CONTENTS

This supplemental material contains all programs and data used in the experiments in our paper submitted to IEICE TRANSACTIONS. The following table is a list of files and folders in this supplemental material.

File/Folder	Explanation
`LICENSE.md`	software license agreement in markdown format
`LICENSE.pdf`	software license agreement in pdf format
`LICENSE.txt`	software license agreement in plain text format
`Makefile`	make file
`README.md`	instruction manual in markdown format
`README.pdf`	instruction manual in pdf format
`README.txt`	instruction manual in plain text format
`experiments/`	sub-folder of data fiels for experiments
`samples/`	sub-folder of samples
`src/`	sub-folder of source codes
`template.latex`	pandoc template to convert markdown into pdf

NAME

- `findsplit` - find an optimal split of a dependency graph
- `gsconv` - convert a C-like scheme description into Tango format

SYNOPSIS

```
findsplit [OPTION]... INFILE
gsconv  <  INFILE  >  OUTFILE
```

DESCRIPTION

The `findsplit` is a program that finds an optimal split of a dependency graph written in a text format (called Tango format) specified below, then outputs three files corresponding to the source and split graphs in Graphviz dot format. To find an optimal solution, `findsplit` generates an intermediate file (named `G.lp` by default) corresponding to an instance of IP (Integer Programming) problem, and invokes one of the following external IP solvers by default.

Solver	Explanation
`gurobi_cl`	Gurobi Optimizer
`scip`	SCIP (Solving Constraint Integer Programs)
`cbc`	Cbc (Coin-or branch and cut)
`glpsol`	GLPK (GNU Linear Programming Kit)
`lp_solve`	lp_solve a Mixed Integer Linear Programming (MILP) solver

Therefore `findsplit` will fail if non of the above solvers are available. If the solver generates a solution file (named `G.sol` by default) and exits successfully, findsplit loads the solution and generates files (named `G0.dot` and `G1.dot` by default) corresponding to the split graphs. `findsplit` also generates a file (named `G.dot` by default) corresponding to the source graph.

The `gsconv` is a simple filter program that converts a C-like scheme description into Tango format.

It is recommended that the output dot files are converted into human-recognizable format by the following tools.

Tool	Explanation
`dot,sfdp`	Graphviz - Graph Visualization Software
`dot2tex`	dot2tex - A Graphviz to LaTeX converter
`pdflatex`	TeX

For example: to convert `G.dot` into `G.pdf`, use them as follows.

```
./gsconv  < samples/sample1.clsd > sample1.tango
./findsplit sample1.tango
dot2tex --usepdflatex -f tikz --autosize -c G.dot > G.tex
pdflatex -interaction=nonstopmode G.tex
```

If the graph is too large (about 100 nodes), `dot2tex` may hung up. To convert such a large graph, do as follows. If you don't like the output, adjust the scale given to `sed`.

```
./gsconv  < samples/sample2.clsd > sample2.tango
./findsplit sample2.tango
dot2tex --usepdflatex -f tikz -c --preproc G.dot | sfdp -Txdot | dot2tex -f tikz -c > G.raw
sed 's/join=bevel,/join=bevel,scale=5/' G.raw > G.tex
pdflatex -interaction=nonstopmode G.tex
```

HOW TO BUILD

First of all, prepare the follows.

- UNIX-like development environment (`sh,mv,cp,rm,chmod,cat,sed,awk,perl,make,...`)
- yacc or bison parser generator
- lex or flex lexical analyzer generator
- clang or gcc c99/c++11 compiler
- boost C++ LIBRARIES
- source code of `findsplit/gsconv`

Then, on the command line (shell), just type

```
    make
```

`findsplit` and `gsconv` are confirmed to build correctly under the following environments with an appropriate configuration.

Environment	Version	Architecture	Note
FreeBSD	10.1	amd64	yacc/lex/clang
Ubuntu	15.04	x86_64	bison/flex/gcc
Cygwin	1.7.27	x86_64	bison/flex/gcc (mingw for gsconv)

DATA FIELS FOR EXPERIMENTS

The files used in the experiments in Section 5 of the paper are stored in `experiments` folder. They are dependency graphs written in Tango format described below. We use `gsconv` converter to generate dependency graphs of DLIN-based Groth-Sahai proofs, which is eventually incorporated into the main part of the dependency graph by hand. The inputs of `gsconv` are stored in the sub-folders of `experiments`. To verify our results, do as follows after building `findsplit`, which is the main body of IPConv.

```
cd experiments
make
```

Results will be displayed on the console. Processing time may somewhat differ from our results because it depends on the environment such as the available solver. Especially, it takes one or more hours to split `TraceGroupEnc.tango` if no solver other than `glpsol` and `lp_solve` is available.

The source and split graphs in pdf format are also stored in `experiments` folder.

FINDSPLIT OPTIONS

options for output filename:

```
-o{filename}: set output file name (default: G.dot)
-f{filename}: set G0 file name (default: G0.dot)
-h{filename}: set G1 file name (default: G1.dot)
-l{filename}: set lp file name (default: G.lp)
-s{filename}: set solution file name (default: G.sol)
-b{basename}: set all of the above basename.(dot,g0,g1,lp,sol)
```

options for output format:

```
--raw       : raw dot format
--tango_dot : tango dot format
--tex       : dot2tex dot format (default)
```

options for IP solver:

```
--anysolver : invoke one of the followings (default)
--gurobi    : invoke gurobi_cl
--scip      : invoke scip
--cbc       : invoke cbc
--glpsol    : invoke glpsol
--lp_solve  : invoke lp_solve
--nosolver  : just output lp file
```

options for backtracking:

```
--montecarlo: Monte Carlo method (2^n)
--bruteforce: backtrack without pruning test
--backtrack : backtrack with pruning test
--naive     : same as bruteforce, but naive implementation (for debug)
--crypto2014: method of crypto2014 paper (for comparison)
--unlimit   : unlimited backtrack mode
--limit=n   : set backtrack-limit = 2^n (n>=0)
--limit     : set backtrack-limit = 2^22 (default)
```

options for optimality check:

```
--check=n   : set check-limit = 2^n (n>=0)
--check     : set check-limit = 2^16
--nocheck   : do not check (default)
```

other options:

```
--ratio=r   : set G1/G0-ratio = r (>=0,default: 1.0)
--ratio     : set G1/G0-ratio = 2.0
--visible   : show constraint nodes (default)
--invisible : hide constraint nodes
--          : end of options
```

GSCONV OPTIONS

`gsconv` has no options.

TANGO FORMAT

Tango format is a plain text language to describe a dependency graph that abstracts a cryptographic scheme. In Tango format, a dependency graph consists of one or more blocks defined by some statements. Each statement is written on a separate line.

A `SYMBOL` in `[` and `]` is called a *block indicator*. A block indicator indicates the start of a new block, that is a block starts with the indicator and ends with the next indicator or the end of the file. There are several kinds of block indicators.

```
[Dependencies]
[Pairings]
[Prohibits]
[Constraints]
[Priority]
[Weight]
```

A block indicator can be written on any newline. The same block indicators can be written any number of times in a file.

There are several types of statement as listed below. The semantics of a statement depends on the block that the statement belongs to.

Type of statement	Example
list	`A,B,C,D`
arrow	`A,B,C,D -> E`
assignment	`A,B,C,D = E`
exclusive assignment	`A,B,C,D != E`
pairing	`(A,B)`

To declare nodes, list the nodes in a `[Dependencies]` block. For example:

```
[Dependencies]
A,B,C,D,E
```

Declarations of nodes can be omitted. Furthermore multiple declarations for a unique node are acceptable.

To define dependencies between some nodes, write arrow statements in a `[Dependencies]` block. For example:

```
[Dependencies]
A,B,C -> D
E,F -> A
```

To define pairings, write pairing statements in a `[Dependencies]` block, or list pairs of nodes in `[Pairings]` block. For example:

```
[Dependencies]
(A,B)
(C,D)
[Pairings]
E,F
G,H
```

It is an obsolete style that remains for backward compatibility to define pairings in a `[Pairings]` block.

To define prohibited nodes, list the nodes in a `[Prohibits]` block. For example:

```
[Prohibits]
A,B
```

This means "`A` and `B` are prohibited nodes."

To define prohibited nodes in a fixed group, write assignment statements in a `[Prohibits]` block. For example:

```
[Prohibits]
A = 1
```

This means "`A` is a prohibited node and `A` is in $\mathbb{G}_1$."

To define constraints between prohibited nodes, write assignment statements in a `[Prohibits]` block. For example:

```
[Prohibits]
A = B
```

This means "`A` and `B` are prohibited nodes and `A` and `B` are in the same group."

To define exclusively prohibited nodes, write exclusive assignment statements in a `[Prohibits]` block. For example:

```
[Prohibits]
A != B
```

This means "`A` and `B` are prohibited nodes and `A` is not in the group that `B` is in."

To define constraints between regular nodes, list nodes in a `[Constraints]` block. For example:

```
[Constraints]
A,B,C
```

This means "A,B and C are in the same group." One list statement corresponds to one constraint.

To define a priority of nodes, list nodes in a [Priority] block. For example:

```
[Priority]
A,B,C
```

This means "A,B and C have a high priority to evaluate." One list statement corresponds to one priority level. The former list has the higher priority in a file. If a node has many priority, the last priority is effective. The non-listed nodes have the lowest priority.

To define a weight of nodes, write assignment statement in a [Weight] block. For example:

```
[Weight]
A,B,C = 10
```

This means "A,B and C have weight 10." Weight must be a natural number. If the weight of a node is defined many times, the last definition is effective. The default weight of a node is 1.

A comment starts with %, extending to the newline character. For example:

```
% can be used for single-line comments.
```

To define nodes with TeX labels, list nodes in a [Dependencies] block with a special comment as follows.

```
[Dependencies]
alpha, beta, gamma % $\alpha$, $\beta$, $\gamma$
```

A semicolon ; is also a statement separator as a newline character is, but it cannot be a comment terminator.

A colon operator : and a definition operator := are identical to a assignment operator =.

A negative assign operator ~=, an assign negative operator =~, and an assign not operator =! are identical to a not assign operator !=.

GSCONV TUTORIAL

An input of `findsplit` must be written in Tango format that abstracts a cryptographic scheme. Tango format is a very simple language to understand, but it may be incomprehensive for a large cryptographic scheme as a large assembly code is. Therefore we provide the `gsconv` program to convert a (K&R) C-like scheme description into Tango format. The following pseudo-code represents a scheme which computes C and E from A,B and D via group operations (multiplication and exponentiation), and outputs a result of pairing e(C,E).

```
Sample(){
    integer x ;
    group A,B,C,D,E ;

    if(x == 0){
        C = A * B ;
        E = D ;
    }else{
        C = D^x ;
        E = D^3 ;
    }
    return e(C,E) ;
}
```

The gsconv program accepts the above code, then outputs the following Tango format.

```
[Dependencies]
A,B -> C
D -> E
D -> C
D -> E
(C,E)
```

The syntax of gsconv language is mostly similar to that of C language.

A group operation must be written using a * operator which can't be omitted.

In gsconv language, ^ means a power operator which has right associativity and higher precedence than * operator. Use (+) for xor operator instead.

gsconv language has inline statement to embed Tango format directly. For example:

```
inline "alpha,beta % $\alpha$, $\beta$" ;
```

gsconv converts the above statement into

```
alpha, beta % $\alpha$, $\beta$
```

A string literal can contain newline characters.

Comments in gsconv language is almost the same as that in C++ language. A multi-line comment starts with /* and end with */. A single-line comment starts with // and extending to the next newline character.

A description of an algorithm must be placed inside a function such as

```
main(){
    ...
}
```

It is no matter that function named other than main is defined. Multiple functions can be defined if all functions have unique names in a file . For example:

```
setup(){ ... }
sign(){ ... }
verify(){ ... }
```

Parentheses after the function name () can't be omitted. Don't write anything in this parentheses currently even though we plan to implement prototype declaration of arguments in the future. Otherwise gsconv will be confused.

Instead of the built-in datatypes of C, gsconv language has the following datatypes.

Type	Explanation
group	source group of pairing
target	target group of pairing
integer	integer (endomorphism ring of the group)
string	string of some data
crs	common reference string of a GS proof system
proof	proof string of a GS proof system

To declare variables, write as follows (declaration statement).

```
group   A,B,C,X,Y,Z,f,g,h ;
integer alpha, beta, gamma, delta ;
target  gT ;
```

Variables always have the global scopes. For example, the variables declared in the above `setup()` function can be referred in `sign()` and `verify()` freely. There is no concept of local scope in the current `gsconv` language.

To assign a variable, write as follows (assignment statement).

```
X = Y^alpha * Z^beta ;
```

`gsconv` converts the above assignment into the following Tango format.

```
Y,Z -> X
```

Similarly, `gsconv` converts a pairing expression

```
gT = e( X^gamma, Z^beta ) ;
```

into Tango format

```
(X,Z)
```

A variable can be initialize when it is declared (initialization statement). For example, the statement

```
group W = X*Y*Z ;
```

in `gsconv` language will be converted as

```
X,Y,Z -> W
```

in Tango format.

Declaration statement and initialization statement can be placed anywhere in the file as long as the name of the variable is new. If a variable of the same name already declared or initialized in the file, `gsconv` will assert an error.

`gsconv` language doesn't have typedef statement, goto statement, case statement and labeled statement.

`gsconv` language has no preprocessor directives.

`gsconv` language has a special syntax to describe a GS (Groth-Sahai) proof system.

To generate a common reference string of a GS proof system, write as follows.

```
 crs crs1 = GS_setup(g) ;
```

This means "initialize the variable of type `crs` named `crs1` with a common reference string derived from a `group` variable `g`." The variable `g` must be declared or initialized as a variable of type `group` beforehand.

To generate a proof string of a GS proof system, write as follows.

```
proof proof1 = GS_proofwi(crs1, e(A,[X]) * e(B,[Y]) == 1 ) ;
```

The above statement means "let `proof1` be a witness indistinguishable proof of the proposition specified in the second argument of `GS_proofwi()` under the common reference string `crs1` and commitment of `X` and `Y`." `crs1` is an initialized variable of type `crs` as the above. A committed variable is described by an appropriate variable of type `group` or type `integer` in `[` and `]`. The commitment of a committed variable is automatically generated. To generate the commitment of a variable `X` explicitly w.r.t. `crs1`, write as follows.

```
crs1.[X] ;
```

And `gsconv` will generate the commitment if it is not yet generated. A proposition to prove is described by an equation. Notice that the equal sign of an equation is not `=` but `==` as in the C language.

To describe a zero knowledge proof write `GS_proofzk()` instead of `GS_proofwi()`. To prove a conjunction of propositions, just write multiple `GS_proofwi()` or write conjunct equations as follows

```
GS_proofwi( crs1,
    alpha * [beta]                   == 1          &&
    alpha * [beta] + [gamma] * delta == 1          &&
    [alpha] * [beta]                 == 1          &&
    [g]^alpha * [h]^beta             == 1          &&
    g^[alpha] * h^[beta]             == f^[gamma]  &&
    [g]^[alpha] * h^[beta]           == [f]^gamma  &&
    e(A,[X]) * e(B,[Y])              == 1          &&
    e(A,[X]) * e([B],[Y])            == 1
) ;
```

To verify a proof string, write as follows.

```
integer result = GS_verify(proof1) ;
```

No nodes or edges will be generated from the above statement, because all the nodes and edges for a GS proof system are generated by `GS_setup()`, `GS_proofwi()`, and `GS_proofzk()` in the current implementation.

To refer a commitment, write as follows.

```
crs1.[X]
```

A commitment has three `group` variables, because current implementation of the GS proof system is based on the DLIN problem. To refer a variable in a commitment, write as follows.

```
crs1.[X].c1
```

`c1` is a field specifier to specify the first `group` variable in the commitment. The statement

```
crs1.[X].c1 = Y^alpha * Z^beta ;
```

generates a Tango format like

```
Y,Z -> $c1_0
```

Available field specifier is one of the followings.

```
c1, c2, c3, d1, d2, d3
```

`c1` and `d1` have an identical semantics. There is no difference between them. `c2` and `d2` are also identical, and so on.