required by cryptographically untrained software engineers. Lucks et al. [21] offer a high level of abstraction, but their compiler does not produce complete code, but only meta code in which first all cryptographic primitives have to be instantiated before being compiled. Moran provides a comprehensive Java library for rapid prototyping of cryptographic software [25]. Our compiler combines an abstract input language and fully functional output.

Finally, a number of compilers which together with the required implementations output formal correctness certificates of the generated code exist [13], [23], [26].

An extensive motivation for cryptographically aware compilers was given by Bangerter et al. [27].

C. Roadmap

In Section II we describe the architecture of our compiler and the graphical user interface. Then, in Section III we describe the rationals underlying the input language, and we give some illustrative sample programs in Section IV. Finally, we briefly conclude in Section V.

II. Architecture and Description

Before describing the input language and features of cPLC we describe the high-level architecture of the compiler and its graphical user interface.

A. Architecture

The rough structure of cPLC is straightforward, and is depicted in Figure 1.

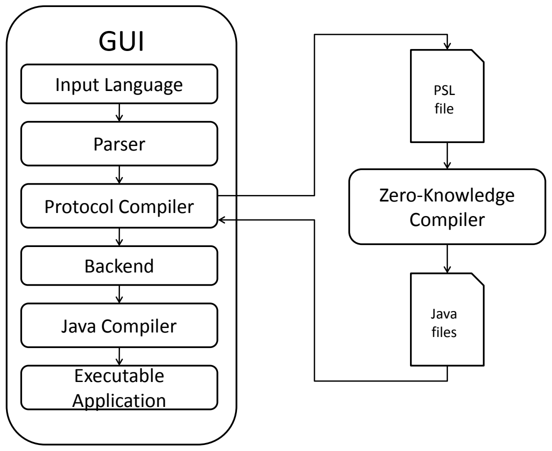

Fig. 1. High-level architecture of cPLC.

The compiler takes as input the specification of a cryptographic protocol, which is parsed using a parser generated by the ANTLR Parser Generator [28], [29]. The resulting internal representation of the protocol is then treated by the main component, the *protocol compiler*, which translates the high-level description of a protocol into a sequence of computations and commands.

The resulting implementation specification is passed to a backend. Currently, only a Java backend is available. However, the modular design of the compiler allows for an easy incorporation of backends for other target languages as well. The files generated for each party are then compiled into byte code using a standard Java compiler.

To achieve a high level of abstraction, cPLC also offers zero-knowledge proofs of knowledge as a basic cryptographic primitive. Various compilers for this primitive have already been proposed in the literature and we use that of Almeida et al. [13] as a sort of plug-in. That is, whenever cPLC encounters a specification of such a proof, it produces an interim output which serves as an input to the zero-knowledge compiler (a so-called *PSL-file*), which in turn outputs a Java implementation of the proof. This code is then integrated into the Java output of cPLC to obtain one single implementation of the overall protocol. Not implementing a zero-knowledge compiler ourselves was one major design decision resulting from the wish to avoid extensive re-implementations of functionalities already existing in other tools.

Except for the graphical user interface, all components of our compiler have been realized in Python 2.5. However, cPLC can be used without having a Python interpreter installed using Jython [30] which is a pure Java implementation of the Python language specification and supports most language features from Python 2.5, especially those needed by our compiler. It compiles Python code to Java byte code which can run directly on a Java virtual machine. The GUI is realized in Java and built with Netbeans Swing GUI Builder [31] and the Swing Application Framework.

B. Graphical User Interface

cPLC could, in principle, be used as a command-line tool, but for improved user friendliness we also implemented a rather self-explanatory GUI. The major advantage of the GUI is giving all of cPLC's functionality the same look and feel. It can be used to specify and compile protocols and to edit configuration and input files. The target audience of cPLC are software engineers who normally are used to integrated development environments. We believe that having all the functionality for cryptographic protocol design and implementation integrated into one GUI facilitates the uptake of cPLC in practice.

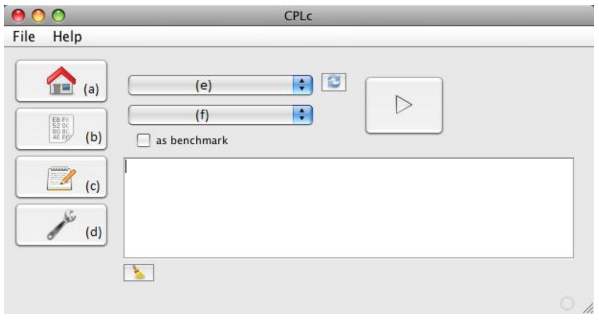

Fig. 2. Main panel of the graphical user interface of cPLC.

When starting cPLC, the GUI shown in Figure 2 opens. This is the main panel of the GUI that can always be re-opened by clicking *(a)*. All protocols which have already been implemented earlier can be chosen from the upper drop-down box *(e)*, whereas the lower one *(f)* allows one to choose the party whose code one wishes to execute. The implementation can then be started by clicking the play button.

When the implementations of all participants of a protocol are running, the protocol run starts automatically. In fact, all implementations of protocol participants always execute immediately. However, they only execute as far as possible, i.e., until they require inputs from the other party to continue. This synchronization happens via send/receive events implemented as sockets.

Input data to the protocol run can either be hard-coded in the input file, which, however, is very inflexible if the protocol is to be used in practice. Therefore, it can also be passed to the protocol via a separate input file. This input file can be edited in the text box of the GUI by selecting the input panel *(b)*. It essentially consists of unformatted text where each line contains the value of a single variable. That is, to define an input n one may just add a line

```
n=1234567890
```

to this text file. The order of the inputs is irrelevant. As some protocols show different behavior for users who know a certain variable, and for those who do not, unknown inputs can be indicated by

```
n=Unknown .
```

If one wants to specify a new protocol, i.e., a protocol which was not available in the drop-down box *(e)* on startup, or to edit an existing one, one can use the editor panel *(c)*. The syntax and rationals of the input language will be discussed in detail in the following sections.

By default, all protocol participants run on IP address 127.0.0.1 (*localhost*). This is useful for first test runs of the implemented protocol, but not realistic for real-world scenarios where both participants will run on different machines with different IP addresses. These can be specified in an XML configuration file which can be edited in the configuration panel *(d)*. Besides the IP address, also the communication port, and logging options for debugging and benchmarking can be specified there.

1) Integrated Benchmarking Mechanism

Typically there are numerous protocols which can be used for a specific purpose, the efficiency of which might vary significantly. Thus, cPLC also allows protocol designers to easily obtain benchmarks for their protocols. To do so, they just have to tick the respective box in the GUI. They are then asked to enter the number of repetitions of the protocol to be benchmarked.

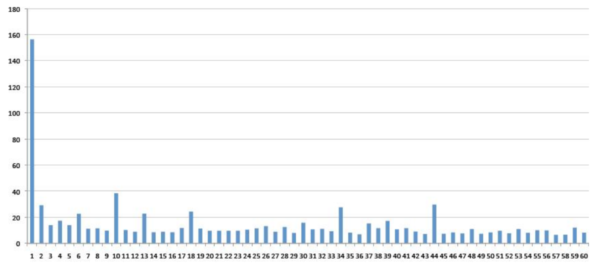

Fig. 3. Typical benchmarking results.

Figure 3 shows the results of a typical benchmarking procedure for the Diffie-Hellman key exchange protocol [32]. As can be seen, the first run of the protocol requires much more time than all subsequent runs. This is due to the way Java loads classes: namely, mathematical or network communication classes are only loaded when they are first used, and not on startup. Thus, the duration of the first protocol run always contains an almost constant overhead for this class loading. However, when a cryptographic protocol is used as a building block of a larger applications, most classes will already be available when the protocol is started. Thus, when comparing the efficiency of different protocols, one might most often want to ignore this first (less efficient) protocol run.

2) System Requirements and Installation

Our tool suite only requires a Java runtime environment and a Java compiler version 6 or higher which are freely available on the Internet and already installed on most machines. To make the compiler as easy to use as possible it does not require any installation but can just be started by double-clicking the main `jar`-archive.

III. Input Language and Features

In the following section we briefly describe the rationals underlying the input language of our compiler on a high level, and the core functionalities which are natively supported. Concrete sample programs will then be given in Section IV, and a formal specification can be found in Appendix A.

a) Operators: All standard operators on variables are available in our input language. In particular this includes standard arithmetical operators (e.g., `a%m` for $a \bmod m$), bitwise operations (e.g., `bitand`), and concatination (`concat`).

b) Conditionals and Loops: As in other programming languages, `if`/`else` branches are available. Also, `for` and `while` loops are available, where the former can be used to iterate over numeric ranges as well as arrays.

c) Data types: Our compiler distinguishes three types of integers: besides basic signed integers (`Int`), it also allows one to specify elements lying within some interval $[a, b]$ (`Interval(a,b)`), or primes of a given bitlength k (`Prime(k)`). Furthermore, as cryptography makes extensive usage of finite groups and fields, the most commonly used groups are natively supported as well. That is, a command like `ZmodMul(p) g` defines an element $g \in \mathbb{Z}_p^*$, and similar for `ZmodAdd(q) x`. Also, because of their practical relevance, Galois fields with 2^m elements can be specified by `GF2(m)`.

For elliptic curves, different implementations are available. On the one hand, a generic implementation allows one to specify an arbitrary elliptic curve over $\mathbb{Z}_p^*$ in Weierstraß form (i.e., $y^2 = x^3 + ax + b$) as `EC(p,a,b,gx,gy)`, where (g_x, g_y) is a generator of the group required for drawing random elements. Alternatively, those curves suggested by Brown [33] are already predefined and can be called by their name, e.g., by `EC(secp112r1)`.

If programmers want to use other groups they can define abstract groups in cPLC, and only have to offer implementations of the group operations in Java at compile time.

Upon reading values from the input file or receiving data from the other party, group membership is always checked. This is consistent to the respective assumption made in most protocol specifications.

d) Basic Built-In Primitives: The input language of our compiler natively supports symmetric encryption using the Advanced Encryption Standard (AES) [34], [35], which can be invoked by `enc(message,key)`, and similar for `dec` for decryption. Furthermore, the Secure Hashing Standard (SHA-2) [36] can be invoked by `hash(message)`. An optional second integer parameter allows to specify whether the output should be truncated to a specific length (default is 512 bits).

e) Zero-Knowledge Proofs of Knowledge: On a high level, a zero-knowledge proof of knowledge is a two-party protocol between a prover and a verifier, which allows the prover to convince the verifier that it knows some secret values that satisfy a given relation (proof of knowledge property), without the verifier being able to learn anything about them (zero-knowledge property) [37]. Such proofs are extensively used in applied cryptography as they allow protocol designers to enforce a protocol participant to assure other parties that its actions are consistent with its internal knowledge state, e.g., [38], [39], [40], [41], [42].

We therefore also incorporated zero-knowledge proofs of knowledge as a basic primitive into our compiler, using the compiler of Almeida et al. [13] as a subroutine. The syntax for such proofs is very much inspired by the standard notation introduced by Camenisch and Stadler [43]. For instance, to prove knowledge of $x = \log_g y$ one can simply write `zkp("(x):y=g^x")`.

f) Benchmarking: As mentioned in Section II-B1 the GUI offers a checkbox whether or not the overall protocol should be benchmarked. However, the input language also offers the feature of benchmarking arbitrary blocks of a protocol. To this end, the block just has to be encapsulated between `startbm(name)` and `stopbm(name)` for an arbitrary string `name`. The running time of this block will then be output upon execution of the protocol. The number of blocks to be benchmarked is not limited by our compiler.

1) Turing Completeness

The main goal when designing the input language of cPLC was to support the user in describing the single rounds of a protocol as efficiently as possible by natively offering the most commonly used primitives, etc. However, we also aimed for generality, i.e., the user should be able to describe any computable function in each round, or put differently, the input language should be Turing complete. This can easily be seen to be satisfied, as all necessary language elements (input via the input file, output, loops, branches, etc.) are available, and neither the number nor the size of variables is limited.

IV. Sample Programs

After having described the comprehensiveness of the input language of cPLC in the previous section, we now give some sample programs illustrating how it can be used to describe concrete protocols.

Generally, except for `if`/`else` blocks and `for` loops, which have to be ended by `end`, line breaks are used as statement delimiters. All other whitespace is ignored.

A. A Cryptographic "Hello World"-Program

We start with a simple and well-known protocol, namely the Diffie and Hellmann key-exchange protocol [32]. The standard way to specify the protocol is illustrated in Figure 4, and the implementation in cPLC is given by Algorithm 1.

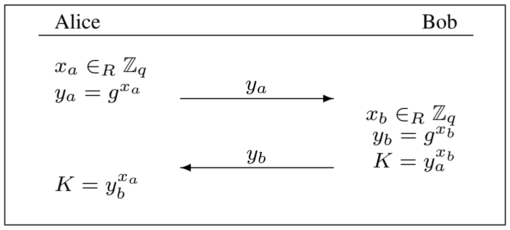

Fig. 4. Protocol flow of the Diffie-Hellmann key exchange as specified in most cryptographic publications.

Algorithm 1 Diffie-Hellmann Key Exchange

```
    Input:
2:    All:
        Prime p, q
4:      ZmodMul(p) g
      Alice:
6:    Bob:
    Protocol:
8:    Alice:
        x_a = random(ZmodAdd(q))
10:     y_a = g^x_a
        send(y_a)
12:   Bob:
        x_b = random(ZmodAdd(q))
14:     y_b = g^x_b
        K = y_a^x_b
16:     send(y_b)
      Alice:
18:     K = y_b^x_a
```

Let us now discuss the single parts of the implementation in more detail. As can be seen, the input file consists of two blocks. In the `Input` block (lines 1 to 6) all values given as inputs to the protocol participants have to be specified. In the `Protocol` block (lines 7 to 18) the actual protocol is specified.

a) The `Input` *Block:* At first, the common inputs to both parties are specified (lines 2 to 4). In this case, two primes p, q and $g \in \mathbb{Z}_p^*$ are given as inputs. Remember that in the Diffie-Hellmann protocol, g is a generator of the subgroup of order q or $\mathbb{Z}_p^*$.

Next, private inputs to the parties can be specified (lines 5 and 6), which do not exist in our example. However, the two lines must not be suppressed as they implicitly also define the names of the parties in the protocol.

b) The `Protocol` *Block:* This block is a straightforward translation from Figure 4. Note that none of the variables in the whole protocol needs to be declared. For instance, in line 9 the type of `x_a` is automatically inferred to be `ZmodAdd(q)`. Similarly, in line 10, the type of `y_a` must be `ZmodMul(p)`. Also, again in analogy to Figure 4, variables sent from one party to the other can directly be referred to by the same name. For instance, this is the case in line 14, where `y_a` was previously sent from Alice to Bob in line 11. This avoids the necessity of an explicit `receive` command, and helps to keep the protocol specification as compact as possible.

B. More Language Elements

While the Diffie-Hellmann protocol illustrates the basic functionality of our input language quite well, some concepts are not used there. We show their usage by some code snippets in the following.

Algorithm 2 More language elements

```
    Definition:
2:    MyGroup(5)
    Input:
4:    All:
        ⋮
6:      MyGroup(a,b,c,d,e) g, h
        ⋮
8:  Protocol:
      ⋮
10:   Alice:
        pw = hash("A shared secret",128)
12:     cipher = enc("Some secret message",pw)
        send(cipher)
14:   Bob:
        plaintext = dec(cipher,hash("A shared secret",128))
16:   ⋮
```

As mentioned in Section III, the user of our compiler has the possibility to define not built-in groups at the onset of the input file. This is shown in Algorithm 2: the optional `Definition` block (lines 1 and 2) to declare this groups. In our case, a group type called `MyGroup` taking 5 constructor arguments is defined, and can be used like built-in groups in the following. For instance, line 6 defined elements g, h in `MyGroup` parametrized by `a,b,c,d,e`. Furthermore, the example shows the usage of the built-in hashing and encryption primitives. In line 11, `pw` is assigned the first 128 bits of the hash of some secret shared between Alice and Bob, which is then used in line 12 to encrypt some secret message using AES with the 128 bit key `pw`. Bob decrypts the message analogously.

Algorithm 3 illustrates the usage of loops and conditional branches on the basis of the Fiat-Shamir identification protocol [44], which allows Alice to prove knowledge of the square

Algorithm 3 Fiat Shamir Identification

```
     Input:
 2:    All:
         n = 553913
 4:      ZmodMul(n) v = 295502
       Alice:
 6:      ZmodMul(n) s = 43215
       Bob:
 8:  Protocol:
       Alice:
10:      ZmodMul(n)[100] r, x
         for i in Interval(0,99):
12:        r[i] = random(ZmodMul(n))
           x[i] = r[i]^2
14:      end
         send(x)
16:    Bob:
         Int[100] e
18:      for e_i in e:
           e_i = random(Interval(0,1))
20:      end
         send(e)
22:    Alice:
         ZmodMul(n)[100] y
24:      for i in Interval(0,99):
           y[i] = r[i]*s^e[i]
26:      end
         send(y)
28:    Bob:
         i = 0
30:      while i < 100:
           if !(y[i]^2 == x[i]*v^e[i]):
32:          stop("Verification failed")
           end
34:        i = i+1
         end
```

root of `v` to Bob. It is possible to define default values of inputs (line 3), which are overwritten if inputs for the respective variables are specified in the input file.

In line 10 two arrays of 100 elements of $\mathbb{Z}_n^*$ are defined. In contrast to variables, arrays must not be introduced on the fly to avoid, e.g., different array sizes in different conditional branches. Then, a `for` loop (lines 11 to 14) is used to assign values to the entries of the arrays. In cPLC, array indices always start at 0. Alternatively, the loop could also iterate over the entries of an array, see line 18. Lines 30 to 35 show the usage of `while` loops. Lines 31 to 33 show a simple conditional construct. Upon reaching a `stop` command, the protocol prints the included message and terminates. Alternatively, the `if` statement could have been written as `assert(y[i]^2 == x[i]*v^e[i])`.

We stress that using the built-in zero-knowledge primitive, the whole protocol could just have been written as `zkp((s):v=s^2)`.

C. Two Real-World Protocols

We finish this section by giving two real-world protocols.

Algorithm 4 Coin Withdrawal Protocol

```
     Input:
 2:    All:
         Prime(1024) p, q
 4:      ZmodMul(p) I, g, h, g_1, g_2, d
       Client:
 6:      ZmodAdd(q) u_1, u_2
       Bank:
 8:      ZmodAdd(q) x
     Protocol:
10:    Client:
         zkp("(u_1, u_2) : I=g_1^u_1 * g_2^u_2")
12:    Bank:
         w = random(ZmodAdd(q))
14:      m = I*d
         z = m^x
16:      a = g^w
         b = m^w
18:      send(z,a,b)
       Client:
20:      m = g_1^u_1 * g_2^u_2 * d
         s = random(ZmodAdd(q))
22:      mp = m^s
         zp = z^s
24:      x_1 = random(ZmodAdd(q))
         x_2 = u_1 * s - x_1
26:      y_1 = random(ZmodAdd(q))
         y_2 = u_2 * s - y_1
28:      z_1 = random(ZmodAdd(q))
         z_2 = s - z_1
30:      A = g_1^x_1 * g_2^x_2 * d^z_1
         B = m^s / A
32:      u = random(ZmodAdd(q))
         v = random(ZmodAdd(q))
34:      ap = a^u * g^v
         bp = b^(s*u) * mp^v
36:      ZmodMul(q) cp = hash(concat(mp,zp,ap,bp,A))
         c = cp / u
38:      send(c)
       Bank:
40:      r = x*c + w
         send(r)
42:    Client:
         assert(g^r == h^c * a and m^r == z^c * b)
44:      rp = r*u + v
```

The first one, i.e., Algorithm 4, stems from an e-cash scheme proposed by Brands [45]. It is used by clients to withdraw money from their bank accounts. The protocol starts with a zero-knowledge proof where a client shows that he indeed has access to the specified bank account in line 11. The remainder of the protocol is the withdrawal of a coin itself. We do not describe the semantics of the protocol in more detail here, but refer to the original paper [45]. However, it can be seen that even for complex protocols, reading the specification in our input language is no more difficult than reading its specification in a diagram.

In line 36 an explicit typecast has to be performed, as the type inference strategy of the compiler would define cp to be an element of $\mathbb{Z}_q$, and not of $\mathbb{Z}_q^*$.

Algorithm 5 Entity certification

```
Input:
  All:
    Int lambda, n
    ZmodMul(n) a_b, g_b
  Alice:
  CredentialAuthority:
    Int d_b
Protocol:
  Alice:
    xprime = random(Interval(0,2^lambda - 1))
    yprime = a_b^xprime
    send(yprime)
  CredentialAuthority:
    xi = random(Interval(0,2^lambda - 1))
    send(xi)
  Alice:
    x = xprime + xi
    y = a_b^x
    z = g_b^y
    send(y,z)
    zkp("(y) : z = g_b^y")
  CredentialAuthority:
    assert(y == yprime * a_b^xi)
    cert = (y+1)^d_b
    send(cert)
```

Our final example, see Algorithm 5, is one of the protocols from an history-based signature scheme by Bussard et al. [46]. Such schemes can be used to rate the trustworthiness of a document the author of which wants to stay anonymous. Upon signing a document, the author has the possibility to include some information about his history into the signature (e.g., he was present at some event). The credentials certificating these facts are issued by some certificate authority. Algorithm 5 shows the initialization protocol, where a new user Alice sets up her secret value x and receives a certificate cert that the setup has been performed correctly.

V. Conclusion and Future Work

We presented a domain-specific language and compiler for arbitrary cryptographic two-party protocols and demonstrated their usability on the basis of some real-world protocols. The tool is easy to use and its input language offers a higher level of abstraction than those of previous tools.

In our opinion there are at least two points for further development. First, a generalization to more than two parties would be useful, and could be reached within a reasonable amount of work. The main task would be to incorporate a receiver into the send() command, and to adapt the network implementations accordingly. Second, currently the compiler does not support user-defined functions. However, this could further increase the readability of protocol specifications as repeatedly used code could be transferred into such a function.

References

[1] M. Seifriz, "CPLc: A Generic Compiler for Fast Prototyping Cryptographic Protocols," Master's thesis, University of Fribourg, 2010.

[2] M. Newman, "Software Errors Cost U.S. Economy $59.5 Billion Annually," NIST, Tech. Rep., June 2002.

[3] OpenSSL, "OpenSSL: The Open Source toolkit for SSL/TLS," http://www.openssl.org/, 1998–2011.

[4] ——, "OpenSSL FIPS Object Module Vulnerabilities," http://www.openssl.org/news/secadv_20071129.txt, 2007.

[5] ——, "Incorrect checks for malformed signatures," http://www.openssl.org/news/secadv_20090107.txt, 2009.

[6] GnuPG, "The GNU Privacy Guard," http://www.gnupg.org/, 2002–2011.

[7] P. Q. Nguyen, "Can We Trust Cryptographic Software? Cryptographic Flaws in GNU Privacy Guard v1.2.3," in *EUROCRYPT 04*, ser. LNCS, C. Cachin and J. Camenisch, Eds., vol. 3027. Springer, 2004, pp. 555–570.

[8] CACE, "Computer Aided Cryptography Engineering ," http://www.cace-project.eu/.

[9] PROCEED, "PROgramming Computation on EncryptEd Data," https://www.fbo.gov/?s=opportunity&mode=form&id=9358278f35a1c1ea713dca2c9a86a05e.

[10] T. Briner, "Compiler for Zero-Knowledge Proof-of-Knowledge Protocols," Master's thesis, ETH Zurich, 2004.

[11] E. Bangerter, T. Briner, W. Heneka, S. Krenn, A.-R. Sadeghi, and T. Schneider, "Automatic Generation of Σ-Protocols," in *EuroPKI 09*, ser. LNCS, F. Martinelli and B. Preneel, Eds., vol. 6391. Springer, 2009, pp. 67–82.

[12] E. Bangerter, S. Barzan, S. Krenn, A.-R. Sadeghi, T. Schneider, and J.-K. Tsay, "Bringing zero-knowledge proofs of knowledge to practice," in 17*th International Workshop on Security Protocols – SPW 2009*, ser. LNCS. Springer, (to appear).

[13] J. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider, "A Certifying Compiler for Zero-Knowledge Proofs of Knowledge Based on Σ-Protocols," in *ESORICS 10*, ser. LNCS, D. Gritzalis, B. Preneel, and M. Theoharidou, Eds., vol. 6345. Springer, 2010, pp. 151–167.

[14] S. Meiklejohn, C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya, "ZKPDL: A Language-Based System for Efficient Zero-Knowledge Proofs and Electronic Cash," in *USENIX Security Symposium 10*. USENIX Association, 2010, pp. 193–206.

[15] A. Yao, "Protocols for Secure Computations (Extended Abstract)," in *FOCS 82*. IEEE Computer Society, 1982, pp. 160–164.

[16] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay – Secure two-party computation system," in *USENIX Security Symposium*. USENIX Association, 2004, pp. 287–302.

[17] I. Damgård, M. Geisler, M. Krøigaard, and J. Nielsen, "Asynchronous Multiparty Computation: Theory and Implementation," in *PKC 09*, ser. LNCS, S. Jarecki and G. Tsudik, Eds., vol. 5443. Springer, 2009, pp. 160–179.

[18] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: Tool for Automating Secure Two-partY computations," in *ACM CCS 10*, E. Al-Shaer and A. D. K. nad V. Shmatikov, Eds. ACM Press, 2010.

[19] A. Yao, "How to Generate and Exchange Secrets (Extended Abstract)," in *FOCS 86*. IEEE Computer Society, 1986, pp. 162–167.

[20] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in *EUROCRYPT 99*, ser. LNCS, J. Stern, Ed., vol. 1592. Springer, 1999, pp. 223–238.

[21] S. Lucks, N. Schmoigl, and E. I. Tatlı, "Issues on Designing a Cryptographic Compiler," in *WEWoRC 05*, ser. LNI, C. Wolf, S. Lucks, and P.-W. Yau, Eds., vol. 74. Gesellschaft für Informatik e.V., 2005, pp. 109–122.

[22] S. Kiyomoto, H. Ota, and T. Tanaka, “A Security Protocol Compiler Generating C Source Codes,” in *ISA 08*. IEEE Computer Society, 2008, pp. 20–25.

[23] J. R. Lewis and B. Martin, “Cryptol: high assurance, retargetable crypto development and validation,” in *Conference on Military Communications - Volume II*, ser. MILCOM 03. IEEE Computer Society, 2003, pp. 820–825.

[24] A. Schröpfer, F. Kerschbaum, D. Biswas, S. Geißinger, and C. Schütz, “L1 - Faster Development and Benchmarking of Cryptographic Protocols,” in *SPEED-CC 09*, 2009.

[25] T. Moran, “The Qilin Crypto SDK: An Open-Source Java SDK for rapid prototyping of cryptographic protocols,” http://qilin.seas.harvard.edu/.

[26] X. Didelot, “COSP-J: A Compiler for Security Protocols,” Master’s thesis, University of Oxford, 2003.

[27] E. Bangerter, M. Barbosa, D. J. Bernstein, I. Damgård, D. Page, J. I. Pagter, A.-R. Sadeghi, and S. Sovio, “Using Compilers to Enhance Cryptographic Product Development,” in *ISSE 09*, N. Pohlmann, H. Reimer, and W. Schneider, Eds. Vieweg + Teubner, 2009, pp. 291–301.

[28] ANTLR, “ANTLR parser generator,” http://www.antlr.org/.

[29] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007.

[30] Jython, “The Jython project,” http://www.jython.org/.

[31] Netbeans, “Netbeans,” http://www.netbeans.orf/.

[32] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. IT-22, no. 6, pp. 644–654, 1976.

[33] D. Brown, *Standards for Efficient Cryptography. SEC 2: Recommended Elliptic Curve Domain Parameters*. Certicom Research, 2010.

[34] J. Daemen and V. Rijmen, “AES proposal: Rijndael,” AES Algorithm Submission, 1999.

[35] FIPS, *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, 2001, `http://csrc.nist.gov/publications/fips/`. Federal Information Processing Standard 197.

[36] ——, *Secure Hash Standard (SHS)*. National Institute of Standards and Technology, 2008, `http://csrc.nist.gov/publications/fips/`. Federal Information Processing Standard 180-3.

[37] M. Bellare and O. Goldreich, “On Defining Proofs of Knowledge,” in *CRYPTO 92*, ser. LNCS, E. F. Brickell, Ed., vol. 740. Springer, 1993, pp. 390–420.

[38] S. Brands, “Untraceable off-line cash in wallet with observers,” 1994, pp. 302–318.

[39] H. Kikuchi, K. Nagai, W. Ogata, and M. Nishigaki, “Privacy-preserving similarity evaluation and application to remote biometrics authentication,” *Soft Computing*, vol. 14, no. 5, pp. 529–536, 2010.

[40] F. Le and S. M. Faccin, “IPv6 address ownership solution based on zero-knowledge identification protocols or based on one time password,” U.S. Patent #7,546,456 issued Jun. 9, 2009, filed 2003.

[41] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin, “Using the Secure Remote Password (SRP) Protocol for TLS Authentication,” Request for Comments: 5054, 2007.

[42] T. Wu, “The SRP Authentication and Key Exchange System,” Request for Comments: 2945, 2000.

[43] J. Camenisch and M. Stadler, “Efficient Group Signature Schemes for Large Groups (Extended Abstract),” in *CRYPTO 97*, ser. LNCS, B. Kaliski, Ed., vol. 1294. Springer, 1997, pp. 410–424.

[44] A. Fiat and A. Shamir, “How to prove yourself: practical solutions to identification and signature problems,” in *CRYPTO 86*, ser. LNCS, A. M. Odlyzko, Ed., vol. 263. Springer, 1987, pp. 186–194.

[45] S. Brands, “An efficient off-line electronic cash system based on the representation problem,” CWI, Tech. Rep. CS-R9323, 1993.

[46] L. Bussard, R. Molva, and Y. Roudier, “History-Based Signature or How to Trust Anonymous Documents,” in *iTrust 2004*, ser. LNCS, C. Jensen, S. Poslad, and T. Dimitrakos, Eds., vol. 2995. Springer, 2004, pp. 78–92.

Appendix

A. ENBF Specification of Input Language

In the following we give the EBNF specification of the input language of cPLC. Note that calls to, e.g., AES, SHA-2, assertions or printing commands are all treated as `functionCalls`, and interpreted at a higher level in the compilation process. This allows for adding further built-in functions without having to change the grammar.

cpl: [definition] input protocol;
definition: ’Definition:’ { userGroupDef | userGroupAlias};
input: ’Input:’ {party blockBegin {varDeclaration eos}};
protocol: ’Protocol’ blockBegin {party ’:’ {statement}};
userGroupDef: type eos;
userGroupAlias: variable ’=’ type eos;
type: Identifier ’(’ expr {’,’ expr} ’)’;
list: ’[’ expr {’,’ expr} ’]’;
party: Identifier;
assignment: [type] variable ’=’ expr | arrayDeclaration ’=’ expr | arrayAccess ’=’ expr;
varDeclaration: [type] variable {’,’ variable};
arrayDeclaration: type ’[’ expr ’]’ variable;
arrayAccess: variable ’[’ expr ’]’;
variable: Identifier;
statement: (assignment | arrayDeclaration | varDeclaration | functionCall | zkpok) eos | controlStructure ;
controlStructure: if | for | while;
if: ’if’ expr blockBegin {statement} (’end’ eos elseIfControl | ’end’ eos elseControl | ’end’ eos);
elseIfControl: ’elseIf’ expr blockBegin {statement} (’end’ eos elseIfControl | ’end’ eos elseControl | ’end’ eos);
elseControl: ’else’ blockBegin {statement} ’end’ eos;
forControl: ’for’ variable ’in’ expr blockBegin {statement} ’end’ eos;
whileControl: ’while’ expr blockBegin {statement} ’end’ eos;
blockBegin: ’:’ [eos];
functionCall: Identifier ’(’ ’)’ | Identifier ’(’ expr {’,’ expr} ’)’;
zkpok: ’zkp’ ’(’ ’”’ ’(’ variable {’,’ variable} ’)’ ’:’ proofGoal ’”’ [’,’ identifier] ’)’;
proofGoal: predicate | proofGoal { (’AND’ | ’OR’) proofGoal } | ’(’ proofGoal ’)’
predicate: math ’=’ math;
expr: ’not’ expr | expr (’and’ | ’or’) expr | comparison;
bitXor: ’bitNot’ expr | expr (’bitAnd’ | ’bitOr’ | ’bitXor’) expr;
comparison: ’!’ math | math (’==’ | ’!=’ | ’>’ | ’<’ | ’>=’ | ’<=’) math;
math: atom | ’(’ math ’)’ | math (’+’ | ’-’ | ’*’ | ’/’ | ’%’ | ’ˆ’) math;
atom: number | functionCall | variable | ’(’ expr ’)’ | arrayAccess | string | bitXor;
number: ’0’ | (’1’ | ... | ’9’) {(’0’ | ... | ’9’)};
identifier: {character};
character: (’A’ | ... | ’Z’) | (’a’ | ... | ’z’) | ’_’ | number;
string: ’”’ {character} ’”’;
eos: [lineComment] [comment] newline {eos};
newline: ’\n’ | ’\r\n’;
lineComment: ’//’ {character} -{’\n’ | ’\r’};
comment: ’/*’ {character | newline} ’*/’;