HyLeak: Hybrid Analysis Tool for Information Leakage*

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

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

Abstract. We present HyLeak, a tool for reasoning about the quantity of information leakage in programs. The tool takes as input the source code of a program and analyzes it to estimate the amount of leaked information measured by mutual information. The leakage estimation is mainly based on a hybrid method that combines precise program analysis with statistical analysis using stochastic program simulation. This way, the tool combines the best of both symbolic and randomized techniques to provide more accurate estimates with cheaper analysis, in comparison with the previous tools using one of the analysis methods alone. HyLeak is publicly available and is able to evaluate the information leakage of randomized programs, even when the secret domain is large. We demonstrate with examples that HyLeaks has the best performance among the tools that are able to analyze randomized programs with similarly high precision of estimates.

1 Introduction

Automated security evaluation. With the increasing complexity of networked systems, it is getting harder and harder for security engineers to analyze a system and give a reasonable guarantee that the system does not jeopardize the security and privacy of the users. A significant effort in research has been devoted towards techniques able to (semi-)automatically identify leakage of confidential information in software and hardware systems, allowing for more formal assurances of security and privacy.

Among automated techniques to quantify the information leakage of a system, we distinguish the two approaches: *precise program analysis* providing a precise result (e.g. [1, 3, 4]) and *statistical analysis* providing an approximate estimation (e.g. [5, 6, 7]). The main difference between them is that precise analysis needs to explore the complete behavior of the system to obtain the exact leakage values, while statistical analysis has to cover only a statistically significant sample of the system's behavior to produce their estimation, and thus tends to scale better. However, when statistical analysis fails to cover many rare events in the system, it does not produce an accurate estimation.

Recently, some researchers have been trying to bridge the gap between precise and statistical techniques by introducing *hybrid methods* combining them [9, 11]. This paper

* 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.

presents the HyLeak tool, the first publicly available leakage computation tool leveraging both precise and statistical analyses. The implementation is based on the hybrid method for estimating mutual information [9] while it also employs many optimization techniques to enhance the estimation performance. As we explain in Section 4 the tool has the best performance in computing leakage among the tools that are able to analyze randomized programs with similarly high precision of estimates.

The HyLeak hybrid analysis strategy. The HyLeak tool takes as input a program written in a simple imperative language (a slight extension of the input language used in the QUAIL tool [3]) and computes its Shannon leakage, i.e., the mutual information between the variables defined as *secrets* and those as *observable outputs* in the given source code.

More specifically, HyLeak divides a program code into (terminal) components and decides for each of them whether to analyze it using precise or statistical analysis, by applying heuristics that evaluate the analysis cost of each component. Then, following the theoretical results in [9], 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.

One of HyLeak's technical novelties lies in the implementation of the code decomposition. The procedure is based on the fact that the cost of analyzing a code fragment with precise analysis is proportional to the amount of traces in the fragment (since precise analysis has to analyze all traces), while the cost of statistical analysis is proportional to the number of possible observable variables (since statistical analysis has to run simulations for each value of the observable variables). Then the amount of traces and observable variable values at each point in the program execution is statically estimated via a heuristic approach. The tool locates in which part of the execution, if any, stochastic simulation becomes more efficient than precise analysis, and records this by inserting a `simulate` statement in the code.

The stochastic simulation usually has to run the code for each value of the secret; however, if in the code fragment the observable variables do not depend on the secret, it is sufficient to simulate the fragment with a single secret value and to apply the results to all the other values of the secret. This technique is called *abstraction-then-sampling* in [9], and can significantly reduce the number of simulations necessary to produce a good estimation, particularly for programs with a large secret domain. If the tool finds it can apply abstraction-then-sampling on the code fragment, it inserts a `simulate-abs` statement instead. In the analysis of the program, when the tool reaches a `simulate` or a `simulate-abs` statement in the code, it switches precise program analysis to stochastic simulation or abstraction-then sampling, respectively.

```
const MAX:=14;
secret int32 sec := [251,750];
observable int32 obs:= 0;
public int32 time := 0;
public int32 loc := 0;
public int32 seed := 0;
public int32 ran := 0;
if (sec <= 425) then
  assign loc:=400;
elif (sec <= 475) then
  assign loc:=450;
elif (sec <= 525) then
  assign loc:=500;
elif (sec <= 575) then
  assign loc:=550;
else
  assign loc:=600;
fi
while (time < MAX) do
  random ran := random(1,6);
  if (ran <= 3) then
    assign loc:=loc+10;
  else
    assign loc:=loc-10;
  fi
  assign time:=time+1;
od
assign obs:=loc;
return;
```

Fig. 1: Random walk.

Motivating example. Consider the following random walk problem (modeled in Fig. 1).

The secret is the initial location of an agent, encoded by a single natural number representing an approximate distance from a given point, e.g. in meters. Then the agent takes a fixed number of steps. At each step the distance of the agent increases or decreases by 10 meters with the same probability. After this fixed number of random walk, the final location of the agent is revealed, and the attacker uses it to guess the initial location of the agent.

This problem is too complicated to analyze by precise analysis, because the analysis needs to explore every possible combination of random paths, amounting to an exponential number in the walking steps. It is also intractable to analyze with a fully statistical approach, since there are hundreds of possible secret values and the program has to be simulated many times for each of them to sufficiently observe the agent's behavior.

As shown in Section 4, HyLeak's hybrid approach computes the leakage significantly faster than the fully precise analysis and more accurately than the fully statistical analysis.

2 HyLeak Implementation

We describe how HyLeak estimates the Shannon leakage of an input program. The tool determines which component of the program to analyze with precise analysis and which with randomized analysis, and inserts appropriate annotations in the code. The components are analyzed with the chosen technique and the results are composed into a joint probability distribution of the secret and observable variables. Finally, the Shannon leakage and its confidence interval are computed. The tool implementation consists of the following 4 steps. Steps 1 and 2 are implemented with different ANTLR parsers [13]. The implementation of Step 3 inherits a large amount of code from the QUAIL tool. This means that QUAIL's optimizations, i.e., parallel analysis of execution traces and compact Markovian state representation, are inherited. However QUAIL's restrictions are inherited as well in HyLeak, meaning that the prior distribution on the secret is assumed to be uniform and no private variable can appear in assignments.

Step 1: Preprocessing

Step 1a. Lexing, parsing and syntax checking. The tool starts by lexical analysis, macro substitution and syntax analysis. In macro substitution the constants defined in the input program are replaced with their declared values, and simple operations are resolved immediately. The tool checks whether the input program correctly satisfies the language syntax. In case of syntax errors, an error message describing the problem is produced.

Step 1b. Loop unrolling and array expansion. `for` loops ranging over fixed intervals are unrolled to optimize the computation of variable ranges and thus program decomposition in Step 2. Similarly, arrays are replaced with multiple variables indexed by their position number in the array. Note that these techniques are used only to optimize program decomposition and not required to compute the leakage in programs with arbitrary loops.

Step 2: Program Decomposition and Internal Code Generation

If a `simulate` or `simulate-abs` statement is present in the code, this step is skipped. Otherwise, for each variable and each code line, an estimation of the number of possible values of the variable at the specific code line is computed. This is used to evaluate at each point in the input program whether it would be more expensive to use precise or statistical analysis, as explained in Section 5 of [9]. The tool adds `simulate` and/or `simulate-abs` statements in the code to signal which parts of the input program should be analyzed with standard statistical sampling and with abstraction-then-sampling. At the end, the input program is translated into a simplified internal language. Conditional statements and loops (`if`, `for`, and `while`) are rewritten into `if-goto` statements.

Step 3: Program Analysis

In this step the tool analyzes the executions of the program using the two approaches.

Step 3a. Precise analysis. The tool performs a depth-first symbolic execution of all possible execution traces of the input program, until it finds a `return`, `simulate`, or `simulate-abs` statement. When reaching a `return` statement the tool recognizes the execution trace as terminated and stores its secret and output values. In the cases of `simulate` and `simulate-abs` statements it halts the execution of the trace, saves the resulting program state, and schedules it for stochastic simulation or for abstraction-then-sampling simulation, respectively, starting from the saved program state.

Step 3b. Randomized analysis. The tool performs all the standard stochastic simulations and abstraction-then-sampling simulations, using the saved program states from Step 3a as starting point of each component to analyze stochastically. The sample size for each simulation is automatically decided by using heuristics to have better accuracy with less sample size. The results of each analysis is stored as an appropriate joint probability sub-distribution between secret and observable values.

Step 4: Leakage Estimation

In this step the tool aggregates all the data collected by the program analysis (performed in Steps 3) and estimates the Shannon leakage of the input program, together with evaluation of the estimation. More specifically, it constructs an (approximate) joint posterior distribution of the secret and observable values of the input program from all the collected data produced by Step 3, as explained in Section 3.1 of [9]. Then the tool estimates the Shannon leakage value from the joint distribution, including bias correction (See more details in Section 4 of [9]). Finally, a 95% confidence interval for the estimated leakage value is computed to roughly evaluate the quality of the analysis.

3 On Using HyLeak

HyLeak is freely available from https://project.inria.fr/hyleak, in both source code and artifact form. Multiple examples and the scripts to generate the results are also provided.

We show how to use HyLeak to analyze the random walk example presented in Section 1. The program code is shown in Fig. 1 on the left; assume it is contained in the file `random_walk_abs-sim.hyleak`. We invoke the tool with the command:

```
./hyleak random_walk_abs-sim.hyleak
```

The tool generates various `.pp` text files with analysis information and the control flow graph of the program (Fig. 2). Finally, it outputs the prior and posterior Shannon entropy estimates, the estimated leakage of the program before and after bias correction, and its confidence interval. HyLeak can also print the channel matrix and additional information; the full list of arguments is printed by `./hyleak -h`.

4 Comparison with Other Tools

The HyLeak tool processes a simple imperative language that is an extension of the language used in the QUAIL tool version 2.0 [2]. The QUAIL tool implements only a precise calculation of leakage that examines all executions of programs. Hence the performance of QUAIL does not scale, especially when the program performs complicated computations that yield a large number of execution traces. On the other hand, HyLeak fully supports the statistical approach and the hybrid approach. Hence HyLeak can analyze large problems that QUAIL cannot handle. Note that an approach combining static and randomized analyses was first proposed by Köpf and Rybalchenko [11] differently.

The stochastic simulation techniques implemented in HyLeak have also been developed in the tools LeakiEst [6] (with its extension [10]) and LeakWatch [7, 8]. Below we compare HyLeak's analysis technique against the full simulation technique implemented in these tools.

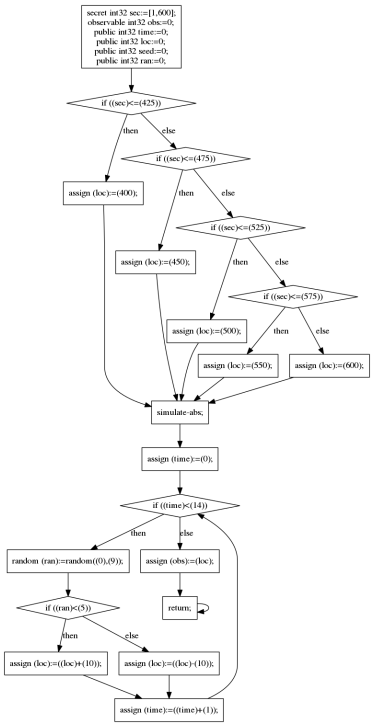

Fig. 2: Control flow graph for the input code of Fig. 1.

The tool Moped-QLeak [4] computes the precise information leakage of a program by transforming it into an algebraic decision diagram (ADD). As noted in [2], this technique is efficient when the program under analysis is simple enough to be converted into an ADD, and fails otherwise even when other tools including HyLeak can handle it.

Many information leakage tools restricted to deterministic input programs have been released, including TEMU [12], squifc [14], jpf-qif [15], QILURA [16], nsqflow [17], and SHARPPI [18]. Some of these tools have been proven to scale to programs of thousands of lines written in common languages like C and Java. Such tools rely on the fact that the Shannon leakage of a deterministic program is bounded from above

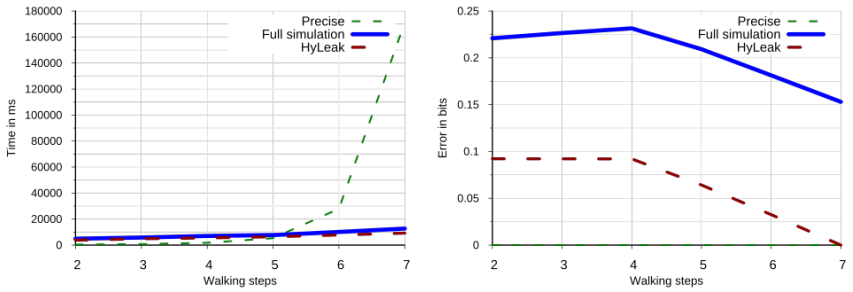

Fig. 3: Random walk experimental results

by the logarithm of the number of possible outputs of the program. The number of possible outputs is usually computed using model counting on a SMT-constraint-based representation of the possible outputs, obtained by analyzing the program. Contrary to these tools, HyLeak can analyze randomized programs[4] and provides a quite precise estimation of the leakage of the program, not just an upper bound. As far as we know, HyLeak is the most efficient tool that has this greater scope and higher precision.

Experimental Results

In this section we compare the performance of the tool HyLeak against the precise analysis technique (implemented in version 2.0 of QUAIL) and the statistical technique (used in LeakiEst and LeakWatch). In the experiments we use an option of HyLeak that deactivates stochastic simulations and performs fully precise analysis, which has basically an identical behavior to the QUAIL tool. As another comparison, we have forced fully randomized analysis like LeakWatch.

The random walk example in Fig. 1 has a conditional branching inside the `while` loop, and thus it has an exponential number of execution traces in the walking time `time`. Hence precise analysis takes an exponential time while both HyLeak and fully randomized analysis take much less time thanks to random sampling of traces (Fig. 3 on the left). Since HyLeak uses an abstraction-then-sampling technique, it has smaller errors than fully randomized analysis with an identical sample size (Fig. 3 on the right).

See Appendix for other examples and the results of their experiments.

References

1. F. Biondi, A. Legay, P. Malacaria, and A. Wasowski. Quantifying information leakage of randomized protocols. In *Proc. of VMCAI 2013*, pages 68–87, 2013.

[4] Some of these tools, like jpf-qif and nsqflow, present case studies on randomized protocols. However, the randomness of the programs is assumed to have the most leaking behavior. E.g., in the Dining Cryptographers this means assuming all coins produce head with probability 1.

2. F. Biondi, A. Legay, and J. Quilbeuf. Comparative analysis of leakage tools on scalable case studies. In *Proc. of SPIN 2015*, pages 263–281, 2015.
3. F. Biondi, A. Legay, L. Traonouez, and A. Wasowski. QUAIL: A quantitative security analyzer for imperative code. In *Proc. of CAV 2013*, pages 702–707, 2013.
4. R. Chadha, U. Mathur, and S. Schwoon. Computing information flow using symbolic model-checking. In *Proc. of FSTTCS 2014*, pages 505–516, 2014.
5. K. Chatzikokolakis, T. Chothia, and A. Guha. Statistical measurement of information leakage. In J. Esparza and R. Majumdar, editors, *TACAS*, 2010.
6. T. Chothia, Y. Kawamoto, and C. Novakovic. A tool for estimating information leakage. In *Proc. of CAV 2013*, pages 690–695, 2013.
7. T. Chothia, Y. Kawamoto, and C. Novakovic. Leakwatch: Estimating information leakage from java programs. In *Proc. of ESORICS 2014, Part II*, pages 219–236, 2014.
8. T. Chothia, Y. Kawamoto, C. Novakovic, and D. Parker. Probabilistic point-to-point information leakage. In *Proc. of CSF 2013*, pages 193–205, 2013.
9. Y. Kawamoto, F. Biondi, and A. Legay. Hybrid statistical estimation of mutual information for quantifying information flow. In *Proc. of FM 2016*, pages 406–425, 2016.
10. Y. Kawamoto, K. Chatzikokolakis, and C. Palamidessi. Compositionality results for quantitative information flow. In *Proc. of QEST 2014*, pages 368–383, 2014.
11. B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *Proc. of CSF 2010*, pages 3–14, 2010.
12. J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In S. Chong and D. A. Naumann, editors, *PLAS*. ACM, 2009.
13. T. Parr. *The Definitive ANTLR Reference: Building Domain Specific Languages*. 2007.
14. Q. Phan and P. Malacaria. Abstract model counting: a novel approach for quantification of information leaks. In *Proc. of ASIA CCS 2014*, pages 283–292, 2014.
15. Q. Phan, P. Malacaria, O. Tkachuk, and C. S. Pasareanu. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.
16. Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. D'Amorim. Quantifying information leaks using reliability analysis. In *Proc. of SPIN 2014*, pages 105–108. ACM, 2014.
17. C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu. Precisely measuring quantitative information flow: 10k lines of code and beyond. In *EuroS&P 2016*, pages 31–46, 2016.
18. A. Weigl. Efficient SAT-based pre-image enumeration for quantitative information flow in programs. In *Proc. of QASA 2016*, pages 51–58, 2016.

A HyLeak imperative language

The input language for programs to be analyzed by HyLeak is almost identical to the input language of the QUAIL tool [3] that HyLeak is based on. The only additions are the bounded FOR loops described in Section A.9 and the Simulate statements described in Section A.10. The rest of the input language semantics is reprinted here from the Appendix of [3] for reference.

A.1 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.

A.2 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.

A.3 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.

A.4 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.

A.5 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.

A.6 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 $1 - \texttt{p}$.

A.7 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
```

A.8 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
```

A.9 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.

A.10 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 [9]. 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.

A.11 Return statements

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

```
return;
```

B Control Flow Graphs

In this section we present in Figure 2 an example of the control flow graph generated by HyLeak. Note that HyLeak has added `simulate-abs` statements to the code, visible in the flow graph.

C Further Examples

In this section we present more application of the tool HyLeak.

C.1 Probabilistically Terminating Loop

The tool HyLeak can analyze programs that terminate only probabilistically. For instance, the program shown in Fig. 4 has a loop that terminates depending on the randomly generated value of the variable `rand`. No previous work has presented an automatic measurement of information leakage by probabilistic termination, as precise analysis cannot handle non-terminating programs, which typically causes non-termination of the analysis of the program. On the other hand, the stochastic simulation of this program supported in HyLeak terminates after some number of iterations in practice although it may take long for some program executions to terminate. When the sample size is 50000 the Shannon leakage computed with HyLeak is about 0.4319 and the analysis takes about 10 seconds.

```
const bound := 10;
secret int32 sec := [0,5];
observable int32 obs;
public int32 time := 0;
public int2 terminate := 0;
public int32 rand;
simulate;
random rand := random(0,9);

//probabilistically terminating
while (terminate != 1) do
  random rand:= random(1,5);
  if (sec <= rand) then
    assign terminate := 1;
  fi
  assign time := time+1;
od

//output
if (time < bound) then
  assign obs := time;
else
  assign obs := bound;
fi
return;
```

Fig. 4: Probabilistically Terminating Loop.

C.2 Smart Grid Privacy

A smart grid is an energy network where users (like households) may consume or produce energy. In Fig. 5 we describe a simple model of a smart grid using the HyLeak language. This example is taken from [2]. The users periodically negotiate with a central aggregator in charge of balancing the total consumption among several users. In practice each user declares to the aggregator its consumption plan. The aggregator sums up the consumptions of the users and checks if it falls within admitted bounds. If not it answers to the users that the consumption is too low or too high by a certain amount, such that they adapt their demand. This model raises some privacy issues as some attacker can try to guess the consumption of a user, and for instance infer whether or not this particular user is at home.

In Fig. 6 we present the experiment results of this smart grid example for different numbers of users. HyLeak takes less time than both fully precise analysis and fully randomized analysis (as shown in the left figure). Moreover it is closer to the true value than fully randomized analysis especially when the number of users is larger (as shown in the right figure).

C.3 Shifting Window

In the shifting window example (Fig .7, from [9]) the secret `sec` can take N possible values, and an interval (called a "window") in the secret domain is randomly selected from 1 to W. If the value of the secret is inside the window, then another window is randomly chosen in a similar way and the program outputs a random value from this second window. Otherwise, the program outputs a random value over the secret domain.

In Fig. 8 we present the results of experiments on the shifting window when increasing the size of the secret domain. The execution time of precise analysis grows proportionally to the secret domain size N while HyLeak and fully randomized analysis do not require much time for a larger N. In the fully randomized analysis the error from the true value grows rapidly while in using HyLeak the error is much smaller.

```
const N:=3;   // N is the total number of users
const S:=1;   // S is the number of users we care about
const C:=3;   // C is the possible consumptions level
const M:=0;   // M is the consumption level of the attacker
const LOWT:=2;   // LOWT is the lower threshold
const HIGHT:=9;   // HIGHT is the upper treshold
// the observable is the order given by the control system
observable int32 order;
observable int1 ordersign;

// The secret is the consumption of each user we care about
secret array [S] of int32 secretconsumption:=[0,C-1];
// The other consumptions are just private
private array [N-(S+1)] of int32 privateconsumption:=[0,C-1];

public int32 total:=M;   // this is the projected consumption
public int32 j:=0;   // this is just a counter

// count the secret consumptions
for (i in [0,S-1]) do
  while (j<C) do
    if (secretconsumption[i]==j) then
      assign total:=total+j;
    fi
    assign j:=j+1;
  od
  assign j:=0;
od

// count the private consumptions
assign i := 0;
assign j := 0;
while (i<N-(S+1)) do
  while (j<C) do
    if (privateconsumption[i]==j) then
      assign total:=total+j;
    fi
    assign j:=j+1;
  od
  assign j:=0;
  assign i:=i+1;
od

if (total<LOWT) then
  assign order := LOWT - total;
  assign ordersign := 0;
elif (total > HIGHT) then
  assign order := total - HIGHT;
  assign ordersign := 1;
else
  assign order := 0;
  assign ordersign := 0;
fi
return;
```

Fig. 5: Smart Grid Example.

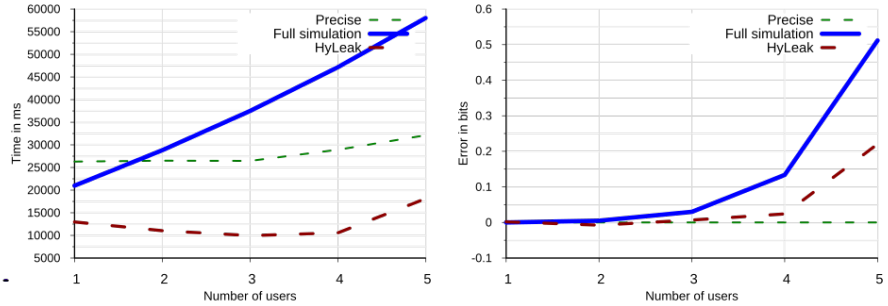

Fig. 6: Smart grid experimental results.

```
const N:=16;
const W:=14;
secret int32 sec := [0,N-1];
public int32 minS;
public int32 sizeS;
observable int32 obs;
public int32 minO;
public int32 sizeO;

random minS := random(0,N-W-1);
if (sec>=minS) then
  random sizeS := random(1,W);
  if (sec<=(minS+sizeS)) then
    random minO := random(0,N-W-1);
    random sizeO := random(1,W);
    random obs := random(minO,minO+sizeO);
  else
    random obs := random(0,N-1);
  fi
else
  random obs := random(0,N-1);
fi
return;
```

Fig. 7: Shifting Window Example.

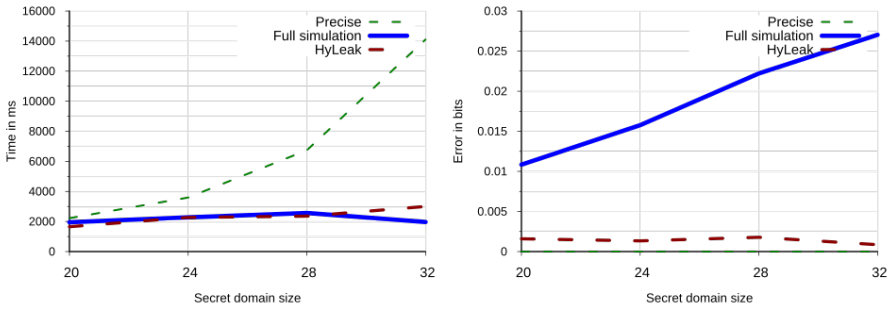

Fig. 8: Shifting window experimental results.