SLXS Language Manual

Contents

1 Introduction

The SLXS (SubLeqXorShr) architecture is an OISC (One Instruction Set Computer),that was developed for cryptographic applications in order to protect the implemented algorithm against SCA (Side-Channel Attacks).
This document contains the definition of the SLXS language on an assembly like format and the SLXS tool which is used as complete tool to compile and run programs on the SLXS CPU. It also contains the macro assembler that translates the language to memory addresses, an intel hex converter and the SLXS simulator with some additional features.

1.1 SLXS Architecture Definition

The SLXS is a Turing-complete machine and supports only a single instruction. It has four parameters (a,b,c,d), and the result is stored as the b parameter.The definition of the model follows:

$$slxsa, b, c, d$$

$$D = *b - *a,\ C = D \veebar c,\ T = C >> 1$$

$$If\ MSB(d) = 0 : *b = C\ else\ if\ MSB(d) = 1 : *b = T$$

$$If\ D <= 0 : goto\ d\ else : goto\ PC + 1$$

2 SLXS Language

The SLXS is an assembly like language that was created as an intermediate language to facilitate communication with the SLXS simulator. Some characteristics of the language are:

- It is a case sensitive programming language, with each line representing one operation (variable/ instruction/comment).
- Indentation is also not used, in the examples below it is used to make the code more readable.
- Empty lines of code can be used and are discarded as comments.
- The length of each variable is 17 bit long.

In the subsections the 3 types of operation are introduced.

2.1 Variables

Variables are the memory addresses that contain the values of the program.
The language recognizes decimal and hexadecimal numbers. The decimal system is the default and the hexadecimal numbers must start with the prefix "0x" or "0X".
The variables stored have a length of 17 bits, with the 17th bit used to differentiate between positive and negative values, with wider numbers capped to 17 bits. Examples of positive numbers are: 1, 0, 0x0ffff and examples of negative numbers are: -1, 0X1ffff
There are four ways to initialize a variable the first is the *Simple*, that initializes one single variable. The *Lists* that initializes an array of variables in one line, the *Assignments* that initializes the variable with the value of another variable. The final is the *Pointer* that stores, as a variable, the address of another variable.

2.1.1 Simple

The regular expression of the simple variable is:

$$([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{:\}([0..9] + |(\{0x\}|\{0X\})[0..9] * [A..F] * [a..f]*)\{\$\}$$

The language recognizes system variables that start with the underscore symbol and they cannot be defined by the programmer, only by the language itself.

$$\{\_\}([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{:\}([0..9] + |(\{0x\}|\{0X\})[0..9] * [A..F] * [a..f]*)\{\$\}$$

Examples:

```
Z:0
temp1:0xaB
```

2.1.2 Lists

The regular expression of the list is:
$([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{: [\}([0..9] + |(\{0x\}|\{0X\})[0..9] * [A..F] * [a..f]*)(\{,\}([0..9] + |(\{0x\}|\{0X\})[0..9] * [A..F] * [a..f]*)) * \{]\$\}$
Example:

```
temp: [0,1,2]
```

The list will be stored in the memory with a pointer at the start of the list and then the values will follow. Each list variable can be accessed with the following pattern:

1. The values of the list can be accessed with the name of the list and the $"(" + number + ")"$. In the previous list the variable temp(1) equals with 1.

2. The list address is stored with the name of the variable given by the programmer. In the temp example the variable temp is equal with the address of temp(0)

An example of the previous list stored in the memory is:

```
Address Values
0004    00005   00000   00001    00002
        temp    temp(0) temp(1)  temp(2)
```

2.1.3 Assignment

The regular expression of the assignment is:

$$([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{:\}([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{\$\}$$

If a variable was previously declared it can be used to initialize another. In the example that follows the $var1$ holds the value 0. The $var1$ can then be used to initialize the $var2$ with the same value.

```
var1 : 0
var2 : var1 //The var2  will hold the value 0.
```

2.1.4 Pointers

The regular expression of the pointer is:

$$\{*\}([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{:\}([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{\$\}$$

If a variable was previously declared it can be used to store its address as another variable. The $var1$ is stored in the address 5 and holds the value 10. By using the star symbol, the $*var2$ will be initialized with the address of $var1$ (5).

```
var1 : 10   // stored in address 5
*var2: var1 // var2 = 5
var3 :var1  // var3 = 10
```

2.2 Instructions

Each instruction has four parameters (a,b,c,d) in the SLXS language and are separated by "," with the end of the instruction denoted by ";". Each instruction uses the variables that have been declared. The parameters a and c can alternatively use decimal numbers or hexadecimal numbers instead of variables. An example is given in the simple instruction.

The regular expressions that follow will help minimize the length of the different types of instructions.

$$number = ([0..9] + |(\{0x\}|\{0X\})[0..9] * [A..F] * [a..f]*)$$
$$var = \{\_\} * ([A..Z][a..z]) + ([A..Z][a..z][0..9])*$$
$$label = \{\_\} * ([A..Z][a..z]) + ([A..Z][a..z][0..9])*$$
$$list\_value = ([A..Z][a..z]) + ([A..Z][a..z][0..9]) * \{(\}number\{)\}$$

2.2.1 Simple

The regular expression of a simple instruction is:

$$var|number|list\_value\{,\}var|list\_value\{,\}var|number|list\_value\{;\}$$

A simple instruction has three variables. If the programmer wants the program to continue the regular flow to the next instruction they can leave the fourth parameter blank and the compiler will add the address of the next instruction to the fourth parameter. Example code of a move command is:

```
Z:0
x:16
y:0
Z,y,x;
```

The previous example can be written with undefined variables. The parameter b (in the example this is denoted as the variable y) must be a variable that was declared by the programmer because the result of the instruction will be saved there.

```
y:0
0,y,0x10;
```

The current version of the tool can store in the memory a fixed number of undefined variables. The same example that saves the result of the instruction on a list is:

```
Z:0
x:16
temp: [0,1,2]
Z,temp(0),x;
```

2.2.2 Functions

The regular expression of a function is:

$$label\{:\}var|number|list\_value\{,\}var|list\_value\{,\}var|number|list\_value\{;\}$$

The use of a label at the start of the instruction can be used to create functions and loops. The address of the specific instruction is saved with the label name. An example on how a function is used is:

```
foo:Z,x,y
```

2.2.3 Jumps

The regular expression of a jump is:

$$var|number|list\_value\{,\}var|list\_value\{,\}var|number|list\_value\{,\}label\{;\}$$

The jump instruction uses all four parameters that were introduced by the SLXS model. The fourth parameter is a label that contains the address of the function we want to jump to. The example uses the previous function foo.

```
Z,x,y,foo;
```

2.2.4 Functions with Jump

The regular expression of a function with a jump is:

$$label\{:\}var|number\{,\}var\{,\}var|number\{,\}label\{;\}$$

This type of instruction combines the two previous types. An example is:

```
foo:Z,x,y,foo1;
```

2.2.5 Labels as Pointers

A label at the start of the instruction can be used to define the address of a current memory position. This is useful when the code must change dynamically in a loop. In the example that follows the foo(0) holds the address of x.

```
foo:Z,x,y;
Z,foo(0),y; //foo(0) holds the address of x.
```

2.2.6 Shifts

For each of the previous types the string "_SH" can be appended as a last parameter to create the shift command. An example of the simple instruction is the following: Regular Expression: $var\{,\}var\{,\}var\{,\}\{\_SH;\}$

```
z,x,y,_SH;
/*With function and a jump:*/
foo:x,y,z,foo1,_SH;
```

2.3 Comments

The language recognizes two types of comments that are both in-line comments. The first is the double dash "//" and the language ignores everything until it finds a newline. The second type is the boxed comments that start with "/*" and end with "*/" and the language ignores everything inside them.

1. **Whole line comments**
   Regular Expression:$\{//\}[\hat{}\,]*\$$
   Example:

   ```
   //I am a comment
   ```

2. **Boxed comments**
   Regular Expression: $\{/*\}[\hat{}\,]*\{*/\}$
   Example:

   ```
   S :/* I am a comment */ 10
   x,y,temp(0)/*first variable inside the list temp*/,_SH;
   ```

3 Predefined Variables/labels/Instructions

The SLXS language has some predefined variables, labels and instructions that are listed below. It can be updated with additional material in the future. Variables and Labels used by the system start with underscore for consistency.
Variables:
_zero: 0
The _zero variable is initialized by the micro assembler. It can also be used by the programmer without implicitly declaring it.

Labels: *_main:*
It is the main function of the program. Each program must have one otherwise it cannot run.
_SH:
The _SH appended at the end of the instruction ensures the assembler knows to execute the shift command.

Instructions:
_zero,_zero,_zero,_main;
This is the first line of code in every program. It is used for the simulator to know where the

_main function is. The assembler always adds it to the start of the code, so the programmer does not have to.
loop:_zero,_zero,_zero,loop;
This instruction indicates the end of the program. It is needed so that the SLXS simulator knows when to stop. It has to be added by the programmer.

3.1 Operations

To define the basic operations we need the following variables to be declared.

```
/*Variables:*/
  Z:   00000
  M:   1FFFF //(-1)
  U:   00001
  Label:  address // A 16 bit address (the 17 bit is for the shift)
  var:10
  var_r: -10
```

Commands:

```
 CLEAR : var  , var, Z;          // var = 0
 JUMP  : Z    , Z  , Z, Label;   // Goto Label
 MOV   : var  , var, var1;       // var = var1
 LOOP  : Z    , Z  , Z  ,Label;  // current address| EOP
 RSHIFT: Z    , var, Z  ,_SH;    // var >> 1
 LSHIFT: var_r, var, Z;          // var = var + var = var << 1
 XOR   : Z    , var, var1;       // var = var ^ var1
 ADD   : M    , var, Z;          // var = var + 1
 SUB   : U    , var, Z;          // var = var - 1
```

The multiply operation can be implemented by a sequence of additions and the division by a sequence of subtractions. The most common multiplications and divisions are computed with the number two as a multiplier/divisor, they can be implemented with the use of two shift operations.
The most complex operations are the AND/OR, due to the need of a sequence of instructions to implement them. Based on the formula $x+y=(x \veebar y)+2(x\&y)$ and $x+y=2(x|y)-(x \veebar y)$. We can extract the two operations:

1. $x\&y=(x+y+(x \veebar y))/2$
2. $x|y=(x+y-(x \veebar y))/2$

The two operations can be translated to the SLXS language:

```
/*Extra Variables:*/
   res = 00000
   xor = 00000
   y,x

AND: res, res, x;     // res = x
     y  , my , Z;     // my  = -y
     my , res, Z;     // res = x + y
     xor, xor, x;     // xor = x
     Z  , xor, y;     // xor = x ^ y
     xor, res, Z,SH;  res = (res - xor ) >> 1

OR : res, res, x;     //res = x
     y  , my , Z;     // my  = -y
     my , res, Z;     // res = x + y
     xor, xor, x;     // xor = x
     Z  , xor, y;     // xor = x ^ y
     xor, mxr, Z;     // mxr = -xor
     mxr, res, Z,SH;  // res = (res + xor) >> 1
```

4 SLXS tool

The SLXS code must go through three stages to provide the end result. At first the code is translated into memory addresses. Next those memory addresses must be converted to an Intel hex file format, one for each of the four parameters of the SLXS language. The final stage is the emulator that takes the four files and outputs the result of the code.
The SLXS automatizes this process with a set of tools that produce output files for each part of the three stages. The tools can be used separately, if the user needs to use only one of them. The five options that the tool provides are:

1. $[-h|--help]$ prints the help menu.
2. $[-i|--input]$ takes a file with SLSX code as an argument and creates an output file with the memory addresses.
3. $[-f|--intel\_hex]$ creates four files in hex_intel format that can be used in the simulator.
4. $[-s|--simulator]$ It simulates an SLXS processor. The four hex files created by the -f option are used.
5. $[-p]$ Prints the variables before and after the emulator runs. All the previous options must be enabled. If the variable does not exist nothing is displayed.

5 Code Examples

5.1 Code example for OR:

This example shows how the OR operation is implemented in the SLXS language. The tool can be used with the command:

```
python slxs.py -i or.txt -f -s -p res
```

That will execute all three stages and print the Variable res before the execution and after.

```
/*Variables*/
Z   : 0          // zero value
mxr : 0          //-xor
```

```
One : 1
xor : 0
res : 0
x   : 0x11
y   : 0x10
my  : 0         // -y

/*main function*/
_main:res, res, x;
    y  , my , Z;
    my , res, Z;
    xor, xor, x;
    Z  , xor, y;
    xor, mxr,Z;
    mxr, res,Z,SH;
    loop,Z,Z,Z,loop;
```

An example of undefined variables on the previous code is:

```
    0x10 , my , 0;
    my , res, 0;
    xor, xor, 0x11;
    0  , xor, y;
    xor, mxr,Z;
    mxr, res,Z,SH;
```

With the option [-i] the output file with the memory addresses is created. The first column indicates the memory address, the other four are the values stored in the addresses in the block.

```
0000 00004 00004 00004 00010 | _zero,_zero ,_zero,_main | Jump to main
0004 00000 00000 00000 00001 | 0 , 0 , 0 ,1             | _zero:0 Z:0   mxr:0   One:1
0008 00000 00000 00011 00010 | 0 , 0 , 17,16            | xor:0   res:0 x:17  y:16
000c 00000 00000 00000 00000 | 0 , 0 , 0 ,0             | my:0    empty slots
0010 00009 00009 0000a 00014 | res,res,x;               | main function res=x
0014 0000b 0000c 00005 00018 | y  , my , Z;             | my  = - y
0018 0000c 00009 00005 0001c | my , res, Z;             | res = x + y
001c 00008 00008 0000a 00020 | xor, xor, x;             | xor = x
0020 00005 00008 0000b 00024 | Z  , xor, y;             | xor = x ^ y
0024 00008 00006 00005 00028 | xor, mxr,Z;              | mxr = -xor
0028 00006 00009 00005 1002c | mxr, res,Z,SH;           | res = (res + xor ) >> 1
002c 00004 00004 00004 0002c | loop                     | End of program
```

Intel Hex Format Files for simulator:
Each Intel Hex block in the SLXS simulator contains the following data:

1. Start code, one character, an ASCII colon ':'.

2. Byte count, two hex digits, indicating the number of bytes (hex digit pairs) in the data field.

3. Address, four hex digits, representing the 16-bit beginning memory address offset of the data.

4. Record type, two hex digits, 00 to 05,in the implementation the 01 and 03 are the only digits used.

5. Data, six hex digits.

6. Checksum, two hex digits, a computed value that can be used to verify the record has no errors.

The output from the assembler has to be split in four files of intel hex format. An example that was created with the [-f] option for the OR is the following:

```
    mem0.hex             mem1.hex            mem2.hex             mem3.hex
:03000000000004f9 |:03000000000004f9 |:03000000000004f9 |:03000000000010ed
:03000100000000fc |:03000100000000fc |:03000100000000fc |:03000100000001fb
:03000200000000fb |:03000200000000fb |:03000200000011ea |:03000200000010eb
:03000300000000fa |:03000300000000fa |:03000300000000fa |:03000300000000fa
:03000400000009f0 |:03000400000009f0 |:0300040000000aef |:03000400000014e5
:0300050000000bed |:0300050000000cec |:03000500000005f3 |:03000500000018e0
:0300060000000ceb |:03000600000009ee |:03000600000005f2 |:0300060000001cdb
:03000700000008ee |:03000700000008ee |:0300070000000aec |:03000700000020d6
:03000800000005f0 |:03000800000008ed |:0300080000000bea |:03000800000024d1
:03000900000008ec |:03000900000006ee |:03000900000005ef |:03000900000028cc
:03000a00000006ed |:03000a00000009ea |:03000a00000005ee |:03000a0001002cc6
:03000b00000004ee |:03000b00000004ee |:03000b00000004ee |:03000b0000002cc6
:00000001FF       |:00000001FF       |:00000001FF       |:00000001FF
```

The four files can be now imported to the simulator.
The option [-s— –simulator] takes the four intel_hex files and creates the final state of the memory in an output file as shown in the code below. Only the addresses are stored in the output file, the two other columns are for explanation. In the position the res variable was stored the value has changed to 0x10 which is the result of x y.

```
0000 00004 00004 00004 00010 | _zero,_zero ,_zero,_main | Jump to main
0004 00000 00000 00000 00001 | 0 , 0 , 0 ,1             | _zero:0 Z:0   mxr:0   One:1
0008 00000 00010 00011 00010 | 0 , 11 , 17,16           | xor:0   res:16 x:17  y:16
000c 00000 00000 00000 00000 | 0 , 0 , 0 ,0             | my:0    empty slots
0010 00009 00009 0000a 00014 | res,res,x;               | main function res=x
0014 0000b 0000c 00005 00018 | y  , my , Z;             | my  = - y
0018 0000c 00009 00005 0001c | my , res, Z;             | res = x + y
001c 00008 00008 0000a 00020 | xor, xor, x;             | xor = x
0020 00005 00008 0000b 00024 | Z  , xor, y;             | xor = x ^ y
0024 00008 00006 00005 00028 | xor, mxr,Z;              | mxr = -xor
0028 00006 00009 00005 1002c | mxr, res,Z,SH;           | res = (res + xor ) >> 1
002c 00004 00004 00004 0002c | loop                     | End of program
```

5.2 List Example

The following code takes two lists XOR them and saves the result in the first list.

```
/*Variables*/
/* List */
List :  [0x1,0x2,3,4,5,6,7,8,9,10]
List2 : [0x1,0x2,3,4,5,6,7,8,9,10]

Z   : 0
One : 1
M   : -1
counter :10

list_add : List // The value of the variable List not used.
*list_add_10: List(9) //The address of List(9) not used.

/*Code*/
/*main function*/

_main: Z,List(0),List2(0);    //List(n)=List(n) xor List2(n)
       M,_main(1),Z;          // List(n+1)
       M,_main(2),Z;          // List2(n+2)
       One,counter,Z;         // counter--
       0,counter,Z,loop;      // LEQ goto loop
       Z,Z,Z,_main;           // goto _main
       loop: Z,Z,Z,loop;      // End of program
```

The following code translates to:

```
0000 00004 00004 00004 00038    |_zero,_zero,_zero,_main;
0004 00000 00006 00001 00002    |_zero,List_address,List(0),List(1)
0008 00003 00004 00005 00006    |List(2),List(3),List(4),List(5)
000c 00007 00008 00009 0000a    |List(6),List(7),List(8),List(9)
0010 00011 00001 00002 00003    |List2_address,List2(0),List2(1),List2(2)
0014 00004 00005 00006 00007    |List2(3),List2(4),List2(5),List2(6)
0018 00008 00009 0000a 00000    |List2(7),List2(8),List2(9),Z
001c 00001 1ffff 0000a 00006    |One,M,counter,list_add
0020 0000f 00000 00000 00000    |list_add_10
0024 00000 00000 00000 00000
0028 00000 00000 00000 00000    |This empty slots are reserved to hold the values
002c 00000 00000 00000 00000    |of undefined variables.
0030 00000 00000 00000 00000
0034 00000 00000 00000 00000
0038 0001b 00006 00011 0003c    |_main: Z,List(0),List2(0);
003c 0001d 00039 0001b 00040    |       M,_main(1),Z;
0040 0001d 0003a 0001b 00044    |       M,_main(2),Z;
0044 0001c 0001e 0001b 00048    |       One,counter,Z;
```

```
0048 00024 0001e 0001b 00050    |       0,counter,Z,loop;
004c 0001b 0001b 0001b 00038    |       Z,Z,Z,_main;
0050 0001b 0001b 0001b 00050    |       loop: Z,Z,Z,loop;
```

The result of the slxs simulator is:

```
0000 00004 00004 00004 00038    | The List has been xored with he List2 and
0004 00000 00006 00000 00000    | the results where saved in the addresses
0008 00000 00000 00000 00000    | of the List.
000c 00000 00000 00000 00000
0010 00011 00001 00002 00003
0014 00004 00005 00006 00007
0018 00008 00009 0000a 00000
001c 00001 1ffff 00000 00006
0020 0000f 00000 00000 00000
0024 00000 00000 00000 00000
0028 00000 00000 00000 00000
002c 00000 00000 00000 00000
0030 00000 00000 00000 00000
0034 00000 00000 00000 00000
0038 0001b 00010 0001b 0003c
003c 0001d 00039 0001b 00040
0040 0001d 0003a 0001b 00044
0044 0001c 0001e 0001b 00048
0048 00024 0001e 0001b 00050
004c 0001b 0001b 0001b 00038
0050 0001b 0001b 0001b 00050
```