Description of ZKPDL

August 12, 2010

1 Syntax

A ZKPDL program consists of two blocks: a `computation` block and a `proof` block. Both blocks are optional; if a user wants a calculator (for either modular or integer arithmetic) then he can use just the `computation` block, but if he has all his values pre-computed and wants the zero-knowledge proof, he can use just the `proof` block.

1.1 The computation block

The `computation` block is itself comprised of two blocks: the `given` blockand the `compute` block. A sample computation block is as follows:

```
1  computation:
2      given:
3          group: G = <g,h>
4          group: G2
5          group: G3 = <f1,f2>
6              modulus: N
7          elements in G: a, b, c
8          exponents in G: x_1, x_2
9          integers: y, z
10     compute:
11         random exponents in G: r_1, r_2
12         random integer in [0,N): s
13         random prime of length 2^(y+z): p
14         c_1 := g^x_1 * h^r_1
```

1.1.1 The given block

This block names all the values necessary for the computation. The sample program above demonstrates the four types of values that can be declared here: group objects, group elements, group exponents, and integers. Group names must start with a letter and cannot have any subscripts; sample group declarations are in lines 3, 4, and 5 of the sample program. The sample program also demonstrates the various options the programmer has when declaring a group. If the generators for the group are to be used later on as bases (which we can see being done here in line 14), then the generators can be named. Furthermore, if the modulus of the group is to be used, it can also be named as seen on line 6.

The declaration of numerical objects (group elements, group exponents, or integers) are relatively straightforward; the only small note is that for both group elements and group exponents, a group name must be specified as seen in lines 7 and 8.

Every value declared in the `given` block must be input to the interpreter at compute time, and an exception will be thrown if any are missing.

1.1.2 The compute block

As seen in our sample program, there are two main types of computations that can be carried out: picking a random value with some given set of constraints, or performing an arithmetic operation. For the first type, there are three possible operations currently supported, as seen in lines 11, 12, and 13; note that in each of these, the value on the right side of the colon will be assigned as it is being declared. On line 11, random exponents in a given group are picked according to the order of the group. On line 12, a random number is being chosen in a given range; note that this range can contain arithmetic expressions (for example 2^x) and is not limited to integers or variable names. Similarly, on line 13 we see a prime number being picked of a certain length; again, the length can be given as an arbitrary arithmetic expression.

Finally, on line 14, we see a basic arithmetic operation being carried out. If this operation is carried out over the integers, the type for the variable being assigned will be an integer. If it is carried out in a group (as is the case here) then the types of the values will be checked to make sure they match up. For example, if the line were changed to read `c_1 = g^x_1 * f1^r_1`, then a compile-time exception would be thrown, as elements from different groups cannot be multiplied together. Similarly, if the line read `c_1 = x_1^g * h^r_1` an exception would again be thrown, this time because the user is attempting to raise a group exponent to a group element power, which is also prohibited. As it is written, however, line 14 is allowed and the resulting `c_1` value will be typed as an element in the group `G` (as well as assigned whatever value the right-hand expression evaluates to).

1.2 The proof block

The proof block consists of three blocks: the `given` block, the `prove knowledge of` block, and the `such that` block. Here is a sample proof block:

```
1  proof:
2      given:
3          group: G = <g,h>
4          integers: lower, upper
5          element in G: c1, c2, c3, d, f
6              commitment to x1: c1 = g^x1 * h^r1
7              commitment to x2: c2 = g^x2 * h^r2
8              commitment to x3: c3 = g^x3 * h^r3
9              commitment to y: d = g^y * h^r
10     prove knowledge of:
11         exponents in G: x1, x2, x3, y, r1, r2, r3, r
12     such that:
13         f = c1^(x2+x3) * c2^r
14         x1 = x2 * x3
15         x3 = y^2
16         range: lower <= y < upper
```

The first block, the `given` block, is almost syntactically identical to the `given` block described in Section 1.1.1. The significance of the block has changed though; whereas before it specified the values needed to perform the computations, it now specifies the common values shared between the prover and verifier. With this in mind, we can look at lines 6 through 9 to see the major difference in the `given` block, which is that commitment forms can be specified. In this case, each of the commitments is a commitment to a single value, but commitments can also be formed for multiple values (in which case all the values are listed in the commitment line).

The `prove knowledge of` block is syntactically fairly simple. It lists, separated by type, the private values that the prover knows and the verifier doesn't. This block may contain elements, exponents, and integers, but not groups (as groups are always regarded as common parameters).

Finally, the `such that` block specifies the relations between the values for which the prover is creating a zero-knowledge proof. As seen in lines 13 through 16, there are four types of proofs that can be specified. The first, on line 13, is proving knowledge of the discrete logarithm form of a value; in this case, proving knowledge of the values `x2`, `x3`, and `r` such that this equation is satisfied. The next type of proof, on line 14, is proving that a secret value is the product of two other secret values. The next type, on line 15, is just the special case of the multiplication proof in which the two secret values are the same. Finally, on line 16, we see range proofs, in which the prover proves that a hidden value is contained within some public range.

1.3 Syntactic sugar

To make programs easier to write, we have added in a number of additional features. We demonstrate these features in the following program:

```
1  computation:
2      given:
3          group: G = <g,h[1:k]>
4          exponents in G: x[1:k], v[1:k], x
5      compute:
6          random exponents in G: r[1:k]
7          for(i, 1:k, c_i := g^x_i * h_i^r_i)
8          C := g^x * for(i, 1:k, *, h_i^v_i)
9
10 proof:
11     given:
12         group: G = <g,h[1:k]>
13         elements in G: c[1:k], C
14             for(i, 1:k, commitment to x_i: c_i = g^x_i * h_i^r_i)
15         integer: len
16     prove knowledge of:
17         exponents in G: x[1:k], v[1:k], x, r[1:k]
18     such that:
19         for(i, 1:k, range: (-(2^len - 1)) <= x_i < 2^len)
20         C = g^x * for(i, 1:k, *, h_i^r_i)
```

As we can see in the program, there are two main features introduced: the use of `x[1:k]` as shorthand for `x_1,...,x_k` and the use of for loops. The first kind of for loop, which we see on lines 8 and 20, is similar to $\sum$ or $\prod$ in mathematical notation. After the `for` symbol, the index used is named, followed by the values over which the index should range, followed by the operation (either

addition or multiplication), followed by the values which should either be summed or multiplied. So, writing `for(i, 1:3, +, x_i)` is just shorthand for writing `x_1 + x_2 + x_3`.

The other type of for loop is more similar to for loops in traditional programming languages. We can see it used in lines 7, 14, and 19.

Finally, we also note the addition of the value `k` in the above program. This is very useful in allowing us to write programs for protocols which allow a variable number of inputs (for example, forming a CL signature on k values). At compile time, the value for k must be given to the interpreter. The constant is then substituted in and propagated throughout the program (and any subexpressions in which it is used are evaluated); more on this can be found in Section 2.

2 Usage

Our interpreter, when given a ZKPDL program, works in two main stages. In the first, which we refer to as *compile time*, the program is checked for syntactic correctness and a number of optimizations are run. First, any constants are substituted in and then propagated, and any for loops are unrolled. The interpreter then performs a number of safety checks. It first checks to see if any variables are being used that were not previously declared, and also checks to see any variables are being declared without being used. It will stop and throw an exception if the former problem occurs, but in the latter case it will just print out a warning message. It will then check that group types match up, and again throw an exception if this check fails. It then performs a number of operations to set up the environment for the next phase, and ends by caching a copy of itself so that if the same program is run again this phase can be skipped. No values need to be given to the interpreter at compile time, although if groups are given for which the generators will be used as bases for multi-exponentiations then the interpreter will perform caching to speed up the exponentation when it is performed later on.

In the second phase, *compute time*, the interpreter is given all the values needed for the computation (although as mentioned above, it may have already been given the groups). If it is acting as the prover, it performs all the computations specified in the `computation` block, followed by all the operations needed to output the zero-knowledge proof specified in the `proof` block. If it is the verifier, it will also take in the proof, and then perform all the computations necessary to verify the proof's validity.

Here is a sample usage, in C++, of our interpreter:

```
hashalg_t hashAlg = Hash::SHA1;
int stat = 80;

const GroupPrime* grp = parameters->getCashGroup();
ZZ x1 = 2;
ZZ x2 = 3;
group_map pg;
variable_map pv;
pg["G"] = grp;
pv["x_1"] = x1;
pv["x_2"] = x2;

InterpreterProver prover;
```

```
prover.check("multiplication.txt", pg);
prover.compute(pv);
ProofMessage proofMsg = ProofMessage(prover.getPublicVariables(),
     prover.computeProof(hashAlg));

group_map vg;
variable_map vv;
vg["G"] = parameters->getCashGroup();

InterpreterVerifier verifier;
verifier.check("multiplication.txt", vg);
verifier.compute(vv, proofMsg.publics);
bool verified = verifier.verify(proofMsg.proof);
```

3 ZKPDL Grammar

To enhance the overview of our language, we provide here a full EBNF grammar.

```
spec := (computation)?  (proof)?
computation := ‘‘computation’’ COLON ‘‘given’’ COLON givenList ‘‘compute’’ COLON
computeRandomList computeEquationList
proof := ‘‘proof’’ COLON ‘‘given’’ COLON givenList ‘‘prove’’ ‘‘knowledge’’ ‘‘of’’ COLON
knowledgeList ‘‘such’’ ‘‘that’’ COLON suchThatList
suchThatList := (suchThatRel)+
suchThatRel := equalRelation | rangeRelation | forRelation
knowledgeList := (exponentDecl | integerDecl)+
givenList := (groupDecl | elementsDecl | exponentDecl | integerDecl)+
randomBndDecl := (‘‘integer’’ | ‘‘integers’’) ‘‘in’’ LBRACKET expr COMMA expr RPAREN COLON
idGeneralDeclList
randomPrimeDecl := (‘‘prime’’ | ‘‘primes’’) ‘‘of’’ ‘‘length’’ expr COLON idGeneralDeclList
computeRandomList := (‘‘random’’ (randExponentDecl | randomPrimeDecl | randomBndDecl))*
groupDecl := ‘‘group’’ COLON identifierDecl (EQUAL setDecl)?  (‘‘modulus’’ COLON
subscriptIdentifierDecl)?
randExponentDecl := (‘‘exponent’’ | ‘‘exponents’’) ‘‘in’’ identifier COLON idGeneralDeclList
exponentDecl := (‘‘exponent’’ | ‘‘exponents’’) ‘‘in’’ identifier COLON idGeneralDeclList
elementsDecl := (‘‘element’’ | ‘‘elements’’) ‘‘in’’ identifier COLON idGeneralDeclList
(elementsEquationList)?
integerDecl := (‘‘integer’’ | ‘‘integers’’) COLON idGeneralDeclList
computeEquationList := (computeEquation)+
computeEquation := equalDeclRelation | comRelation | forRelation
forRelation := ‘‘for’’ LPAREN identifier COMMA expr COLON expr COMMA (rangeRelation
| genEqual) RPAREN
forCom := ‘‘for’’ LPAREN identifier COMMA expr COLON expr COMMA comRelation RPAREN
elementsEquationList := (elementsEquation)+
elementsEquation := comRelation | forCom
comRelation := ‘‘commitment’’ ‘‘to’’ idSubList COLON subscriptIdentifier EQUAL expr
genEqual := identifier (SUBSCRIPT (ID | INTLIT))? (EQUAL | CEQUAL) expr
equalRelation := subscriptIdentifier EQUAL expr
equalDeclRelation := subscriptIdentifierDecl CEQUAL expr
rangeRelation := ‘‘range’’ COLON expr (LTHAN | LEQ) expr (LTHAN | LEQ) expr | (GTHAN | GEQ)
expr (GTHAN | GEQ) expr
expr := prodExpr (ADD prodExpr | SUB prodExpr)*
forExpr := ‘‘for’’ LPAREN identifier COMMA expr COLON expr COMMA (ADD | MUL) COMMA expr
RPAREN
prodExpr := powExpr (MUL powExpr | DIV powExpr)*
powExpr := unaryExpr | baseExpr
unaryExpr := SUB baseExpr | baseExpr
baseExpr := INTLIT | subscriptIdentifier | LPAREN expr RPAREN | forExpr
setDecl := LTHAN idGeneralDeclList GTHAN
idSubDeclList := (subscriptIdentifierDecl)+
idGeneralDeclList := (idDeclGeneral)+
idDeclList := (identifierDecl)+
idSubList := (subscriptIdentifier)+
idList := (identifier)+
idDeclGeneral := idDeclRange | subscriptIdentifierDecl
idDeclRange := identifierDecl LBRACKET expr COLON expr RBRACKET
subscriptIdentifierDecl := identifier (SUBSCRIPT (ID | INTLIT))?
identifierDecl := ID
subscriptIdentifier := identifier (SUBSCRIPT (ID | INTLIT))?
identifier := ID
ID := (’a’..’z’ | ’A’..’Z’)(’a’..’z’ | ’A’..’Z’ | ’0’..’9’)*
INTLIT := ’0’ | (’1’..’9’)(’0’..’9’)*
```