QUAIL: A Quantitative Security Analyzer for Imperative Code

Fabrizio Biondi, Axel Legay, Louis-Marie Traonouez, Andrzej Wasowski

QUAIL: A Quantitative Security Analyzer for Imperative Code*

Fabrizio Biondi[1], Axel Legay[2], Louis-Marie Traonouez[2], and Andrzej Wąsowski[1]

[1] IT University of Copenhagen, Denmark
[2] INRIA Rennes, France

Abstract. Quantitative security analysis evaluates and compares how effectively a system protects its secret data. We introduce QUAIL, the first tool able to perform an arbitrary-precision quantitative analysis of the security of a system depending on private information. QUAIL builds a Markov Chain model of the system's behavior as observed by an attacker, and computes the correlation between the system's observable output and the behavior depending on the private information, obtaining the expected amount of bits of the secret that the attacker will infer by observing the system. QUAIL is able to evaluate the safety of randomized protocols depending on secret data, allowing to verify a security protocol's effectiveness. We experiment with a few examples and show that QUAIL's security analysis is more accurate and revealing than results of other tools.

1 Introduction

The Challenge. Qualitative analysis tools can verify the complete security of a protocol, i.e. that an attacker is unable to get any information on a secret by observing the system—a property known as *non-interference*. Non-interference holds when the system's output is independent from the value of the secret, so no information about the latter can be inferred from the former [20]. However, when non-interference does not hold, qualitative analysis cannot rank the security of a system: all unsafe systems are the same.

Quantitative analysis can be used to decide which of two alternative protocols is more secure. It can also asses security of systems that are insecure, but nevertheless useful, in the qualitative sense, such as a password authentication protocol, for which there is always a positive probability that an attacker will randomly guess the password. A quantitative analysis is challenging because it is not sufficient to find a counterexample to a specification to terminate. We need to analyze all possible behaviors of the system and quantify for each one the probability that it will happen and how much of the protocol's secret will be revealed. So far no tool was able to perform this analysis precisely.

Quantitative analysis with QUAIL. We use Quantified Information Flow to reduce the comparison of security of two systems to a computation of expected amount of information, in the information-theoretical sense, that an attacker would learn about the secret by observing a system's behavior. This expected amount of information is known as *information leakage* [9,15,8,12,21] of a system. It amounts to zero iff the system is

* Partially supported by MT-LAB — VKR Centre of Excellence on Modeling of IT

non-interfering [15], else it represents the expected number of bits of the secret that the attacker is able to infer. The analysis generalizes naturally to more than two systems, hence allowing to decide which of them is less of a threat to the secrecy of the data.

To compute information leakage we use a stochastic model of a system as observed by the attacker. The model is obtained by resolving non-determinism in the system code, using the prior probability distribution over the secret values known to the attacker before an attack. Existing techniques represent this with a channel matrix from secret values to outputs [5]. They build a row of the channel matrix for each possible value of the secret, even if the system would behave in the same way for most of them. In contrast, we have proposed an automata based technique [3], using Markovian models. One state of a model represents an interval of values of the secret for which the system behaves in the same way, allowing for a much more compact and tractable representation.

We build a Markov chain representing the behavior observed by the attacker, then we hide the states that are not observable by the attacker, obtaining a smaller Markov chain—an *observable reduction*. Then we calculate the correlation between the output the attacker can observe and the behavior dependent on the secret, as it corresponds to the leakage. Since leakage in this case is mutual information, it can be computed by adding the entropy of the observable and secret-dependent views of the system and subtracting the entropy of the behavior depending on both. See [3] for details.

QUAIL (QUantitative Analyzer for Imperative Languages) implements this method. It is the first tool supporting arbitrary-precision quantitative evaluation of information leakage for randomized systems or protocols with secret data, including anonymity and authentication protocols. Systems are specified in a simple imperative modeling language further described on QUAILS website.

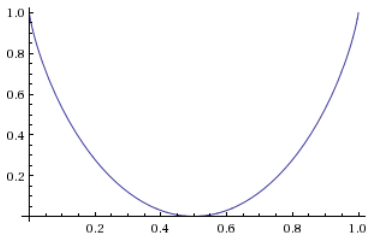


Fig. 1: Bit XOR leakage as a function of $\Pr(r = 1)$

QUAIL performs a white-box leakage analysis assuming that the attacker has knowledge of the system's code but no knowledge of the secret's value, and outputs the result and eventually information about the computation, including the Markov chains computed during the process.

Example. Consider a simple XOR operation example. Variable h stores a 1-bit secret. The protocol generates a random bit r, where $r = 1$ with probability p. It outputs the result of exclusive-or between values of h and r. The attacker knows p and can observe the output, so if $h = r$, but not the values of r or h.

If $p = 0.5$ the attacker cannot infer any information about h, the leakage is zero bits (non-interference). If $p = 0$ or $p = 1$ then she can determine precisely the value of h, and thus the leakage is 1 bit. This can be verified efficiently with language-based tools like APEX [10]. However, QUAIL is the only tool able to precisely compute the leakage for all possible values of p with arbitrary precision. Figure 1 shows that XOR protocol leaks more information as the value of r becomes more deterministic. For instance $p = 0.4$ is safer than $p = 0.8$.

2 QUAIL Implementation

The input model is specifed in QUAIL's imperative language designed to facilitate succinct and direct modeling of protocols, providing features such as arbitrary-size integer variables and arrays, random assignments, `while` and `for` loops, named constants and conditional statements. Figure 2 presents the input code for the bit XOR example.

For a given input code QUAIL builds an annotated Markov chain representing all possible executions of the protocol, then modifies it to encode the protocol when observed by the attacker whose aim is to discover the protocol's secret data. Finally, QUAIL extracts a model of the observable and secret-dependent behavior of the system, and computes the correlation between them, which is equivalent to the amount of bits of the secret that the attacker can infer by observing the system. We now discuss QUAIL implementation following the five steps of the method proposed in [3]:

Step 1: Preprocessing. QUAIL translates the input code into a simplified internal language. It rewrites conditional statements and loops (`if`, `for` and `while`) to conditional jumps (`if-goto`) and substitutes values for named constant references.

Step 2: Probabilistic symbolic execution. QUAIL performs a symbolic forward execution of the input program constructing its semantics as a finite Markov chain (a fully probabilistic transition system) with a single starting state. To this end, QUAIL needs to know the attacker's probability distributions over the secret variables. For each conditional branch, we compute the conditional probability of the guard being satisfied given the values of the public variables and the probability distributions over the secret variables. Then QUAIL generates two successor states, one for the case in which the guard is satisfied and one when not satisfied. This is the most time-consuming step, so QUAIL uses an on-the-fly optimization to avoid building internal states that would be removed in the next step. For instance, it does not generate new states for assignments to a non-observable public variable. Instead it changes the value of the variable in the current state.

Step 3: State hiding and model reduction. To represent what the attacker can examine, QUAIL reduces the Markov chain model by iteratively hiding all unobservable states. For the standard attacker, these are all the internal states, i.e. all the states except the initial and the output states. A state is hidden by creating transitions from its predecessors to its successors and removing it from the model. This operation normally eliminates more than 90% of the states of the Markov chain model, building its *observable reduction*. This operation also detects non-terminating loops and collapses then in a single non-termination state. States are equipped with a list of their predecessors and successors to quicken this step. An observable reduction looks like a probability distribution from the starting states to the output states, since all other states are hidden.

Step 4: Quotienting. Recall from Sect.1 that we have to quantify the correlation between the observable and secret-dependent views of the system. QUAIL relies on the notion of quotients to represent different views of the system and compute their correlation. A quotient is a Markov chain obtained by merging together states in the observable

```
1  observable int1 l; // bit l is the output
2  public int1 r; // bit r is random
3  secret int1 h; // bit h is the secret
4  random r:=randombit(0.5); // randomize r
5  if (h==r) then // calculate the XOR
6  assign l:=0;
7  else
8  assign l:=1;
9  fi
10 return; //terminate
```

```
1  observable int1 l;
2  public int1 r;
3  secret int1 h;
4  random (r):=randombit(0.5);
5  if ((h)==(r))
6  then goto 8;
7  else goto 10;
8  assign (l):=(0);
9  goto 11;
10 assign (l):=(1);
11 return;
```

Fig. 2: Bit XOR example: input code (on the left) and preprocessed code (on the right).

reduction that give the same value to some of the variables. QUAIL quotients the observable reduction separately three times to build three different views of the system. QUAIL uses the attacker model again to know which states are indistinguishable as they assign the same values to the observable variables. These states are merged in the *attacker's quotient*. Similarly, in the *secret's quotient* states are merged if they have the same possible values for the secret, while in the *joint quotient* states are merged if they both have the same values for the secret and cannot be discriminated by the attacker. Since information about the states' variables is not needed to compute entropy, quotients carry none, reducing time and memory required to compute them.

Step 5: Entropy and leakage computation. The *information leakage* can be computed as the sum of the entropies of the attacker's and secret's quotients minus the entropy of the joint quotient [3]. The three entropy computations are independent and can be parallelized. QUAIL outputs the leakage with the desired amount of significant digits and the running time in milliseconds. If requested, QUAIL plots the Markov chain models using Graphviz.

3 On Using QUAIL

QUAIL is freely available from https://project.inria.fr/quail, including source code, binaries and example files. We demonstrate usage of QUAIL to analyze the bit XOR example. Let bit_xor.quail be the file containing the input shown in Fig. 2. The command

```
quail bit_xor.quail -p 2 -v 0
```

executes QUAIL with precision limited to 2 digits (`-p 2`), suppressing all output except the leakage result (`-v 0`). In response QUAIL generates a file bit_xor.quail.pp with the preprocessed code shown in Fig. 2, analyzes it and finally answers `0.0` showing that in this case the protocol leaks no information (so non-interference). For different probability of the random bit r in line 4 QUAIL obtains a different leakage (cf. Fig. 1). For instance, for $p = 0.8$ the leakage is $\sim$0.27807 bits.

4 Comparison with Other Tools

QUAIL precisely evaluates the value of leakage of the input code. This not only allows proving non-interference (absence of leakage) but also enables comparing relative safety of similar protocols. This is particularly important for protocols that exhibit

Table 1: QUAIL analysis of the leakage in an authentication program

Password length	2	32	64	500
Leakage	$8.11 \cdot 10^{-1}$	$7.78 \cdot 10^{-9}$	$3.54 \cdot 10^{-18}$	$1.52 \cdot 10^{-148}$

inherent leakage, such as authentication protocols. For instance, with a simple password authentication, the user inputs a password and is granted access privilege if the password corresponds to the secret stored in the system. The chance of an attacker guessing a password is always positive (although it depends on the password's length). Also, even if the attacker gets rejected she learns something about the secret—the fact that the attempted value was not correct. QUAIL can quantify the precise leakage as a function of the bit length of the password, as shown in Table 1.

Existing *qualitative* tools can establish whether a protocol is completely secure or not, i.e. whether it respects non-interference. They cannot discriminate protocols that allow acceptable and unacceptable violations of non-interference. APEX [10] is an analyzer for probabilistic programs that can check programs equivalence, while PRISM [14] is a probabilistic model-checker. With these tools authentication protocols will always be flagged as unsafe, and a comparison between them is impossible.

QUAIL can be used also to analyze anonymity protocols, like the grade protocol and the dining cryptographers [6]. The interested reader can find discussion and input code for these examples on the QUAIL website. These protocols provide full anonymity on the condition that some random data is generated with a uniform probability distribution; their effectiveness in these cases can be efficiently verified with the qualitative tools above. If the probability distribution over the random data is not uniform some private data is leaked, and QUAIL is again the only tool that can quantify this leakage. Presently, the models for these protocols tend to grow exponentially, so the analysis becomes time-consuming already for about 6–7 agents.

Qualitative tools and technique are more closely related to QUAIL; we present some of them and discuss the main differences. It is worth noting that most of them either do not work for analyzing probabilistic programs [1,11,13,17] or are based on a channel matrix with an impractical number of lines [4,7].

JPF-QIF [19] is a tool that computes an upper bound of the leakage by estimating the number of possible outputs of the system. JPG-QIF is much less precise than QUAIL, and it is not able for instance to prove that the security of an authentication increases by increasing the password size.

McCamant and Ernst [16] and Newsome, McCamant and Song [18] propose quantitative extensions of taint analysis. This approach, while feasible even for large programs, still does not allow to analyze probabilistic programs, making it unsuitable for security protocols.

Bérard et al. propose a quantification of information leakage based on mutual information, though they name it restrictive probabilistic opacity [2] and do not refer to some of the core papers of the subject, like the works of Clark, Hunt and Malacaria [8,9]. The approach tries to quantify leakage on probabilistic models, and is thus phylosophically close to ours. They compute mutual information as the expected difference between

prior and posterior entropy, and since the latter depends on all possible values of the secret we expect that an eventual implementation would be in general very inefficient compared to the QUAIL quotient-based approach.

References

1. Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *IEEE Symposium on Security and Privacy*, 2009.
2. Béatrice Bérard, John Mullins, and Mathieu Sassolas. Quantifying opacity. In G. Ciardo and R. Segala, editors, *QEST'10*. IEEE Computer Society, September 2010.
3. Fabrizio Biondi, Axel Legay, Pasquale Malacaria, and Andrzej Wasowski. Quantifying information leakage of randomized protocols. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *VMCAI*. Springer, 2013.
4. Konstantinos Chatzikokolakis, Tom Chothia, and Apratim Guha. Statistical measurement of information leakage. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, 2010.
5. Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. In *Information and Computation*. Springer, 2006.
6. David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.
7. Tom Chothia and Apratim Guha. A statistical test for information leaks using continuous mutual information. In *CSF*, pages 177–190. IEEE Computer Society, 2011.
8. David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. *Electr. Notes Theor. Comput. Sci.*, 59(3):238–251, 2001.
9. David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15, 2007.
10. S. Kiefer, A. S. Murawski, J. Ouaknine, B. Wachter, and J. Worrell. Apex: An analyzer for open probabilistic programs. In *Proc. CAV'12*, LNCS. Springer, 2012.
11. Vladimir Klebanov. Precise quantitative information flow analysis using symbolic model counting. In Fabio Martinelli and Flemming Nielson, editors, *QASA*, 2012.
12. Boris Köpf and David A. Basin. An information-theoretic model for adaptive side-channel attacks. In *ACM Conference on Computer and Communications Security*, 2007.
13. Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, 2012.
14. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. CAV'11*, volume 6806 of *LNCS*. Springer, 2011.
15. Pasquale Malacaria. Algebraic foundations for information theoretical, probabilistic and guessability measures of information flow. *CoRR*, abs/1101.3453, 2011.
16. Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In Rajiv Gupta and Saman P. Amarasinghe, editors, *PLDI*. ACM, 2008.
17. Chunyan Mu and David Clark. A tool: Quantitative analyser for programs. In *QEST*. IEEE Computer Society, 2011.
18. James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In S. Chong and D. A. Naumann, editors, *PLAS*. ACM, 2009.
19. Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Pasareanu. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.
20. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
21. Geoffrey Smith. On the foundations of quantitative information flow. In Luca de Alfaro, editor, *FOSSACS*, volume 5504 of *LNCS*, pages 288–302. Springer, 2009.

A Running QUAIL through an example

We detail in this appendix the result of the computations performed by QUAIL when analyzing the bit XOR example presented in Fig. 2. We consider the value of 0.8 for the random bit probability on line 4. We launch QUAIL as explained in Section 3:

```
quail bit_xor.quail -p 3 -v 0 -mc 0 1 2 3 4 -Tpdf
```

This will compute the leakage of the program with a precision of 3 digits for the computations. Using the options `-mc 0 1 2 3 4 -Tpdf` it will also produce 5 Markov chains in a PDF format, corresponding to the different intermediate steps of the computation. As a result the tool outputs the leakage of the program which is:

```
0.28
```

To compute this result Step 1 converts the input program given on the left of Fig. 2 into a preprocessed program shown on the right. This program is written in a separate file `bit_xor.quail.pp`.

In Step 2, QUAIL parses the preprocessed code and builds the Markov chain model shown in Fig. 3 that corresponds to all the executions of the program. Note that public variables are labeled with a precise value in each state (e.g. `l=0`) while private variables have intervals of allowed values (e.g. `h=[0,1]`).

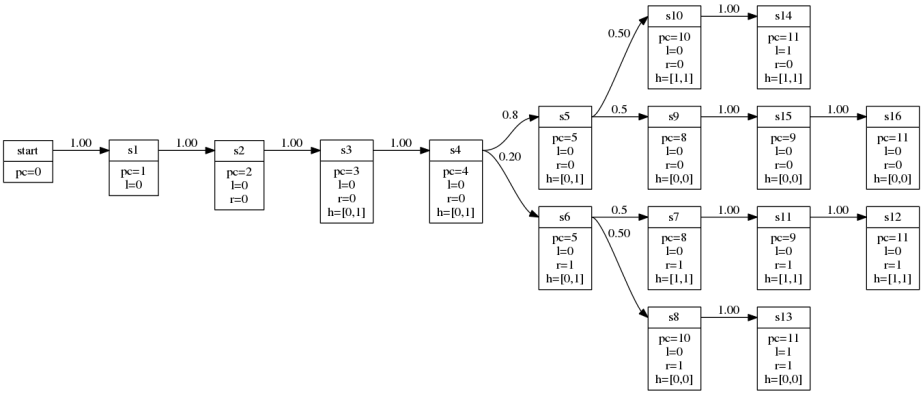


Fig. 3: Bit XOR example: Markov chain model of the program

In Step 3, QUAIL hides all non observable states, i.e. the internal states, and produces a Markov chain with only transitions between the initial states and the output states, as shown in Fig. 4.

In Step 4, QUAIL computes the quotient Markov chains. In the attacker's quotient, since in both states s13 and s14 the observable variable l equals to 1, these two states are merged together, and similarly for the states s12 and s16 in which $l = 0$. The result is shown in Fig. 5a.

In the secret's quotient, the states s12 and s14 share the value $h = 0$ for the secret variable, whereas the states s13 and s16 share the value $h = 1$. These states are merged together as shown in Fig. 5b.

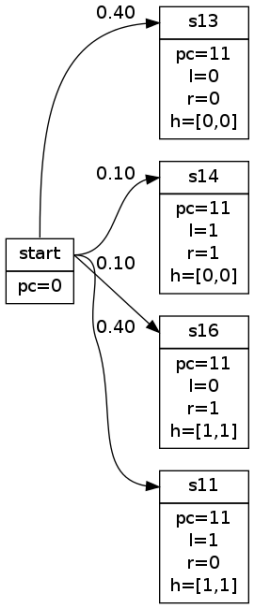


Fig. 4: Bit XOR example: observable reduction of the Markov chain model

Finally, in the joint quotient, we consider the intersection of the two previous discrimination relations, i.e. only states with the same value to both observable and secret variables can be merged together. This is not the case for any state in the observable reduction, and therefore the joint quotient Markov chain shown in Fig. 5c is similar to the original observable reduction of Fig. 4.

In the final Step 5, QUAIL computes the entropy of the three quotients Markov chains. The entropy of the attacker's quotient is:

$$H_a = -0.5 * log(1/0.5) - 0.5 * log(1/0.5) = 1$$

The entropy of the secret's quotient is the same:

$$H_s = -0.5 * log(1/0.5) - 0.5 * log(1/0.5) = 1$$

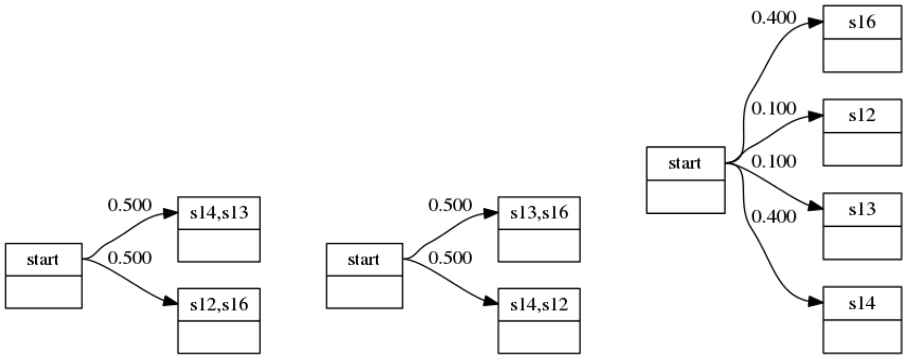
The entropy of the joint quotient is:

$$H_j = -0.4 * log(1/0.4) * 2 - 0.1 * log(1/0.1) * 2 = 1.058 + 0.664 = 1.722$$

The information leakage is the sum of the first two entropies minus the entropy of the joint quotient, therefore:

$$L = H_a + H_s - H_j = 2 - 1.722 = 0.278$$

This result is rounded to 0.28 in the final output.



(a) Attacker's quotient (b) Secret's quotient (c) Joint quotient

Fig. 5: Markov chains of the quotients

B QUAIL imperative language

B.1 Variable declarations

All variables in QUAIL 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.

B.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.

B.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.

B.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.

B.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.

B.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 $1 - p$.

B.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
```

B.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
```

B.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.

B.10 Return statements

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

```
return;
```

C Further examples

C.1 The Grade protocol

In the Grades protocol a group of k students $s_1, ..., s_k$ is given each a secret grade g_i between 0 and m-1. The students want to compute the sum of their grades without disclosing them. To this aim they produce k random numbers between 0 and $n = (m-1) * k + 1$ such that the number r_i is known only to the students s_i and $s_{(i+1)\%k}$. Then each student s_i outputs a number $d_i = g_i + r_i - r_{(i+1)\%k}$, and the sum of all grades is equivalent to the sum of the outputs modulo n. The input code for the Grade protocol is shown on Fig. 6 on the left.

To prove the security of the protocol, and thus the secrecy of the grades, we need to show that the information the attacker gains by knowing the declarations and the sum is the same as the information he would gain by knowing only the sum; the input code for the latter system is shown in Fig. 6 on the right. The leakage of the protocol for different numbers of students and grades is shown in Table 2(a); the leakage of the protocol declaring only the final sum is shown in Table 2(b). The tables are identical, demonstrating that when the attacker knows the students' declarations and the sum of the grades, she does not learn more information than the sum of the grades.

Table 2: Grades: leakage tables for attacker knowing a) the outputs and b) the sum only

(a)

Grades \ Students	**2**	**3**	**4**	**5**
2	1.500	1.811	2.030	2.198
3	2.197	2.525	2.745	2.910
4	2.655	2.984	3.201	3.365
5	2.999	3.325	3.541	timeout

(b)

Grades \ Students	**2**	**3**	**4**	**5**
2	1.500	1.811	2.030	2.198
3	2.197	2.525	2.745	2.910
4	2.655	2.984	3.201	3.365
5	2.999	3.325	3.541	timeout

C.2 The Dining Cryptographers protocol

The Dining Cryptographers protocol is an anonymity protocol in which a number of agents collaborate to a shared computation depending on each agent's secret [6]. A group of n cryptographers is dining around a round table. At the end of the dinner, the waiter informs them that the bill has already been settled by someone who would prefer to remain anonymous. The cryptographers respect the payer's wish for anonymity,

```
// S is the number of students
const S:=2;
// G is the number of grades (from 0 to G
    -1)
const G:=2;
// n is the number of possible random
    numbers generated
public int32 n;
// this is the sum that will be printed
observable int32 output;
// this is an internal counter for the sum
public int32 sum:=0;
// these are the random numbers; each one
    is shared between two students
public array [S] of int32 numbers;
// these are the public announcements of
    each student
observable array [S] of int32 announcements
    ;
// there are S secret votes, each one with
    G possible values:
secret array [S] of int32 h := [0,G-1];
// these are just counters
public int32 i:=0;
public int32 j:=0;

// calculating n
assign n:=((G-1)*S)+1;

// generate the random numbers
for (num in numbers) do
  random num:=random(0,n-1);
od

// producing the declarations according to
    the secret value
while (i<S) do
  assign j:=0;
  while (j<G) do
    if (h[i]==j) then
      assign announcements[i]:=(j+numbers[i
          ]-numbers[(i+1)%S])%n;
    fi
    assign j:=j+1;
  od
  assign i:=i+1;
od

//computing the sum, producing the output
    and terminating
for (a in announcements) do
  assign sum := sum+a;
od
assign output := sum%n;

return;
```

```
// S is the number of students
const S:=2;
// G is the number of grades (from 0
    to G-1)
const G:=2;
// this is the sum that will be
    printed
observable int32 output;
// this is an internal counter for the
    sum
public int32 sum:=0;
// there are S secrets, each one with
    G possible values:
secret array [S] of int32 h := [0,G
    -1];
// these are just counters
public int32 i:=0;
public int32 j:=0;

// computing the sum of the secrets
while (i<S) do
  assign j:=0;
  while (j<G) do
    if (h[i]==j) then
      assign sum := sum + j;
    fi
    assign j:=j+1;
  od
  assign i:=i+1;
od

//producing the output and terminating
assign output := sum;

return;
```

Fig. 6: Grade example: input code for Grade (on the left) and for a protocol revealing only the sum of the grades (on the right).

but would like to know whether the benefactor is one of them or an external party. To determine this, each pair of adjacent cryptographers toss a coin hidden from everybody else, so that each cryptographers knows the value of the coin to its left and to its right. Then each cryptographers declares aloud the exclusive OR of the two coins he sees, i.e. 0 if they have the same value and 1 otherwise. If one of the cryptographers is the payer, he declares the opposite. In the end, if an even number of ones is declared then someone else paid the bill, while if an odd number of ones is declared one of the cryptographers is the benefactor.

Figure 7 on the left shows the QUAIL input code for the Dining Cryptographers protocol. The leakage of the protocol depends on the randomness of the coin that the cryptographers toss; as the coin become more deterministic, so the probability of getting a head gets closer to 0 or 1, the attacker is more able to determine the identity of the payer.

Some results are shown in Fig. 7 on the right; for different numbers of cryptographers we show that as the probability of the coin toss approaches 0 or 1 the leakage increases. When it is 0 or 1 the leakage is equivalent to the bit size of the secret, i.e. the logarithm in base 2 of $n + 1$, proving that the whole secret gets leaked, and thus the attacker learns the identity of the payer, whoever he is.

```
1  // N is the number of cryptographers at the
       table
2  const N:=3;
3
4  // this bit represents the output
5  observable int1 output;
6
7  // these bits represent the coin tosses
8  public array [N] of int1 coin;
9
10 // these are the observable coins
11 observable array[2] of int1 obscoin;
12
13 // this is just a counter
14 public int32 i:=0;
15
16 // these bits represent the bits declared
       by the three cryptographers
17 observable array [N] of int1 decl;
18
19 // the secret has N+1 possible values:
20 // 0 if someone else paid
21 // 1 if Cryptographer A paid
22 // 2 if Cryptographer B paid
23 // 3 if Cryptographer C paid
24 // ... and so on
25 secret int32 h := [0,N];
26
27 // tossing the coins
28 for (c in coin) do
29   random c:=randombit(0.5);
30 od
31
32 // if the attacker is one of the
       cryptographers, he can observe two of
       the coins.
33 // To encode an external attacker comment
       the next two lines.
34 assign obscoin[0]:=coin[0];
35 assign obscoin[1]:=coin[1];
36
37 // producing the declarations according to
       the secret value
38 while (i<N) do
39   assign decl[i]:=coin[i]^coin[(i+1)%N];
40   if (h==i+1) then
41     assign decl[i]:=!decl[i];
42   fi
43   assign i:=i+1;
44 od
45
46 //producing the output bit and terminating
47 for (d in decl) do
48   assign output := output^d;
49 od
50
51 return;
```

		Cryptographers			
		3	**4**	**5**	**6**
Coin probability	**0**	2	2.32	2.59	2.81
	0.1	1.76	1.90	2.03	2.15
	0.3	1.56	1.49	1.45	1.41
	0.5	1.50	1.37	1.25	1.15
	0.7	1.56	1.49	1.45	1.41
	0.9	1.76	1.90	2.03	2.15
	1.0	2	2.32	2.59	2.81

Fig. 7: Dining Cryptographers example: input code for the Dining Cryptographers (on the left) and leakage table as a function of the number of cryptographers and of the probability of the random coin toss (on the right).