User Manual for HyLeak v.1.0: Hybrid Analysis Tool for Information Leakage

Fabrizio Biondi[1], Yusuke Kawamoto[2], Axel Legay[3], and Louis-Marie Traonouez[3]

[1] CentraleSupélec Rennes, Rennes, France
[2] AIST, Tsukuba, Japan
[3] Inria Rennes, Rennes, France

Abstract. This is a user manual of the tool HyLeak for reasoning about the quantity of information leakage in programs. The tool takes as input a program code written in a simple imperative language and computes the amount of leaked secret information in the observables outputs of the given program.

The tool HyLeak [1] takes as input a program code written in a simple imperative language (a slight extension of the input language used in the QUAIL tool) and computes its Shannon leakage, i.e., the mutual information between the variables defined as secrets and those as observable outputs in the given program.

Following the theoretical results in Kawamoto et al. [5], HyLeak divides the input program into (terminal) components and decides for each of them whether to analyze it using precise analysis (based on QUAIL [2]) or statistical analysis (using a similar approach to `LeakiEst` [3] and `LeakWatch` [4]), by applying heuristics that evaluate the analysis cost of each component. Then, HyLeak composes the analysis results of all components into an approximate joint probability distribution of the secret and observable variables in the program. Finally, the tool estimates the Shannon leakage and its confidence interval.

1 Getting Started

The source code and examples of HyLeak can be downloaded from https://project.inria.fr/hyleak/files/2017/04/hyleak.tar.gz ($\sim$6.6 MB).

To decompress and compile it:

1. HyLeak is written in Java. Make sure you have the latest version of the Java Runtime Environment installed.
2. Unzip the program by running

   ```
   $ tar zxvf hyleak.tar.gz
   ```

3. Enter the HyLeak directory by running

```
$ cd hyleak/
```

4. Compile HyLeak by running

```
$ make
```

HyLeak is now compiled.

Refer to Section 2 for information on how to use HyLeak, to Section 3 for the input language of the programs analyzed by HyLeak, and to Section 4 for the intermediate files that HyLeak generates.

2 Usage

You can execute HyLeak with an input code `program.hyleak` in a terminal by running:

```
> ./hyleak [arguments] program.hyleak
```

Arguments

HyLeak is based on the QUAIL v 2.0 code, and shares some of its functionalities. The possible arguments for HyLeak are:

`-c | --clean`

Delete the temporary preprocessed file.

`--config=<filename>`

Alternative location for the configuration file.

`-h | -? | --help`

Print the help and terminate.

`-v <level>`

Verbose mode between 0 (minimum) and 4 (maximum). Default value if not specified is 2.

`-verb:<vname>:=<vvalue>`

Set verbosity for location `<vname>` to value `<vvalue>` (advanced users only).

`--verb-locations`

Print verbosity locations and terminate.

`-th <number>`

Sets the number of parallel threads HyLeak should use. Default is the number returned by Runtime.getRuntime().availableProcessors(), minimum 1.

`-p <value>`

Define computation precision as a number of digits. Default value if not specified is 15.

`--const:<cname>:=<cvalue>`

Replace the value of a constant `<cname>` in the model by the value `<cvalue>`.

`--preprocess`

Run only the preprocessor and terminate.

`--no-preprocess`

Do not run the preprocessor (advanced users only; input file must be in if-goto format).

`-i <iterations>`

Sets the total number of simulations to use on all components. Default: 50000.

`-t <time_limit>`

Sets the time limit (in milliseconds) for the simulation. A value of 0 means no time limit. Default: 0 (meaning no time limit).

`--no-cfg`

By default and if the dot utility is installed, HyLeak draws the Control Flow Graph (CFG) of the program to be analyzed as a file cfg.png. This option prevents HyLeak from drawing the CFG of the program.

3 HyLeak Language

The input language for programs to be analyzed by HyLeak is almost identical to the input language of the QUAIL tool [2] that HyLeak is based on. The only additions are the bounded FOR loops described in Section 3.10 and the Simulate statements described in Section 3.11. See Appendix A for the full syntax of the language.

3.1 Comments

Comments in the program starts with `//`. Then the rest of the line is ignored.

3.2 Variable declarations

All variables in HyLeak are fixed sized integers. They are declared at the beginning of the program. Constants can be declared in the following manner:

```
const N := 4;
```

They are replaced by their value during the preprocessing step.

Public variables are either public or observable. In the latter case the attacker will be able to distinguish their value. They are declared in the following manner:

`public int4 var;` or `observable int4 var;`

declares a 4 bits integer variable whose name is var, either public or observable.

```
public int4 var := 5;
```

declares `var` and initializes it to value 5. Any expression can be used to initialized a variable, provided that the variables used in the expression are public or constants and have been previously declared. Variables not initialized are implicitly initialized to the value 0.

Private variables are either private or secret. The attacker will only infer knowledge on the latter. They are declared in the following manner:

`private int4 var;` or `secret int4 var;`

declares a 4 bits integer variable whose name is var, either private or secret.

```
private int4 var := [0,1][2,5];
```

declares `var` and restricts its range to the two intervals [0,1] and [2,5]. Again any expression can be used in the bounds of the intervals.

3.3 Arrays

Variables can also be arrays of integers and multi-dimensional arrays. Arrays are declared in addition to the integer type of a variable.

```
public array[4] of int4 tab;
```

declares a public variable `tab` that is an array of 4 bits integer of size 4 whose indices range from 0 to 3, while

```
public array[1..4] of int4 tab;
```

declares `tab` as an array of size 4 whose indices range from 1 to 4. The size of an array can be any expression that evaluates to an integer.

Arrays are replaced during the preprocessing. Therefore, an array variable named `tab`, whose indices range from 0 to 3, declares 4 variables, whose name are `tab[0]`, `tab[1]`, `tab[2]` and `tab[3]`. They have the same publicity and the same integer type as the array.

An array may be initialized with a set of initial values:

```
public array[1..4] of int4 tab := {1,1,2,2};
```

initializes `tab` such that `tab[1]` and `tab[2]` are equal to 1, while `tab[3]` and `tab[4]` are equal to 2. Private arrays can be initialized like any private variable, with a set of intervals:

```
private array[1..4] of int4 tab := [0,1];
```

In that case all the variables in the array are initialized to the same range of integers.

3.4 Expressions

Expressions are used in guards, assignments, variables initialization and arrays indices. Binary operators (`||`,`&&`,`^`,`+`,`-`,`*`,`/` and `%`) and unary operators (`-`,`!`) can used. Classical operators precedence is assumed. For boolean operations integer variables are considered as a true value if non null, and false if null. Only public variables, constants and integers can be used in expressions.

3.5 Guards

Guards are limited to a single comparison between a variable on the left side (either public, or private, or constant, or an integer value) and an expression on the right side. Any comparison operator among `<`,`>`,`<=`,`>=`,`==` and `!=` can be used.

3.6 Assignments

An assignment statement is written in the following manner:

`assign var :=` *`expr`*`;`

where `var` is a public variable (possibly with indices) and *`expr`* is an expression containing no private variables.

3.7 Random assignments

The program can used two types of random primitives to assign values to a variable.

`random var := random(`*`expr_min`*`,`*`expr_max`*`);`

assigns to a public variable `var` a random value, chosen between the values of *`expr_min`* and *`expr_max`*, with a uniform probability distribution.

`random var := randombit(p);`

where `p` is a float value lower than 1, assigns to a public variable `var` a random bit value, that is 0 with probability `p`, and 1 with probability $1 - \texttt{p}$.

3.8 IF statements

IF conditional statements starts with the keyword `if`, possibly followed by `elif` and `else`, and ends with `fi`. The consequent statements are listed after the keyword `then`. For example the following structures are allowed:

```
if (h <= l) then assign var:=1;
fi
```

```
if (h <= l) then assign var:=1;
else assign var:=2;
assign var:=var+1;
fi
```

```
if (h <= l) then assign var:=1;
elif (h==l) then assign var:=2;
```

```
fi
if (h <= l) then assign var:=1;
elif (h==l) then assign var:=2;
elif (h==l+1) then assign var:=2;
else assign var:=2;
fi
```

3.9 WHILE statements

Conditional WHILE loop starts with the keyword `while`, followed by a guard, and the statements included in the loop are listed between the keywords `do` and `od`. For example the following structure is allowed:

```
while (h <= l) do
assign l := 1;
assign var := 2;
od
```

3.10 FOR statements

A FOR loop can be used to browse all the elements of an array. The syntax is:

```
for (var in tab) do
assign var := var+1;
od
```

The variable `var` is a local variable that must only be used inside the loop. It will take successively each value in the array `tab`. Note that if `tab` is a multi-dimensional array `var` is also an array.

A bounded FOR loop is also available to loop over commands while a variable takes values from an interval. The syntax is:

```
for (var in interv) do
assign var := var+1;
od
```

The variable `var` is a local variable that must only be used inside the loop. It will take successively each value in the interval `interv`. For instance, writing `for (var in [0,i-1])` will assign to variable `var` all values from 0 to the value of variable `i` minus 1.

3.11 Simulate statements

A `simulate` statement indicates to the tool that at this point of the program the precise analysis has to halt and statistical simulation has to be started instead. The syntax is:

```
simulate;
```

The `simulate-abs` is similarly use to halt precise analysis and start abstraction-then-sampling analysis, as described in Section 4.3 of [5]. The syntax is:

```
simulate-abs;
```

If no `simulate` nor `simulate-abs` is introduced in the input program by the user, the tool decides heuristically which parts of the input program to analyze

with statistical simulation and with abstraction-then-sampling and inserts the statements accordingly, as described in Section 2 of [1].

3.12 Return statements

The program ends when a return statement is reached. Its syntax is simply:

```
return;
```

4 Intermediate Files

When you execute HyLeak with an input program `program.hyleak`, the preprocessor of HyLeak generates the following intermediate files:

```
program.hyleak.1.pp
program.hyleak.2.pp
program.hyleak.2.values
program.hyleak.3.pp
program.hyleak.4.pp
```

The meaning of each file is:

—.1.pp: constants removed, expressions reduced, checks performed
—.2.pp: loop unrolled and arrays expanded
—.2.values: information on rough ranges of variables
—.3.pp: annotate by heuristics
—.4.pp: final `if-goto` code

Details of the `if-goto` language is explained in Appendix B.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number JP17K12667, by JSPS and Inria under the Japan-France AYAME Program, by the MSR-Inria Joint Research Center, by the Sensation European grant, and by région Bretagne.

References

1. F. Biondi, Y. Kawamoto, A. Legay, and L. Traonouez. Hyleak: Hybrid analysis tool for information leakage. In *Proc. of ATVA'17*, pages 156–163, 2017.
2. F. Biondi, A. Legay, L. Traonouez, and A. Wasowski. QUAIL: A quantitative security analyzer for imperative code. In *Proc. of CAV'13*, pages 702–707, 2013.
3. T. Chothia, Y. Kawamoto, and C. Novakovic. A tool for estimating information leakage. In *Proc. of CAV'13*, pages 690–695, 2013.
4. T. Chothia, Y. Kawamoto, and C. Novakovic. Leakwatch: Estimating information leakage from java programs. In *Proc. of ESORICS'14, Part II*, pages 219–236, 2014.
5. Y. Kawamoto, F. Biondi, and A. Legay. Hybrid statistical estimation of mutual information for quantifying information flow. In *Proc. of FM'16*, pages 406–425, 2016.

A Syntax of HyLeak Language

Finally we present the syntax of HyLeak Language.

```
program := (declaration)* (statement)*
declaration := const_declaration | public_declaration | private_declaration
declaration := 'const' IDENTIFIER ':=' expression ';'
public_declaration := public_type var_type IDENTIFIER ';'
                    | public_type var_type IDENTIFIER ':=' init_val ';'
public_type := 'public' | 'observable'
private_declaration := private_type var_type IDENTIFIER ';'
                     | private_type var_type IDENTIFIER ':=' (init_interval)+ ';'
private_type := 'private' | 'secret'
var_type := INT | 'array[' expression ('..' expression)? ']' 'of' var_type
init_val := expression | '{' number_list '}'
number_list := expression (',' expression)*
init_interval := '[' expression ',' expression ']'
statement := assignment_st | return_st | random_st | while_st | for_st
           | if_st | simulate_st
assignment_st := 'assign' public_variable ':=' expression ';'
return_st := 'return' ';'
random_st := 'random' public_variable ':=' 'randombit' '(' FLOAT ')' ';'
           | 'random' public_variable ':=' 'random' '(' expression ',' expression ')' ';
while_st := 'while' '(' guard ')' 'do' (statement)* 'od'
for_st := 'for' '(' IDENTIFIER 'in' IDENTIFIER indices ')'
          'do' (statement)* 'od'
        | 'for' '(' IDENTIFIER 'in' '[' IDENTIFIER ',' IDENTIFIER ']' ')'
          'do' (statement)* 'od'
if_st := if_case (elif_case)* (else_case)? 'fi'
simulate_st := 'simulate'|'simulate-abs'
if_case := 'if' '(' guard ')' 'then' (statement)*
else_case := 'else' (statement)*
elif_case := 'elif' '(' guard ')' 'then' (statement)*
guard := leaf_term comp_operator expression
comp_operator := '<=' | '>=' | '>' | '<' | '==' | '!='
expression := expression binary_operator expression | unary_expression
binary_operator := '||'|'&&'|'^'|'+'|'-'|'*'|'/'|'%'
unary_expression := unary_operator term | term
unary_operator := '!'|'-'
term := '(' expression ')' | public_variable | NUMBER
leaf_term := NUMBER | IDENTIFIER (index)*
public_variable := IDENTIFIER (index)*
index := '[' expression ']'
```

The lexical terms of the grammar are:

```
INT := 'int''0'..'9'+
IDENTIFIER := ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
NUMBER := '0'..'9'+
FLOAT := '0''.'('0'..'9')+
```

The keywords of the language are: `const`, `public`, `observable`, `private`, `secret`, `array`, `of`, `int`, `return`, `assign`, `random`, `randombit`, `if`, `elif`, `else`, `fi`, `then`, `while`, `for`, `do`, `od`, `simulate`, `simulate_abs`.

B `if-goto` Language

The analyzer takes as input a program written in a simplified lower level language. Briefly, the differences are the following:

- It has no array and no constant variable.
- All the terms in expressions must be enclosed within brackets.
- It has only if conditional (no while and no for), without elif case, that are not closed with fi.
- And additional goto statement is used whose syntax is:

  ```
  goto line;
  ```

  where line is a line number in the program.
- if and else consequences are limited to a single goto statement.
- The program has no comments and no blank line.

The preprocessor compiles automatically a HyLeak program to this language.