# Security analysis

Example program for buffer overflow:

program

high int x;

low int y;

low int z;

low int k;

read x;

while x!=1 do

y := 1;

x := x - 1;

od

z := z\*k;

write z;

write y; (\* security violation \*)

end

The program presented above is the example program for security analysis. After the while loop, there is a violation of security since high level information is indirectly passed from *x* to *y.* Hence, *y* gets high security level and after the while loop statement *write y* will leak high level information about value of variable *x*. However, statement *write z* does not leak information.

### Security analysis

This subsection addresses the definition of the security analysis for the project language.

### Definition

The security analysis is defined as

where *L* is a complete lattice, is a set of transfer functions, *F* is a finite flow , *E* is a finite set of extremal labels, is an extremal value and is a mapping from labels to transfer functions.

#### Lattice L

The complete lattice *L* is defined as

where *ctx* security level of the context*.*

#### Mapping

The mapping of labels to transfer functions is constructed as

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

#### 

The which determines the signs of expressions is given by

|  |  |  |
| --- | --- | --- |
|  |  | *{low}* |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

where which could be and is specified by

Finally, the which could be and /, and which could only be are defined in the and .

Table 3.1

|  |  |  |
| --- | --- | --- |
|  | low | high |
| low | {low} | {high} |
| high | {high} | {high} |

#### 

is given by

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Finally, the and are defined with the tables below respectively.

Table 3.3

|  |  |  |
| --- | --- | --- |
|  | low | high |
| low | {low} | {high} |
| high | {high} | {high} |

Table 3.4

|  |  |  |
| --- | --- | --- |
| & | low | high |
| low | {low} | {high} |
| high | {high} | {high} |

The algorithms and implementation part are not presented in the report since the security analysis has the same structure as detection of signs analysis with the difference that that security level is always single for any variable. Therefore, these sections are omitted.

### Limitations and discussion regarding improvements

The program for security analysis is implemented in a way that the context security level is remembered before it can be modified in the boolean condition and after the end of the statement containing the conditional test the security context is returned to the level which was before the this conditional statement. This feature provides improvements to precision which are shown in the next section by the execution of our test program. However, we did not come up with theoretical explanation of this feature. That is why it is not expressed in the definition.

Moreover, the security analysis is limited due to implemented advanced feature, thus, nested conditional statements are not handled.

The improvement would be to implement handling of nested conditional statements.

### Benchmarking

### Benchmark 1

This example program is our test program which shows the advantages of our implementation. The program is presented in Code 5.1. Since the security context is reduced to the low level after the exit from the while loop, the security level of the *z* variable is kept low. That is why the only possible information leakage is in *write y* statement, because the security level of *y* is raised to high due to high context.

The detailed output of our analysis is presented in Code 5.2.

program

high int x;

low int y;

low int z;

low int k;

read x;

while x!=1 do

y := 1;

x := x - 1;

od

z := z\*k;

write z;

write y; (\* security violation \*)

end

Code 5.1 Benchmark 1program

Program graph:

(1,read x;,2), (2,x!=1,3), (3,y := 1;,4), (4,x := x-1;,2), (2,!x!=1,5), (5,z := z\*k;,6), (6,write z;,7), (7,write y;,8)

Security Level Analysis solutions table 19:

1: ctx={low} z={low} k={low} y={low} x={high}

2: ctx={high} z={low} k={low} y={high} x={high}

3: ctx={high} z={low} k={low} y={high} x={high}

4: ctx={high} z={low} k={low} y={high} x={high}

5: ctx={low} z={low} k={low} y={high} x={high}

6: ctx={low} z={low} k={low} y={high} x={high}

7: ctx={low} z={low} k={low} y={high} x={high}

8: ctx={low} z={low} k={low} y={high} x={high}

Security level violations:

(7,write y;,8)

Code 5.2 Analysis result of benchmark 1

### Benchmark 2

The example program below is selected to show limitations of our implementation. Our system does not detect possible security violation in the while loop, where a number is output by *write(0)* statement. However the rest 3 security violations are detected. Moreover, the last statement *write(h3)* does not violates the security policy since the security level of the variable *h3* is downgraded by the assignment of the low security level data.

The benchmark is provided by Andrius Andrijauskas and Lars Bonnicshen and presented in Code 5.3. The detailed output of our program is presented in Code 5.4.

program

high int h1;

high int h2;

high int h3;

low int l1;

low int l2;

low int l3;

l3 := l1 + h2; (\* l3 is now HIGH \*)

h3 := 2 + l1; (\* h3 is now LOW \*)

if h2 = 0 then

l2 := 5; (\* l2 is now HIGH \*)

else

l2 := 9; (\* l2 is now HIGH \*)

fi

while h1 != l1 do

write(0); (\* Warn execution depends on HIGH expression \*)

l1 := l1 + 1; (\* l1 is now HIGH \*)

od

write(l2); (\* Warn l2 is HIGH\*)

write(l3); (\* Warn l3 is HIGH\*)

write(h2); (\* Warn h2 is HIGH \*)

write(h3);

end

Code 5.3 Benchmark 1program

Program graph:

(1,l3 := l1+h2;,2), (2,h3 := 2+l1;,3), (3,h2=0,4), (3,!h2=0,5), (4,l2 := 5;,6), (5,l2 := 9;,6), (6,h1!=l1,7), (7,l1 := l1+1;,6), (6,!h1!=l1,8), (8,write l2;,9), (9,write l3;,10), (10,write h2;,11), (11,write h3;,12)

Security level violations:

(9,write l2;,10), (10,write l3;,11), (11,write h2;,12)

Code 5.4 Analysis result of benchmark 1