SFDL Specification - Version 2.0

September 4, 2008

Programs in Secure Function Definition Language (SFDL) instruct a virtual "trusted party" what to do. The SFDL compiler compiles the program into a low level language that can be read by secure function evaluation peer to peer software. When the multi-party, secure function evaluation, peer to peer software run the compiled form of the program, they implement correctly and securely the fictional trusted party.

This document specifies version 2.0 of the SFDL language which is intended for multiparty computation. Version 1.0 was intended for two-party computation and was described in "Fairplay: a secure two-party computation system", by Malkhi, Nisan, Pinkas, and Sella, *Usenix security symposium 2004*.

The SFDL2.0 compiler is available on the FairplayMP project web site at: `http://www.cs.huji.ac.il/project/Fairplay`.

1 Example

```
/**
* Performs a 2nd price auction between 4 bidders.
* At the end only the winning bidder and the seller know the identity of the winner
* Everyone knows the 2nd highest price,
* Nothing else is known to anyone.
**/
program SecondPriceAuction{
  const nBidders = 4;
  type Bid = Int<4>; // enough bits to represent a small bid.
  type WinningBidder = Int<3>; // enough bits to represent a winner (nBitters bits).

  type SellerOutput = struct{WinningBidder winner, Bid winningPrice};
  type Seller = struct{SellerOutput output}; // Seller has no input

  type BidderOutput = struct{Boolean win, Bid winningPrice};
  type Bidder = struct{Bid input, BidderOutput output};

  function void main(Seller seller, Bidder[nBidders] bidder){
    var Bid high;
    var Bid second;
    var WinningBidder winner;

    winner = 0;
    high = bidder[0].input;
    second = 0;

    // Making the auction.
    for(i=1 to nBidders-1){
      if(bidder[i].input > high){
        winner = i;
        second = high;
        high = bidder[i].input;
      }
      else {
        if(bidder[i].input > second)
        second = bidder[i].input;
      }
    }

    // Setting the result.
    seller.output.winner = winner;
    seller.output.winningPrice = second;
    for(i=0 to nBidders-1){
      bidder[i].output.win = (winner == i);
      bidder[i].output.winningPrice = second;
    }
  }
}
```

2 General

In this document all the keywords of the languge are in **Boldface** notation.

2.1 Lexical Issues

Identifiers have unlimited length, can contain letters, digits, and the underscore characters. They must start with a letter or an underscore and are case sensitive. Identifiers are separated by any amount of whitespace: space, newline, and comments. The comment formats recognized are:

```
// comments to end of line
/* comment to closing */
```

2.2 Program Structure

```
program-name {
    header-declarations
    function-definitions
}
```

In the header, the programmer defines the constants, data types, and global variables that he will use. Beyond the usual usage of types, types defining the input and output of each player must be defined.

In the function definitions, the programmer defines a sequence of functions. The main functionality of the program is the evaluation of the last function defined. This function must be called main and receives as its parameters the list of players.

3 Program Header

3.1 Constants

Constant definitions may appear in the header. The syntax is simple, e.g.,

```
const numberOfBits = 16;
```

The right hand side may be any expression involving only compile-time constants.

3.2 Types

Data types can be defined using the type command. Here are the supported data types:

1. Boolean: false or true.
2. Integer types of any fixed number of bits. Every integer declaration needs a <L>, where L is the length of the integer in bits. Signed 2s complement integers are always used, thus for example 5-bit integers, defined as "Int<5>", can hold the range (-16 .. +15).
3. Structures.
4. Arrays: single dimensional, indices always start at 0. Thus an array defined as "Boolean[4]" has 4 Boolean entries, indexed 0 .. 3.

New data types can be defined using the type statement, and variables can be defined to be of any type. Here are examples of type declarations:

```
type Short = Int<16>;
type Byte = Boolean[8];
type Color = Int<8.bitSize>; // enough bits to hold 8 colors in 2s complement (5 bits).
type Pixel = struct{Color color, Int<10>[2] coordinates};
```

3.3 Player Types

The type declarations must define for each player involved in the computation exactly its input and output. This is done by defining for each player a type that holds its input and output fields. The type is defined as a struct that contains at least one of the fields input and output, of the appropriate type for the player. E.g. if a voter in some protocol needs to input his choice of candidate (in the range 0..5) and gets as output the number of votes of each candidate (which, for every candidate, fit in 10 bits, say), then one would define:

```
const n = 6;
type Voter = struct {Int<n.bitSize> input; Int<10>[n] output};
```

Such declared types can then used in the header of the main function of the program:

```
function void main (Voter voters[80]) {
    ...
}
```

3.4 Global Variables

Static variables with global scope can be declared after the type declarations. The format is standard:

```
var type var name;
```

For example:

```
var Boolean win;
var Int<10> xCoord; var Int<10> yCoord;
var Color[8] palette;
```

All variables are initialized to 0. Declarations of integer bit-size and array size must specify a compile-time constant as the size.

4 Function Declarations

After the global variable declarations comes a sequence of function declarations. Each function has the following structure:

```
function return_type function name(type arg-name, type arg-name, ... ){
    type-declarations
    local-variable-declarations
    function-body
}
```

In function headers, the types of parameters may be generic allowing integer types or array types of any length. E.g.:

```
function Boolean majority (Boolean[] votes) {
  Int<votes.length.bitSize> yes;
  Int<votes.length.bitSize> no;
  for (j = 0 to votes.length-1) {
    if (votes[j])
      yes = yes + 1;
    else
      no = no + 1;
  }
  majority = (yes>no);
}
```

The return_type may be void denoting that no value is returned. The return_type may also be specified as generic - see below for exact syntax and semantics.

4.1 Header

The function header defines the number of parameters of the function, their types, and the return data type. After the header come type declaration and local variable declarations which are similar to their definition above (for the general program).

All parameters are passed by value, including arrays.

4.2 Commands

1. Assignment:

   ```
   lvalue = expression;
   ```

   The lvalue can be a variable, a field of a struct (using x.y notation) or an array entry (using x[exp] notation, where exp is an arbitrary expression). The expression on the RHS must be of the same type as the lvalue (Boolean can be treated also as 1 bit integer). In case of a struct or an array a shallow copy will be made. If the length in bits of the RHS is longer than the lvalue only the first lvalue length bits (starting from the lsb) will be copied. If the length in bits of the RHS is shorter than the lvalue zero padding and sign-extension will be used for widening.

2. If (if/else):

   ```
   if (Boolean expression) statement
   ```

   or

```
if (Boolean expression) statement
else statement
```

3. For:
   ```
   for (index=low_val_expression to high_val_expression) statement
   ```
   The index of the "for" loop is a variable which must not have been previously defined. The low and high ranges of the "for" loop must be compile-time constants.

4. block:
   ```
   { statement ; ... ; statement }
   ```

5. Calling a previously defined void function. No recursion is allowed.

6. There is no return command. Function values are returned Pascal-style, by assigning a value to a variable with the function's name. E.g.:
   ```
   function int<8> triple (Int<8> x) {triple = x + x + x;}
   ```

4.3 Expressions

Expressions are used for computing values. They are used in assignment statements, to denote conditions, to send arguments to functions, etc. Expressions are built from atomic values using operations.

4.3.1 Atomic Values

The following are the atomic values allowed:

1. A Boolean constant: false, true.
2. An integer constant: e.g. 34,-56, 0, 123456789123456789.
3. A (global or local) previously defined variable name or constant name in scope: e.g. i, price.
4. A field in a struct using x.y notation. (Here x is a struct, and y is a name of a field defined in that struct.)
5. An array entry using a[i] notation. (Here a is an array and i is an integer expression.)
6. Length of an array using a.length notation.
7. Size (in bits) of any integer using i.bitSize notation.

Note: the index in an array entry or a bit of an integer need not be a compile time constant but rather may be an arbitrary expression. Note however that there is an enormous difference in efficiency between these two cases: access to an index that is a compile-time constant incurs no cost at all, while access to an index that is a run-time expression incurs cost that is linear in the total size of the array or integer.

4.3.2 Operators

The following operators are defined:

1. +,- : Addition and subtraction (in 2's complement). Accepts k-bit long integers and return a k-bit long result.
2. &&, ||, ! : Logical and , or , not. Accept k-bit long arguments and return k-bit results.
3. $<, >, ==, >=, <=, !=$: 2's complement comparison operators. Accept k-bit long arguments and return a 1-bit result.
4. Function call: e.g. f(x, y), where f is a previously defined function and x, y, .. are arbitrary expressions.
5. *, / : Multiplication and division. Multiplication takes operands of lengths k1 and k2 and returns a result of length k1+k2; division returns a result of length k1 (the numerator size).

Note: Narrow and wide operands may be combined in an operation, and the narrower value is always widened using sign-extension.

4.3.3 Compile-time constants

An expression is a compile time constant if it does not use any variables. Indexes of "for" loops are compile time constants, as are x.bitSize and y.length. Compile-time constants must be used as the range in for loops, and as the integer-size and array-length in declarations. In all cases they are pre-evaluated and are more efficient than general expressions.

4.4 Generic functions

In some cases we want that the return value of a function will be depend on the function parameters. For example we would like to write a function that shifts any int, regardless of its size. This means that the return value is depended on the given input (for int<4> it should return int<4> and for int<8> it should return int<8>). In order to allow such declaration of return values using the bit size or the length of the function parameters, one may use a variable-like place holder for the size or length instead of the return type. In this case the keyword `generic` is used as the type, and the type itself is given in Pascal-style after the function name and parameters.

```
function generic shiftRight (Int<> v) : Int<v.bitSize> {
  for (i = 0 to v.bitSize-2)
    shiftRight[i] = v[i+1];
  shiftRight[v.bitSize-1] = 0;
}
```

5 More examples

```
/**
 * Performs a voting between two candidates.
 * At the end all the voters know who won.
 * Nothing else is known to anyone.
 **/
program voting{
  const nVoters = 5;
  type VotesCount = Int<4>; // Enough bits to vount up to 5 voters.

  // Two candidates (in two's compliment).
  type Vote = Int<2>;
  type Voter = struct{Vote input, Vote output};

  function void main(Voter[nVoters] voters){
    var VotesCount[2] vc;
    var Vote win;

    // Making the voting.
    for(i=0 to nVoters-1){
      if(voters[i].input == 0)
        vc[0] = vc[0] + 1;
      else
        vc[1] = vc[1] + 1;
    }

    if (vc[1] > vc[0])
      win = 1;

    // Setting the result.
    for(i=0 to nVoters-1)
      voters[i].output = win;
   }
}
```

More examples can be found in the Fairplay project web site.