Crypto Grammar

Sanat Anand

July 2017

1 Grammar

⟨*program*⟩ ::= ⟨*protocol_list*⟩

⟨*protocol_list*⟩ ::= ⟨*protocol*⟩
| ⟨*protocol*⟩ ⟨*protocol_list*⟩

⟨*protocol*⟩ ::= ⟨*prot_decl*⟩ '{'⟨*opt_uses*⟩ ⟨*statement_list*⟩ '}'

⟨*prot_decl*⟩ ::= PROTOCOL NAME '('⟨*opt_arg_list*⟩ ')'':'PARTY
⟨*p_list*⟩

⟨*p_list*⟩ ::= ⟨*p_list_elem*⟩
| ⟨*p_list*⟩ ','⟨*p_list_elem*⟩

⟨*p_list_elem*⟩ ::= ⟨*exp*⟩
| ⟨*in_decl*⟩

⟨*in_decl*⟩ ::= IN ⟨*opt_each*⟩ ⟨*term*⟩ '{'⟨*decl_list*⟩ '}'

⟨*decl_list*⟩ ::= ⟨*decl*⟩
| ⟨*decl*⟩ ';'⟨*decl_list*⟩

⟨*decl*⟩ ::= ⟨*type*⟩ ⟨*varlist*⟩

⟨*opt_arg_list*⟩ ::= ϵ
| ⟨*arg_list*⟩

⟨*arg_list*⟩ ::= ⟨*arg*⟩
| ⟨*arg*⟩ ','⟨*arg_list*⟩

⟨*arg*⟩ ::= ⟨*type*⟩ ⟨*term*⟩
| '('⟨*prot_decl*⟩ ')'

⟨*term*⟩ ::= ⟨*var*⟩ ⟨*dimlist*⟩
| ENV
| IDEAL
| RAND_PORT
| ENV_PORT

⟨*dimlist*⟩ ::= ϵ
| '['⟨*exp*⟩ ']'⟨*dimlist*⟩
| '['']'⟨*dimlist*⟩

⟨*exp*⟩ ::= ⟨*exp*⟩ '+' ⟨*exp*⟩
| ⟨*exp*⟩ '-' ⟨*exp*⟩
| ⟨*exp*⟩ '*' ⟨*exp*⟩
| ⟨*exp*⟩ '/' ⟨*exp*⟩
| '-' ⟨*exp*⟩
| ⟨*exp*⟩ B_AND ⟨*exp*⟩
| ⟨*exp*⟩ OR ⟨*exp*⟩
| NOT ⟨*exp*⟩
| ⟨*exp*⟩ EQ ⟨*exp*⟩
| ⟨*exp*⟩ LT ⟨*exp*⟩
| ⟨*exp*⟩ GT ⟨*exp*⟩
| ⟨*exp*⟩ NE ⟨*exp*⟩
| ⟨*exp*⟩ LE ⟨*exp*⟩
| ⟨*exp*⟩ GE ⟨*exp*⟩
| '('⟨*exp*⟩ ')'
| ⟨*exp*⟩ '|' ⟨*exp*⟩
| ⟨*exp*⟩ '.' ⟨*exp*⟩
| ⟨*exp_list*⟩
| ⟨*fixed_list*⟩
| ⟨*term*⟩

⟨*fixed_list*⟩ ::= DOUBLE_OPEN ⟨*exp*⟩ DOUBLE_DOT ⟨*exp*⟩ DOUBLE_CLOSE

⟨*var*⟩ ::= NAME
| '#' NAME
| '@'
| RAND
| ⟨*constant*⟩

⟨*opt_uses*⟩ ::= ϵ
| USES ⟨*prot_decl_list*⟩ ';'

⟨*prot_decl_list*⟩ ::= ⟨*prot_decl*⟩
| ⟨*prot_decl*⟩ ⟨*prot_decl_list*⟩

⟨*statement_list*⟩ ::= ⟨*statement*⟩
| ⟨*statement_list*⟩ ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*assignment_statement*⟩
| ⟨*mult_statement*⟩
| ⟨*pvt_statement*⟩
| ⟨*send_statement*⟩
| ⟨*session_call*⟩
| ⟨*tying_statement*⟩

| ⟨*declaration_statement*⟩
| ⟨*protocol*⟩
| ⟨*loop_statement*⟩
| ⟨*cond_statement*⟩
| ⟨*wrap_statement*⟩
| ⟨*connect_statement*⟩
| ⟨*abort_statement*⟩
| ⟨*seq_statement*⟩
| ⟨*start_statement*⟩
| ⟨*set_statement*⟩
| ⟨*forward_statement*⟩

⟨*abort_statement*⟩ ::= ABORTING ';'

⟨*wrap_statement*⟩ ::= WRAP ⟨*exp*⟩ ⟨*statement*⟩

⟨*start_statement*⟩ ::= START ⟨*exp*⟩ ';'

⟨*seq_statement*⟩ ::= BEFORE ⟨*opt_each*⟩ ⟨*statement*⟩ DO ⟨*statement*⟩
| AFTER ⟨*opt_each*⟩ ⟨*statement*⟩ DO ⟨*statement*⟩

⟨*opt_each*⟩ ::= ϵ
| EACH

⟨*assignment_statement*⟩ ::= ⟨*exp*⟩ ASSIGN ⟨*exp*⟩ ';'

⟨*mult_statement*⟩ ::= '{'⟨*statement_list*⟩ '}'

⟨*pvt_statement*⟩ ::= IN ⟨*opt_each*⟩ ⟨*term*⟩ '{'⟨*statement_list*⟩ '}'

⟨*send_statement*⟩ ::= ⟨*exp*⟩ SEND ⟨*exp*⟩ ';'

⟨*session_call*⟩ ::= ⟨*opt_partial*⟩ OPEN ⟨*term*⟩ AS ⟨*prot_call*⟩ '{'⟨*statement_list*⟩ '}'

⟨*opt_partial*⟩ ::= ϵ
| PARTIAL

⟨*prot_call*⟩ ::= NAME '('⟨*opt_id_list*⟩ ')'

⟨*opt_id_list*⟩ ::= ϵ
| ⟨*id_list*⟩

⟨*id_list*⟩ ::= ⟨*variable*⟩
| ⟨*id_list*⟩ ','⟨*variable*⟩

⟨*tying_statement*⟩ ::= ⟨*exp*⟩ ':'⟨*exp*⟩ ';'

⟨*declaration_statement*⟩ ::= ⟨*type*⟩ ⟨*varlist*⟩ ';'

⟨*type*⟩ ::= PARTY
| INTEGER
| SESSION

| UNKNOWN
| PORT
| INPORT
| OUTPORT
| BOOL
| FIELD
| NAME

⟨*varlist*⟩ ::= ⟨*term*⟩
| ⟨*term*⟩ ‘,’⟨*varlist*⟩

⟨*loop_statement*⟩ ::= FOR EACH ⟨*variable*⟩ OF ⟨*exp*⟩ ⟨*opt_cond*⟩ ⟨*statement*⟩
| FOR EACH ⟨*exp*⟩ ⟨*opt_cond*⟩ ⟨*statement*⟩
| FOR EACH ⟨*variable*⟩ OF ⟨*exp*⟩ ⟨*statement*⟩
| FOR EACH ⟨*exp*⟩ ⟨*statement*⟩

⟨*opt_cond*⟩ ::= EXCEPT ⟨*exp*⟩
| WHERE ⟨*exp*⟩

⟨*cond_statement*⟩ ::= IF ‘(’⟨*exp*⟩ ‘)’⟨*statement*⟩
| IF ‘(’⟨*exp*⟩ ‘)’⟨*statement*⟩ ELSE ⟨*statement*⟩

⟨*set_statement*⟩ ::= SET ⟨*exp*⟩ AS ⟨*prot_call*⟩ DOUBLE_COLON ⟨*exp*⟩
‘;’

⟨*connect_statement*⟩ ::= CONNECT ⟨*exp*⟩ ⟨*to_and*⟩ ⟨*exp*⟩ ‘;’

⟨*to_and*⟩ ::= TO
| AND

⟨*exp_list*⟩ ::= ‘{’⟨*exp_list_elem*⟩ ‘}’

⟨*exp_list_elem*⟩ ::= ⟨*exp*⟩
| ⟨*exp_list_elem*⟩ ‘,’⟨*exp*⟩

⟨*forward_statement*⟩ ::= FORWARD ⟨*exp*⟩ TO ⟨*exp*⟩ ‘;’

⟨*variable*⟩ ::= NAME

⟨*constant*⟩ ::= INTEGER_NUMBER

2 Semantics

2.1 Protocol

- protocol: prot_decl ’{’ opt_uses statement_list ’}’
- prot_decl: PROTOCOL NAME ’(’ opt_arg_list ’)’ ’:’ PARTY p_list

Standard protocol definition. Protocols may be defined within other protocols. Standard scoping rules are applied. We have an optional USES statement which must specify at the very begining which other protocols are called by this one. The argument list can be used to pass standard formal variables (may be arrays but bounds must be specified beforehand) or other protocols. i.e. other protocols can also be passed as arguments to protocols. The list of real parties for the protocol must be specified (may be arrays of parties but bounds must be specified). In addition, it can also be used to specify list of ports in the parties which are in the protocol. This port declaration is done because ports are outward facing. i.e. protocols calling this protocol may need knowledge of the ports in the parties. These ports must be declared in an IN clause as ports are private to parties.
The prot_decl part of the protocol is the actual declartion. It is externally advertised for other protocols to call it.

2.2 Term

- term: var dimlist | ENV | IDEAL | RAND_PORT | ENV_PORT

There are two special parties - The environment and the Ideal party. The environment is used to denote the caller of every party in the protocol. In reality, each party has a different caller but since callers are not part of the protocol, we use a single symbol Environment to denote all of them. The Ideal party is a special party which doesn't really exist. It is used to specify fuctionalities which are then compiled into protocols by replacing the code for Ideal party by some interative protocol (eg - GMW compiler, BGW compiler etc) or by simply running the Ideal party code in some physical trusted party.
Similarly, every party has two special ports - Random port and Environment port. The RANDOM port is assumed to contain a lot of true randomness (in reality it'll be simulated by some PRG). The party receives bits from the RANDOM port whenever it wishes to sample random bits. The environment port is used to mediate communication with the party's own environment.
The language also allows you to build multi-dimentional arrays of any type (int, party, port etc.) whose bounds may or may not be specified beforehand (Depends case to case, specifics yet to be figured).

2.3 Expression

2.3.1 Arithmetic Expression

- exp: exp '+' exp | exp '-' exp | exp '*' exp | exp '/' exp | '-' exp

The standard arithemetic expressions like BODMAS are part of the language and it can easily be extended to include modulo and other requisite operations if required. More importantly, the particular FIELD must specify the operations and how they are to be performed on its elements.

2.3.2 Boolean Expression

- | exp B_AND exp | exp OR exp | NOT exp | exp EQ exp | exp LT exp | exp GT exp | exp NE exp | exp LE exp | exp GE exp | '(' exp ')'

Standard relational and boolean operations are also present

2.3.3 Party Expression

- | exp '|' exp | exp '.' exp

There are two party operations '.' (sub-party) and '—' (port) - The sub-party operation is used to denote the particular party inside a party which is being referred to. It can have arbitrary nesting. The port operation is used to specify the particular port in a party.

2.3.4 List expressions

- | exp_list | fixed_list | term

An language also allows for expression lists (although the lists themselves should be typed - debatable) and fixed lists (python like) to specify a numerical range or something of that sort.

2.4 Variables

- var: NAME | '#' NAME | '@' | RAND | constant

Array indices can also be qualified by a '#' which indicates that particular dimention is being iterated through. This would be used in any EACH statement (IN EACH, FOR EACH etc). The language will automatically detect array bounds for that dimension and set the iteration accordingly. The variable which is qualified by '#' will denote the indice value in that iteration for statements inside the EACH clause.
We use the '@' symbol inside the WRAP statement as a macro to denote the party which is being wrapped. This is entirely syntactic sugar.
An assignment of a variable to a '$' symbol represents the variable to take a random value. The method of generating randomness must be specified in the field of which the variable is a part. (Later we can qualify this to take distributions as argument).

2.5 Types

2.5.1 General types

- type: PARTY | INTEGER | SESSION | UNKNOWN

The UNKNOWN type is used a special type which can take any other type value. It is like a base class for every other type and allows only equality checking (useful for communication when you don't know what type of message you'll receive).

2.5.2 Port types

- | PORT | INPORT | OUTPORT

Ports, inports and outports are just buffers for communication. Messages can be sent to and received from them. An inport only allows receiving messages from it and an outport only allows sending to it. (The receiving or sending is done at the other side by the port it may be connected to).

2.5.3 Field types

- | BOOL | FIELD | NAME

Field is a meta-type (like class or struct) used to specify other types. Each field declared is a type in itself. Fields can't be declared in the code body and must be specified as formal arguments for the protocol. Because each field must be instantiated at runtime and only formal variables are visible to the runtime. Every field which is declared is a type in itself and we must type-check our code considering all if them seperately.
Later we can have other meta-constructs like RINGS, GROUPS, POSETS etc.

2.6 Statements

2.6.1 Abort statement

- abort_statement: ABORTING ';'

If a party sees abort, it immediately signals to the parent party and all its children party. Then it terminates. By chain action, the parents and children all terminate after signalling to their parents and children. This ways, the whole execution of any party associated with that terminates. (Later we can also add qualifiers to specify if something other than usual is to be done on abort).

2.6.2 Set Statement

- set_statement: SET exp AS prot_call DOUBLE_COLON exp ';"

Set statement is used to simulate a real party in the current protocol as some party of another protocol. It sets the execution steps of that particular party as that dictated by in the protocol it is set to. (We can decide whether we're allowed to reset the same party to something else).

2.6.3 Wrap Statement

- wrap_statement: WRAP exp statement

Wrap statement is used to create a wrapper around the code of a party which has been has set. It is used to modify the execution steps for that party and can reference the code for the party by before and after statements. i.e. using the wrap, certain sequence of statements will be executed whenever a certain statement is seen in the code of the party it has been set to.
Note that wrap statement can only be used for parties which've been set. Otherwise it'd be meaningless. (It is debatable whether it should be allowed to wrap unset parties). Also, the macro '@' can be used to denote the party which is wrapped (inside the wrapped statement).

2.6.4 Start statement

- start_statement: START exp ';'

Start statement is used to indicate parties to begin running the code for the parties they've been set to (can only be done for set parties). Note that this may be just the bare code for the party it has been set to or there may be some wrapper around the original code specified by the wrap statement.

2.6.5 Sequence Statements

- seq_statement: BEFORE opt_each statement DO statement | AFTER opt_each statement DO statement

The before and after statements are used inside the wrap block to modify original execution steps for a set party. The before statement specifies that whenever a particular statement is seen in the original code block, a certain sequence of steps are to be executed before it. The after statement does so after executing the original statement first. The each qualifier specifies that every instance of that statement in the original code block will be preceded/succeeded by the corresponding execution steps. An unqualified statement will just modify the first occurrence.
(We could also look into allowing before/after for a sequence of statements. i.e. execution steps are modified when a sequence of statements is seen)

2.6.6 Private Statement

- pvt_statement: IN opt_each term '{' statement_list '}'

The IN block is used to write code which is privately executed by some party. We can have nested IN block but each level of nesting can only correspond to parties within parties.

2.6.7 Send Statement

- send_statement: exp SEND exp ';'

Send statement is used to communicate between parties, real or virtual. Two kinds of structures can send or receive data - parties and ports.
Communication between parties is also of three types - Communication with your own environment, communication with children parties and communication with other parties (Note that sessions are also counted as children parties). Communication with children parties and environment may be unqualified at the other end. i.e. you need not specify a variable at the other end. In standard party to party communication, you need to specify both the sending variable in the first party and the receiving one in the second party.
In addition to that, you can simply read from or your to your own ports. This will be a port to party communication. A variable must be specified in the party side into which the information is read or from which the information is written. Port to port send operations can also happen between ports of the same party (or children party??). This will cause a variable to be written from the sending port to the receiving port.
Exactly which send operations are allowed is up for discussion. (Which ports are accessible to you for sending and receiving? Parent? Children? Own?)

2.6.8 Session Call Statement

- session_call: opt_partial OPEN term AS NAME '(' opt_id_list ')' '{' statement_list '}'

Seesion call is just a shorthand for creating additional parties and setting them as some protocol's parties. Then connecting it with the real party ports. This is subsumed by the SET statements (as discussed with Rahul) but is a more commonly used construct. It is equivalent to a function call in the sense we can pass it parameters and attach parties. In the corresponding statement block, all session parties must be attached to real protocol parties (this can only be checked at runtime).
Additionally, we may decide to partially open a session whereby we need not tie all the parties but then all the open ports (for parties which have not been tied) must be seperately handled. (Like currying)

2.6.9 Tying Statement

- tying_statement: exp ':' exp ';'

Tying statement is used inside the session call statement block to tie session parties to real protocol parties. It must be checked at runtime that all session parties are tied (multiple session parties may be tied to the same protocol party).

2.6.10 Loop Statement

- loop_statement: FOR EACH variable OF exp opt_cond statement | FOR EACH exp opt_cond statement | FOR EACH variable OF exp statement | FOR EACH exp statement

Only FOR loops are permitted where index runs over some array index or list index (specified by a '#' on the variable). The '#' acts as an indication as well as a declaration for the index variable. An indexless array may be written which is a shorthand for a hashed index but the hashed variable is never used.
We may also qualify the loop with a "variable OF" clause where the written variable takes the value of the specified expression in each iteration. If there are multiple '#' variables, then it acts as a nested loop whereby all possible combinations of values of '#' indices are run over in the loop in lexicographical order.
The optional conditional has two flavors - WHERE and EXCEPT. WHERE will make the statements execute when the condition is true and EXCEPT will do it when the condition is false.

2.6.11 Connect Statement

- connect_statement: CONNECT exp to_and exp ';'

Connect is used to permanently join two ports such that all information received in one port is forwarded to the other port. Ports across parties can be connected (exact restrictions can be debated - whether any two ports can be connected or should there be a restriction for parent-child and the like). It has two flavors - TO and AND. TO forms a one sided connection where messages are only forwarded from the left port to the right. AND creates a dual connction where messages are forwarded in both directions.

2.6.12 Forward Statement

- forward_statement: FORWARD exp TO exp ';'

Forward is a one-time connect statement where all the information is a port is forwarded to the other port.

3 Type Rules

This section formally defines the type rules for Cooler. The types in our language are as follows:
Int, *Str*, *Field*

3.1 Type Environment

The *type* environment of Cooler consists of two parts.

- An object environment that maps identifiers to types.

  $$O(Idf)=T$$

- The other environment is called the context environment denoted by C.

3.2 Type Checking Rules

The notation used for each of the type checking rule is as follows :

$$\frac{\vdots}{O, C \vdash e : T}$$

here O and C represents the two type environments respectively and the expression e evaluates to type T.

Variable declaration The rule for variable declaration is as follows

$$\frac{O(Idf)}{O, C \vdash Id : T} \text{ [Var]}$$

General Types The following are rules for general types

$$\frac{i \text{ is an integer constant}}{O, C \vdash i : Int} \text{ [Int]}$$

$$\frac{s \text{ is an string constant}}{O, C \vdash s : Str} \text{ [String]}$$

$$\frac{}{O, C \vdash true : Bool} \text{ [True]}$$

$$\frac{}{O, C \vdash false : Bool} \text{ [False]}$$

$$\frac{}{O, C \vdash unknown : Unknown} \text{ [Unknown]}$$

Port Types consists of three types namely Port, InPort and OutPort

$$\frac{\begin{array}{l} O(Id) : Str \\ O, C \vdash e_1 : Int \\ \vdots \\ O, C \vdash e_n : Int \end{array}}{O, C \vdash Id[e_1] \ldots [e_n] : Str} \text{ [Port]}$$

Statements The following rules type check statements

$$\frac{\begin{array}{l} O, C \vdash e_1 : Bool \\ O, C \vdash e_2 : T \\ O, C \vdash e_3 : T \end{array}}{O, C \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{ [If]}$$

$$\frac{}{O, C \vdash \text{for each } e_1} \text{ [For-Each]}$$

Expressions The following rules type check expressions

$$\frac{\begin{array}{l} O, C \vdash e_1 : T_1 \\ O, C \vdash e_2 : T_2 \\ \vdots \\ O, C \vdash e_n : T_n \end{array}}{O, C \vdash \{e_1; e_2; \ldots e_n;\} : T_n} \text{ [}Sequence\text{]}$$

$$\frac{\begin{array}{l} O, C \vdash e_1 : T_1 \\ O, C \vdash e_2 : T_2 \end{array}}{O, C \vdash e_1 \Longrightarrow e_2 : T_2} \text{ [}\Longrightarrow\text{]}$$

Operations Type checks for operations are as follows:

$$\frac{\begin{array}{l} O, C \vdash e_1 : Bool \\ O, C \vdash e_2 : Bool \\ op \in \{AND, OR, NOT\} \end{array}}{O, C \vdash e_1 \; op \; e_2 : Bool} \text{ [Binary-operation]}$$

$$\frac{\begin{array}{l} O, C \vdash e_1 : Int \\ O, C \vdash e_2 : Int \\ op \in \{+, -, *, /\} \end{array}}{O, C \vdash e_1 \; op \; e_2 : Int} \text{ [Arithmetic-operation]}$$

$$\frac{\begin{array}{l} O, C \vdash e_1 : Field \\ O, C \vdash e_2 : Field \\ op \in \{\odot, \oplus\} \end{array}}{O, C \vdash e_1 \; op \; e_2 : Field} \text{ [Field-operation]}$$

4 Operational Semantics

...