Comparative Analysis of Leakage Tools on Scalable Case Studies

Fabrizio Biondi, Axel Legay, Jean Quilbeuf

Comparative Analysis of Leakage Tools on Scalable Case Studies

Fabrizio Biondi[1], Axel Legay[1], and Jean Quilbeuf[1]

Inria

Abstract. Quantitative security techniques have been proven effective to measure the security of systems against various types of attackers. However, such techniques are often tested against small-scale academic examples.
In this paper we use analyze two scalable, real life privacy case studies: the privacy of the energy consumption data of the users of a smart grid network and the secrecy of the voters' voting preferences with different types of voting protocols.
We analyze both case studies with three state-of-the-art information leakage computation tools: LeakWatch, Moped-QLeak, and our tool QUAIL equipped with a new trace analysis algorithm. We highlight the relative advantages and drawbacks of the tools and compare their usability and effectiveness in analyzing the case studies.

1 Introduction

The protection of privacy and data security is one of the main concerns of computer science. Security often falls down to the impossibility for an attacker to obtain a given secret value. Such an impossibility can be defined by non-interference [17]. However this definition rejects any program which publishes any variable whose value depends on the secret. For instance, publishing the results of an election when each individual vote is secret breaks non-interference. Such a yes/no approach does not consider that an attacker may have a partial information about a secret.

Information-theoretical techniques have the advantage of considering the secret not as an atomic object but as a known number of secret bits, allowing the definition of measures of effectiveness of an attack based on the amount of secret bits that the attack compromises. The amount of secret bits that are compromised by an attack are known as *information leakage*. Leakage depends on the information about the secret known to the attacker before the attack, known as *prior information* and usually modeled as a *prior probability distribution* over the values of the secret. This approach dates back to Denning [15]. Different information leakage measures have been introduced, including Shannon leakage [18], min-entropy leakage [29] and the g-leakage [1], encoding different security properties of the system. All the tools we compare in this work can compute both Shannon and min-entropy leakage with no significant difference in computation time. We compare them on the computation of Shannon leakage, but we expect no significant difference if the tools were to be compared on the computation of of min-entropy leakage.

Among the results in the field, Köpf et al. studied leakage of side-channel attacks [2,20], while Boreale has defined leakage for process calculi [6] and characterized the best attack strategy of an adaptive attacker [9].

In this work we compare the three tools that compose the state of the art in Shannon leakage computation: QUAIL [5] equipped with a new trace analysis algorithm, LeakWatch [13], and Moped-QLeak [11].

QUAIL is a recent but already well established tool for precise and exact information leakage computation, and later tools by multiple authors have used it as comparison [13,24]. Nonetheless, QUAIL needs to produce a full Markov chain model of the system-attacker scenario to produce a meaningful result.

The first contribution of this paper is a new algorithm for precise information leakage computation, which is able to compute information leakage following the same Markovian semantics we introduced previously [3, Section 4] by performing a depth-first search analyzing the execution traces of the system. This avoids building the full Markov chain model of the system.

LeakWatch is the most recent of a family of tools for statistical approximation of information leakage developed by Chothia et al. [12,14]. LeakWatch analyzes Java code, requiring the programmer to annotate the code of the system with secret and observable values, then simulates the system repeatedly using the Java Virtual Machine and estimates the correlation between the secret and observable values. LeakWatch follows a different perspective than QUAIL and Moped-QLeak, since LeakWatch computes an approximated result, contrarily to QUAIL's arbitrary precision and Moped-QLeak's fixed double precision. LeakWatch's approximation can be improved at the cost of running more simulations, which is time expensive.

Moped-QLeak [11] uses the Moped tool [16] to compute a symbolic summary of the program under analysis as an Algebraic Decision Diagram (ADD), and then computes the leakage using the ADD representation. The symbolic approach is very efficient when the program can be represented in a compact way using ADDs, and in these cases Moped-QLeak is significantly faster than the other tools.

In this paper we introduce two scalable case studies for the benchmarking of quantitative information leakage tools. Both case studies arise from real-life privacy problems. The case studies are anonymity of user data in Smart Grids and privacy comparison in voting protocols.

Smart Grids are in the family of interconnected objects and have received a growing interest over the last years. Our case study is based on a real system deployed at fortiss[1] labs [21]. In our case study, we focus on the negotiation between a set of *prosumers* and an *aggregator*. The prosumers (PROducer conSUMERs) consume, store and produce energy. To stabilize the grid, the prosumers negotiate with the aggregator how much energy they will exchange with the grid for the next period of time. This exchange might expose the consumption of one of the prosumers, and, in turn, allow a potential attacker to deduce that a house is empty or that a factory has increased its production. In that example, the difficulty is to decide not only whether the exact information can be deduced or not, but also how well an attack can approximate it. Measuring the leakage indicates how much of the secret is unveiled through the negotiation phase. We show that increasing the number of prosumers also increases security.

In the voting protocols comparison case study, we compare two different voting protocols: the *Single Preference*, where each voter expresses a single vote for his favorite

[1] http://fortiss.org

candidate, and the *Preference Ranking*, where each voter ranks all candidates from his most to his least favorite. In both cases there are multiple voters and candidates, and the secret is the preference of each voter. Both protocols have a large number of possible secrets and outputs, so they become cumbersome to analyze even with a small number of voters and candidates.

We compare the tools on their computation time, precision of the answer returned, scalability and usability. Since no tool works strictly better than the others in all category, we determine the problem classes that are better suited to be analyzed by each tool.

2 Background: Information Leakage

The information leakage of a program is a measure quantifying how much information an attacker infers about the program's secret by observing the program's output. We assume that the attacker has access to the program's source code, unlimited computational power, and some prior information about the secret (e.g. the bit size of the secret). Leakage corresponds to the reduction in the attacker's uncertainty about the secret.

Let $\mathtt{h}$ be a random variable with values in a domain $D(\mathtt{h})$ representing the value of the secret and $\mathtt{o}$ be a random variable with values in a domain $D(\mathtt{o})$ modeling the value of the output. The information the attacker has on the secret is modeled by a discrete probability distribution, i.e. for a discrete random variable X a function $\pi : D(X) \to [0, 1]$ such that $\sum_{x \in D(X)} \pi(x) = 1$. The information that the attacker has on the secret before the attack is modeled by the *prior distribution* $\pi(h)$ while the information the attacker has after observing the output is modeled by the *posterior distribution* $\pi(h|o)$. We consider the prior distribution as given, since it is part of the model of the attacker. Let U be an uncertainty measure defined on probability distributions, including Shannon entropy, min-entropy, and g-vulnerability. Computing leakage for the measure U reduces to computing the prior and posterior distributions and applying the formula

$$Leakage_U = U(\pi(h)) - U(\pi(h|o)) \tag{1}$$

$$= U(\pi(h)) - \sum_{\bar{o} \in D(o)} \pi(o = \bar{o}) U(\pi(h|o = \bar{o})) \tag{2}$$

In this work we want to compute Shannon leakage, and thus we use Shannon entropy as the measure of uncertainty: $U(\pi(x)) = \sum_{x \in D(X)} \pi(x) \log_2 \pi(x)$

3 Quantitative Information Leakage Tools

We introduce the quantitative information leakage computation tools that will be tested on the case studies.

3.1 QUAIL

QUAIL [5] computes Shannon and min-entropy leakage of a program written in an imperative WHILE language. The language allows the user to program naturally with

constants, arrays, and loops, and QUAIL compiles such language in an if-goto Markovian semantics. Given the prior information of the attacker, QUAIL represents the program as a Markov chain, and computes the information leakage from the Markov chain with an arbitrary number of precision digits.

Syntax We present the syntax of the QUAIL imperative language we use to model programs. We distinguish the variables in *public* and *private* variables according to their level of abstraction: public variables have precise values, while private variables have sets of possible values. The observable variable `o` is public, while the secret variable `h` is private. Let v (resp. h) range over names of public (resp. private) variables and x range over reals from $[0; 1]$. Let L (resp. H) be the set of assignments of values to public variables (resp. sets of values to private variables). Assume that the secret is a private variable `h` taking values in a known domain $D(\texttt{h})$ and the observable is a public variable `o` taking values in a known domain $D(\texttt{o})$.

Let `label` denote program points and `f` (`g`) pure arithmetic (Boolean) expressions. Assume a standard set of expressions and the following statements:

stmt ::= `public int`*n* `v:=`*k* | `private int`*n* `h` | `v := f(`L`)` | `v := rand` x |
`skip` | `goto` *label* | `return` | `if g(`L, H`) then goto` l_a
`else goto` l_b

The first statement declares a public variable `v` of size n bits with a given value k, while the second statement similarly declares a private variable `h` of size n bits with allowed values ranging from 0 to $2^n - 1$. The third statement assigns to a public variable the value of expression `f` depending on public variables; assignment to private variables or depending on the value of private variables is not allowed. The fourth statement assigns zero with probability x, and one with probability $1-x$, to a public variable. The `return` statement outputs values of all public variables and terminates. A conditional branch first evaluates an expression `g` dependent on private and public variables, and it jumps to label l_a if `g` is true and to label l_b otherwise.

Since only a single variable scope exists, loops can be added in a standard way as syntactic sugar.

As a contribution, we present a method to compute information leakage of a program by analyzing the execution traces of the program. We introduce the Markovian semantics of our langage by means of a function computing the successors of each state. Then we explain how we perform a depth-first exploration of the traces of the system, obtaining a set Q of final states that represent all possible output states of the system. Finally, we show how to compute the posterior entropy from Q.

Semantics The Markovity of the semantics allows us to define states containing enough information to determine a probability distribution over all traces originating from any state.

Definition 1. *A* state *in a Markovian semantics is a tuple* (pc, L, H, p) *where* $pc \in \mathbb{N}^0$ *is the program counter,* L *an assignment function assigning a value to each public*

$$\frac{pc\text{: } \texttt{public int}n\ \texttt{v} := k}{succ(pc, L, H, p) = \{(pc+1, L \cup \{(L(v) = k, n)\}, H, p)\}}$$

$$\frac{pc\text{: } \texttt{private int}n\ \texttt{h}}{succ(pc, L, H, p) = \{(pc+1, L, H \cup \{H(h) = \{0, ..., 2^n - 1\}, n)\}, p)\}}$$

$$\frac{pc\text{: } \texttt{skip}}{succ(pc, L, H, p) = \{(pc+1, L, H, p)\}} \qquad \frac{pc\text{: } \texttt{v := f}(L)}{succ(pc, L, H, p) = \{(pc+1, L \cup \{(L(\texttt{v}) = \texttt{f}(L), n)\}, H, p)\}}$$

$$\frac{pc\text{: } \texttt{v := rand } x}{succ(pc, L, H, p) = \{(pc+1, L(\texttt{v}) = 0, H, p \cdot x), (pc+1, L(\texttt{v}) = 1, H, p \cdot (1-x))\}}$$

$$\frac{pc\text{: } \texttt{goto } label}{succ(pc, L, H, p) = \{(label, L, H, p)\}} \qquad \frac{pc\text{: } \texttt{return}}{succ(pc, L, H, p) = \emptyset}$$

$$\frac{pc\text{: } \texttt{if g}(L, H)\ \texttt{then goto } l_a\ \texttt{else goto } l_b}{\begin{array}{c} succ(pc, L, H, p) = \{(l_a, L, H|\texttt{g}(L, H), p \cdot Pr(\texttt{g}(L, H)|\pi(h))), \\ (l_b, L, H|\neg\texttt{g}(L, H), p \cdot Pr(\neg\texttt{g}(L, H)|\pi(h)))\} \end{array}}$$

Fig. 1: Successor function for each state in the Markovian trace semantics.

variable, H an assignment function assigning a set of values to each private variable, and $0 \leq p \leq 1$ is the probability of the state.

The *initial state* of the semantics is $(1, \emptyset, \emptyset, 1)$. The *set of successor states* of a state (pc, L, H, p) depends on the statement pointed at by the program counter pc. States pointing to a `return` statement have 0 successors, states pointing to a `rand` or `if` statement have up to 2 successors, and any other state has 1 successor. The successor function defining the semantics of the language is shown in Figure 1. If a state has zero probability, e.g. when a conditional is always true, it is removed from the set of successors.

We call a state *final* if it has no successors, meaning that the program counter of the state points to a `return` statement. The trace analysis terminates when a final state is encountered. This means that the analysis terminates if and only if the program under analysis terminates, so non-terminating programs cannot be analyzed with this technique. Non-termination of the program under analysis raises other issues in leakage computation [4], and is not considered here.

Conditional states and random assignment states have two successors. The successors of a conditional state correspond to the guard being true or false. Since the guard can depend on the secret, both successor states may have positive probability depending on the prior distribution $\pi(h)$ on the secret, which is available at this time. The successors of a random assignment state correspond to the bit being set to 0 or 1. In both cases the probability of each successor state is computed and one of the successor states with non-zero probability is chosen to be the next step in the analysis. Successors with probability zero are dropped, pruning unreachable leaves from the trace tree.

Because of the Markovian semantics, each state contains the information to compute the probability distribution over its outgoing transitions. The probability of a trace is computed as the product of the probabilities of the transitions composing the trace. In the successor states of the conditional statement, $H|_{\mathtt{g}\,(L,H)}$ (resp. $H|_{\neg\mathtt{g}\,(L,H)}$) represents the assignment function obtained by removing from the sets of values assigned to the private variables those values that contradict (resp. respect) the guard $\mathtt{g}\,(L,H)$.

Similarly, $Pr(\mathtt{g}\,(L,H)\,|\pi(h))$ (resp. $Pr(\neg\mathtt{g}\,(L,H)\,|\pi(h))$) refers to the probability that the guard $\mathtt{g}\,(L,H)$ is true (resp. false) considering the prior probability distribution $\pi(h)$ on the private variables.

When the analysis of a single trace terminates, the corresponding final state $(\bar{pc}, \bar{L}, \bar{H}, \bar{p})$ is produced, in which pc points to a `return` statement. The sets of allowed values assigned to the private variables in $\bar{H}$ have been appropriately reduced to account for the conditional statements visited by the trace.

Depth-first Trace Exploration We perform a depth-first exhaustive exploration of the execution traces of the system, starting from the initial state $(1, \emptyset, \emptyset, 1)$. Each trace is explored until it gets to a final state, then the final state gets added to the multiset Q of final states. When a state with more than one successor is found during the exploration of a trace, one of the successor states is chosen as the next state to explore and all other successors are put on the stack of the states still to be explored, following a depth-first strategy. When all traces have been explored, the full multiset of final states Q of the system is produced. We then use Q to compute the posterior entropy of the system using Algorithm 1 presented below. The leakage of the system is computed as the difference between the prior and posterior entropy, as explained in Section 2.

Note that the exploration also depends on the prior distribution $\pi(h)$: values of the secret with a probability zero in the prior distribution are not explored. This behavior is intended, as is avoids unnecessarily exploring traces that have probability zero.

The depth-first exploration algorithm can be parallelized to take advantage of multi-core architectures and is implemented in the current release of the QUAIL tool, available at http://project.inria.fr/quail.

Posterior Uncertainty Computation We show how to compute the posterior uncertainty $U(\pi(h|o))$ of a system with a secret $\mathtt{h}$ and an observable $\mathtt{o}$, given the uncertainty measure U and a multiset Q of final states of the system. Q encodes the posterior joint probability of all variables in the system and can be produced by the depth-first exploration algorithm presented above.

Let (pc, L, H, p) be a final state in Q, where L represents the assignments of given values to the public variables, H the assignments of sets of values to the private variables, and p the joint probability of such assignments. Since different traces may produce the same final assignments to variables (L, H), the joint probability of these assignments is the sum of the probabilities of all such final states. To apply the formula (2) $U(\pi(h|o)) = \sum_{\bar{o}\in D(o)} \pi(o = \bar{o}) U(\pi(h|o = \bar{o}))$, we need to compute the marginal probability distribution $\pi(o)$ and for each observable output $\bar{o} \in D(o)$ s.t. $\pi(o = \bar{o}) > 0$ the corresponding conditional probability distribution on h, i.e. $\pi(h|o = \bar{o})$.

Data: uncertainty measure U, multiset Q of final states
Result: posterior uncertainty $U(\pi(h|o))$
1 Initialize $\pi(o)$ and all $\pi(h, o = \bar{o})$ to zero;
2 **forall the** $s = (pc, L, H, p) \in Q$ **do**
3 Let $\bar{o} = L(o)$, $\{k_1, ..., k_n\} = H(h)$;
4 Set $\pi(o = \bar{o}) \leftarrow \pi(o = \bar{o}) + p$;
5 **for** $i = 1...n$ **do**
6 Set $\pi(h = k_i, o = \bar{o}) \leftarrow \pi(h = k_i, o = \bar{o}) + p/n$;
7 **end**
8 **end**
9 For each $\bar{o} \in D(o)$ let $\pi(h|o = \bar{o}) \leftarrow \pi(h, o = \bar{o})/\pi(o = \bar{o})$;
10 Return $U(\pi(h|o)) = \sum_{\bar{o} \in D(o)} \pi(o = \bar{o}) U(\pi(h|o = \bar{o}))$

Algorithm 1: Posterior uncertainty computation

Algorithm 1 computes $\pi(o)$ and each $\pi(h|o = \bar{o})$ by analyzing a multiset of final states. For each state (pc, L, H, p) the value of the observable variable $\circ$ and set of values of the secret variable $\mathtt{h}$ are analyzed (lines 2-8). The probability of observing the value $\bar{o}$ of the observable variable in the state is increased by p (line 4), and the probability of observing each of the n values of the secret variable conditioned on $\bar{o}$ is increased by p/n (line 6). Finally, the probability on each subdistribution $\pi(h, o = \bar{o})$ is normalized to 1 by dividing it by $\pi(o = \bar{o})$ to obtain the conditional probability $\pi(h|o = \bar{o})$ (line 9) since $P(X|Y) = {}^{P(X,Y)}/_{P(Y)}$.

Theorem 1. *Algorithm 1 terminates and outputs the posterior uncertainty $U(\pi(h|o))$ of the posterior distribution represented by Q.*

3.2 LeakWatch

LeakWatch [13] estimates the leakage of a Java program with secrets and observations by running it several times for each possible value of the secret and inferring a probability distribution on the observations for each secret. The tool automatically terminates the analysis when the precision of the estimation is deemed sufficient, but different termination conditions can be used.

For small secrets, LeakWatch reliably computes leakage of complex Java programs with minimal interaction from the analyst. For larger secret, i.e. more than 10 bits, LeakWatch takes more time to return a value.

However, the user can decide an acceptable error level for the tool to reduce the computation time necessary to obtain an answer. Also, if the tool is terminated prematurely, it can still provide an answer, even if it will be potentially quite imprecise. This makes LeakWatch the only tool of the three considered that can always provide an answer in a time-limited scenario, since QUAIL and Moped-QLeak generate a leakage result only if they complete their execution.

Finally, LeakWatch provides many command-line options for tuning the analysis parameters. In particular, one of the options displays the current estimation of the leakage at regular intervals, which can be very useful when developing.

Syntax and usage The syntax is the same as the Java language, with the additional commands `secret(name,value)` to declare a secret with a given name and value, and `observe(value)` to declare an observation of a given value. The analysis evaluates how much information leaks from the secret to the observable values. In particular, LeakWatch can compute leakage from a point of a program to another point of the program, and not necessarily from the start to the termination of the program.

To run LeakWatch, a Java programa annotated with `secret` and `observable` statements has to be compiled linking the LeakWatch library:

```
javac -cp leakwatch-0.5.jar:. MyClass.java
```

The tool is then run passing the name of the compiled class as a parameter:

```
java -jar leakwatch-0.5.jar MyClass
```

We have used the `-n` parameter to fix the number of executions of the program when we experimented with different precisions and computation times. Normally LeakWatch determines automatically when it has run enough executions. The tool returns its leakage estimate for the Java program.

3.3 Moped-QLeak

Moped-QLeak [11] uses the Moped tool [16] to compute a symbolic Algebraic Decision Diagram (ADD) representation of the *summary* of a program, which contains the relation between the inputs and outputs of the program. Moped-QLeak then computes Shannon or min-entropy leakage from this ADD representation using two algorithms introduced by the authors. To obtain the ADD representation of the program, Moped basically performs a fix-point iteration.

Moped's ability to build a symbolic representation of a program depends on the program's complexity. When such representation is computed, Moped-QLeak computes the information leakage with a small time overhead. On the other hand, some programs are not easy to reduce to a symbolic representation, and in this case Moped-QLeak's computation does not terminate within a reasonable time.

The ADD-based representation of probability distributions in Moped-QLeak allows the tool to model examples with large secret and observation spaces. In particular, the authors test it with 32-bit secrets and observables, whereas QUAIL's computation time tends to be exponential in the size of the observables and LeakWatch's in the size of the secret. This suggests that the ADD approach is a key improvement on the state of the art, allowing the analysis tools to analyze off-the-shelf programs using 32- and 64-bit variables.

3.4 Syntax and usage

The tool analyzes programs written in a variant of Moped's Remopla language. We provide here a simplified version of the syntax used by Moped-QLeak.

$$
\begin{aligned}
\mathit{stmt} ::= \ & \texttt{skip ;} \mid \mathit{ident} \texttt{ = } \mathit{exp}\texttt{;} \mid \texttt{pchoice (::}\mathit{prob}\texttt{->}\mathit{stmt}\texttt{)}^{+} \texttt{ choicep} \\
& \mid \texttt{do :: } \mathit{exp} \texttt{ -> } \mathit{stmt} \texttt{ :: else -> } \mathit{stmt} \texttt{ od} \\
& \mid \texttt{if :: } \mathit{exp} \texttt{ -> } \mathit{stmt} \texttt{ :: else -> } \mathit{stmt} \texttt{ fi}
\end{aligned}
$$

The `if` and `do` constructs from Remopla, originally non-deterministic, have been made deterministic in this version. The language has also been enriched with a probabilistic choice operator, `pchoice` which allows the programmer to probablistically define the next statement (e.g. by giving a probability *prob* to each statement). Remopla supports loops, arrays and integers of arbitrary size. The language is normally used to encode systems for model checking against temporal logics.

The language does not provide constructs to declare secrets and observables, but assume that all global variables are at the same time secret and observable. More precisely, the initial values are considered as the input and the final values as the output. In practice, a variable is made secret by assigning it the same value in all final states.

Moped is executed on a Remopla file `MyFile.rem` by calling

```
mql -shannon MyFile.rem
```

where `-shannon` specifies that the tool will compute Shannon leakage. The tool returns the leakage value for the Remopla program.

4 Case Studies

We evaluate the three tools described in the previous Section with two scalable case studies. In order to compare them, we consider the following criteria:

Speed Evaluating the time required by the tool to provide a result;
Accuracy Evaluating the precision of the result returned by the tool;
Scalability Evaluating how the tool behaves on larger instances of the case studies;
Usability Evaluating how easy it is for the user to model a case study in the language used by each tool and to run the analysis.

4.1 Case Study A: Smart Grids

A Smart Grid is an energy network where every node may produce, store and consume energy. Nodes are called *prosumers* (PROducer consSUMERS). The *Living Lab* demonstrator [21] is an instance of such a prosumer, whose data can be accessed online[2]. The prosumers periodically negotiate with an aggregator in charge of balancing the consumption and production among several prosumers. Figure 2 depicts a grid with 3 prosumers. Each prosumer declares its plan, that is, how much

[2] livinglab.fortiss.org

it intends to consume or produce during the next period of time. The aggregator sends back a value indicating the excess of energy production or consumption. An excess of 0 indicates that the plans are feasible and terminates the negotiation. Otherwise, the prosumers adapt their plan accordingly and send the updated version.

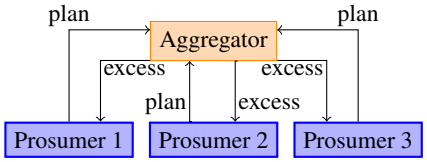

Fig. 2: Smart grid overview

Smart grid and smart sensors raise several security and privacy concerns. The platform can ensure the information cannot flow directly between prosumers [10]. However, stability requires a feedback from the aggregator that potentially conveys infomation about other prosumer, where only the software can limit information leakage. In general, knowing the consumption of a particular household may reveal some sensitive information about the house (presence of people in the house, type of electrical devices . . .). Therefore, the consumption of a prosumer should remain secret. The privacy of a prosumer with respect to the aggregator can be ensured in several ways [28]. However, each prosumer receives some information about the consumption of other prosumers through the excess value sent back by the aggregator.

An attacker might use the information obtained through the grid in order to decide whether a given house is occupied or not. In our scenario, we assume different types of houses with different consumptions. Each house is modeled by a private boolean value, which is true if the house is occupied. An occupied house consumes a fixed amount of energy, according to its type. An empty house does not consume anything. Table 1 presents how much a given house consumes, in two different cases that we consider.

Table 1: Consumption of houses wrt size

Size	Case A	Case B
Small	1	1
Medium	2	3
Large	3	5

For this experiment, we assume that the attacker observes the global consumption of the quarter. We consider different targets for the attack and thus different secrets. Either the attacker targets a single house of a given type (i.e. S, M or L) and only the bit corresponding to the presence in that house is secret, or the attacker wants to obtain informations about all the houses and the whole array of bits indicating the presence in each house is secret.

Usability We model the above scenario in the three tools. We consider two versions depending on the target. When targeting all the houses, the secret is an array of boolean.

When targeting a single house, the secret is a single boolean, which is of course supported by all the tools. However, the presence in the other houses is not a secret, but still an unknown and unobservable input of the program. In QUAIL, the `private` keyword allows the programmer to declare directly such variables. With LeakWatch, we chose these values randomly but do not declare them as secret. In Moped-QLeak, we choose these values randomly, as in LeakWatch.

The first two columns indicate the case, as presented in Table 1 and the number of houses in the model. For a model with N houses, there are $N/3$ houses of each type. The columns S, M and L indicates the leakage of the variable representing the presence in a house of the corresponding type. The column “Global leakage” contains the leakage

of the whole array of presence information bits and the column "Global leakage/bit" indicates the average leak per bit of secret.

Table 2: Leakage of presence information through the global consumption

Case	Nb of Houses	Single house leakage S	M	L	Global leakage	Global leakage/bit
A	3	0.7500	0.7500	0.7500	2.7500	0.9166
A	6	0.0688	0.1466	0.2944	3.4210	0.5701
A	9	0.0214	0.0768	0.1771	3.7363	0.4151
A	12	0.0135	0.0544	0.1273	3.9479	0.3289
B	3	1.0000	1.0000	1.0000	3.0000	1.0000
B	6	0.1965	0.1965	0.3687	4.0243	0.6707
B	9	0.0241	0.0808	0.2062	4.3863	0.4873
B	12	0.0074	0.0510	0.1443	4.6064	0.3838

Table 3: Time to compute or approximate the leakage for a large house

Case	House Nb	Time QUAIL	Time LW	Time mql
A	3	0.1s	0.3s	0.02s
A	6	0.3s	0.3s	0.02s
A	9	0.6s	0.4s	0.02s
A	12	1.6s	0.4s	0.03s
B	3	0.2s	0.3s	0.02s
B	6	0.3s	0.5s	0.02s
B	9	0.6s	0.4s	0.02s
B	12	1.7s	0.4s	0.03s

In Case B with only 3 houses, the presence information can be deduced from the global consumption information, which is indicated by a leakage of 1 for each presence bit. Otherwise, the average leakage per bit from a global attack is more important that the information obtained by focusing on a single house. This means than obtaining information about the whole array, for instance the number of occupied houses, is easier than obtaining information about a single bit, i.e. presence information of a single house. In both cases, the leakage, and thus the loss of anonymity of prosumers, diminishes when the number of houses increases.

Speed In Table 3 we show the time needed by QUAIL, LeakWatch and Moped-QLeak for computing the leakage of the presence information in a house of size L. Moped-QLeak takes around 20ms to compute this value, LeakWatch takes between 300 and 500 ms and QUAIL takes between 100 and 1700 ms, depending on the size of the model. Furthermore, Moped-QLeak and QUAIL compute the exact leakage value, whereas LeakWatch computes an approximation. For a more precise comparison, we need to take precision into account.

Table 4: Average relative error over 100 runs and computation time for analyzing with LeakWatch the leakage of the presence in a large house within 12 houses in Case B.

Simulations	`mql`	Default	1000	2000	**QUAIL**	5000	10000	20000	50000
Error	0%	14.0%	10.4%	6.4%	**0%**	4.8%	2.8%	2.1%	1.4%
Time	0.031s	0.4s	0.7s	1.0s	1.7s	2.1s	3.7s	6.9s	16.6s

Accuracy We compare QUAIL, LeakWatch and Moped-QLeak on computing the leakage of the presence information information of a single large house, in Case B. QUAIL takes 1.7s to compute the exact leakage. With the default parameters, LeakWatch takes 0.4s to

compute an approximation with a relative error of 14% (average on 100 runs). It requires 500 to 700 simulations. To compare execution times with respect to errors, we did an additional experiment, where we requested LeakWatch to run more simulations. For each requested number of simulations we provide in Table 4 the average relative error (over 100 runs) and the time needed for the computation. We see that for an equivalent amount of time, LeakWatch provides a result with a relative error of 4 to 6%, whereas QUAIL returns the exact result. Moped-QLeak is the fastest and most precise.

Scalability Finally, we evaluate the scalability of the tools by increasing the number of houses until the analysis time reaches 1 hour. For this experiment, we try to evaluate the leakage of the presence information, in Case B, for a single house of size L (1 bit of secret), or for all the houses simultaneously (N bit of secret). The results are shown in Table 5. We see that LeakWatch can handle a very large number of houses when computing the leakage from a small secret, but is not much more scalable than QUAIL with a large secret. Recall that LeakWatch provide an approximation of the leakage, whereas QUAIL and Moped-QLeak provide the exact value. Moped-QLeak scales relatively well with both a small and a large secret to analyze.

Table 5: Maximal size analyzable in one hour

Target	LW	QUAIL	mql
L-size house	150000	27	234
All houses	15	12	150

4.2 Case Study B: Voting Protocols

In an election, each voter is called to express his preference for the competing candidates. The *voting system* defines the way the voters express their preference: either on paper in a traditional election, or electronically in e-voting. After the votes have been cast, the *results* of the vote are published, usually in an aggregated form to protect the anonymity of the voters. Finally, the winning candidate or candidates is chosen according to a given *electoral formula*.

In this section we present two different typologies of voting, representing two different ways in which the voters can express their preference: in the *Single Preference* protocol the voters declare their preference for exactly one of the candidates, while in the *Preference Ranking* protocol each voter ranks the candidate from his most favorite to his least favorite.

Single Preference The Single Preference protocol typology models all electoral formulae in which each of the N voters expresses one vote for one of the C candidates, including plurality and majority voting systems and single non-transferable vote [23]. The votes for each candidate are summed up and only the results are published, thus hiding information about which voter voted for which candidate. The candidate or candidates to be elected are decided according to the electoral formula used.

The secret is an array of integers with a value for each of the N voters. Each value is a number from 0 to $C - 1$, representing a vote for one of the C candidates. The observable is an array of integers with the votes obtained by each of the C candidates.

The protocol is simple, and its information leakage can be computed formally, as shown by the following lemma:

Lemma 1. *The information leakage for the Single Preference protocol with n voters and c candidates corresponds to*

$$-\sum_{k_1+k_2+\ldots+k_c=n} \frac{n!}{c^n k_1!k_2!\ldots k_c!} \log_2 \left(\frac{n!}{c^n k_1!k_2!\ldots k_c!} \right)$$

The proof for Lemma 1 is in the Appendix. While the lemma provides a formula to "manually" compute the leakage, it is very hard to find such a formula for an arbitrary process. Therefore automated tools should be employed.

Preference Ranking The Preference Ranking protocol typology models all electoral formulae in which each of the n voters expresses an order of preference of the c candidates, including the alternative vote and single transferable vote systems [23]. In the Preferential Voting protocol the voter does not express a single vote, but rather a ranking of the candidates; thus if the candidates are A, B, C and D the voter could express the fact that he prefers B, then D, then C and finally A. Then each candidate gets c points for each time he appears as first choice, $c-1$ points for each time he appears as second choice, and so on. The points of each candidate are summed up and the results are published.

The secret is an array of integers with a value for each of the N voters. Each value is a number from 0 to $C! - 1$, representing one of the possible $C!$ rankings of the C candidates. The observable is an array of integers with the points obtained by each of the C candidates. The full model for this protocol is shown in the Appendix due to space constraints.

Table 6: Voting protocols: percent of secret leaked by Single Preference (on the left) and Preference Ranking (on the right)

SP		Candidates				
		2	**3**	**4**	**5**	**6**
Voters	**3**	60.4 %	65.7 %	69.0 %	71.3 %	73.0 %
	4	50.8 %	56.5 %	60.2 %	62.9 %	64.9 %
	5	44.0 %	49.6 %	53.5 %	56.4 %	58.6 %
	6	38.9 %	44.4 %	48.3 %	51.2 %	53.5 %

PR		Candidates		
		2	**3**	**4**
Voters	**3**	60.4 %	61.9 %	62.0 %
	4	50.8 %	51.0 %	timeout
	5	44.0 %	43.4 %	timeout
	6	38.9 %	37.9 %	timeout

Experimental Results

Usability We model the two voting systems, where the secret is the votes, and the observable the results. In single preference voting, the secret is an array of integer that represent individual votes. The range of this integer corresponds to the number of candidate. In QUAIL, it is possible to declare the range of a secret integer. In LeakWatch, each vote is drawn uniformly in the valid values and then declared secret. In Moped-QLeak, this case requires more work. The range of a secret integer depends on the chosen size bits. A special variable, `out_of_domain`, is set to true if one of the votes is not in

the valid range and the corresponding input is not considered. Furthermore, when using this variable, it's not possible to use local variables, which complicates the modeling.

For the Preferential Voting, we were not able to produce a Moped-QLeak program that terminates. We suspect that Moped is unable to compute a symbolic representation of the Preferential Voting protocol due to its inherent complexity. Indeed, this program decodes an integer betwen 0 and the factorial of the number of candidates into a sorted list of the candidates, to assign the corresponding points to the candidates.

Table 7: Percent error of the leakage obtained by LeakWatch relatively to the exact value for Single Preference (on the left) and Preference Ranking (on the right)

SP		Candidates				
		2	**3**	**4**	**5**	**6**
Voters	**3**	-3.8 %	-3.7 %	-3.2 %	-2.8 %	-2.2 %
	4	-5.1 %	-3.7 %	-2.6 %	-2.3 %	-2.1 %
	5	-5.0 %	-3.2 %	-2.6 %	-2.2 %	-1.9 %
	6	-5.1 %	-3.2 %	-2.4 %	timeout	timeout

PR		Candidates		
		2	**3**	**4**
Voters	**3**	-3.8 %	-2.6 %	timeout
	4	-5.1 %	-2.6 %	timeout
	5	-5.0 %	-2.2 %	timeout
	6	-5.1 %	timeout	timeout

Accuracy Table 6 shows the percentage of the secret leaked by the Single Preference and Preference Ranking protocols for different numbers of voters and candidates. We note that the results for 2 candidates are identical, since in this case in both protocols the voters can vote in only 2 different ways. The results obtained for Single Preference are correct with respect to the formula stated in Lemma 1. The table shows that the Single Preference protocol leaks a greater percentage of its secret than the Preference Ranking protocol.

Table 8: Time in seconds needed to compute the leakage for Single Preference with QUAIL (left), LeakWatch (middle) and Moped-QLeak (right). Timeout is set to one hour.

SP		Candidates				
QUAIL		**2**	**3**	**4**	**5**	**6**
Voters	**3**	0.2	0.3	0.4	0.5	0.8
	4	0.3	0.5	1.0	1.6	2.7
	5	0.3	0.9	2.5	6.8	13.3
	6	0.5	2.8	13.3	56.7	214.4

SP		Candidates				
LW		**2**	**3**	**4**	**5**	**6**
Voters	**3**	0.4	0.8	2.5	6.9	19.1
	4	0.5	2.4	14.1	64.6	232.3
	5	0.7	8.1	81.6	549.4	2688.3
	6	1.1	29.0	481.6	to	to

SP		Candidates				
mql		**2**	**3**	**4**	**5**	**6**
Voters	**3**	0.8	0.8	0.9	1.0	1.1
	4	0.9	0.5	0.9	1.2	1.6
	5	1.0	1.1	1.2	6.8	2.7
	6	1.1	1.2	1.6	2.5	4.6

Table 7 shows the percent error of the leakage value obtained with LeakWatch. Indeed, LeakWatch computes an approximation of the leakage based on simulation,

whereas QUAIL and Moped-QLeak compute the exact value. Furthermore, the leakage computed by LeakWatch for a given program may change at each invocation of the tool, because LeakWatch samples random executions. Here, LeakWatch slightly underestimates the leakage, by 2 to 5%.

Speed We compare the execution time of the three tools in Table 8 for Single Preference and in Table 9 for Preference Ranking. These execution times have been obtained on a laptop with a i7 quad-core running at 3.3GHz and 16GB of RAM. The results show that QUAIL is significantly faster than LeakWatch on these examples. This shows that QUAIL performs better than LeakWatch with large secrets, in line with previous results [5]. For single preference, Moped-QLeak clearly outperforms QUAIL on large examples. The results for Moped-QLeak in the preferential voting case studies are missing from Table 9 because the tool did not terminate in this case study, even with the smallest instance of 2 voters and 2 candidates.

Table 9: Time in seconds needed to compute the leakage for Preference Ranking with QUAIL (on the left) and LeakWatch (on the right). Timeout is set to one hour.

PR		**Candidates**		
QUAIL		**2**	**3**	**4**
Voters	**3**	0.3	2.0	89.4
	4	0.4	9.0	timeout
	5	0.7	76.7	timeout
	6	1.1	2987.8	timeout

PR		**Candidates**		
LW		**2**	**3**	**4**
Voters	**3**	0.4	13.7	timeout
	4	0.5	121.0	timeout
	5	0.8	1267.3	timeout
	6	1.2	timeout	timeout

Scalability Concerning the Scalability, we see that QUAIL and Moped-Qleak are more scalable than LeakWatch, since the latter times out in Tables 8 and 9. For Single Preference, QUAIL stops at 7 voters and 6 candidates, due to an error. Moped-QLeak finished with 12 voters and 6 candidates but returned `-inf` as leakage value, instead of 11. With 9 voters and 6 candidates, the result has approximately 1 bit of errors. Therefore, we conjecture that the `-inf` value is a precision error. On these examples, no tool seems to be much more scalable than the others, due to various reasons.

5 Conclusions

In this paper, we provided two scalable case studies for the leakage computation and used them for comparing the existing tools able to perform such an approximation. We have compared the state of the art in information leakage tools – LeakWatch, QUAIL and Moped-QLeak – on their speed, accuracy, scalability and usability in addressing the case studies. We summarize here our observations and experience with the tools.

Speed Concerning the execution time, Moped-QLeak is usually the fastest tool in providing an exact result. However, in the preferential voting example Moped-QLeak was unable to terminate its analysis in less than one hour even for the smallest instances of the problem. We can note that LeakWatch is faster than QUAIL on small secrets (e.g.

1 bit) but QUAIL outperforms LeakWatch on larger secrets. Finally, LeakWatch is very fast on small secrets, but its result and evaluation of the system (presence or absence of leakage) tends to change between different executions of the tool.

Accuracy The tool giving the most accurate result is QUAIL because it supports arbitrary precision. LeakWatch provides an approximated result and therefore is imprecise by definition. Moped-QLeak does not implement arbitrary precision analysis, and consequently suffers from approximation errors. For instance, we found an error in the order of 1 bit on the majority voting protocol with 9 voters and 6 candidates, for which we have the exact result. Also, for the same protocol with 12 voters and 6 candidates Moped-QLeak reported a leakage of negative infinity bits, which we conjecture is caused by approximation and division-by-zero errors in the computation.

Scalability For small secrets, LeakWatch scales better than the other tools analyzed. In the Smart Grid case study, we managed to analyze the leakage for an aggregation of 150000 houses in less than one hour. However, this result has to be balanced with the fact the returned result is obtained statistically, and varies from one execution to the other.

For large secrets, the winner is Moped-QLeak, as it scales much better than QUAIL on the Smart Grid case study. However, for the voting protocol, QUAIL manages to analyze only two voters less than Moped-QLeak (6 against 8), before approximation issues make Moped-QLeak's results incorrect.

Usability Since all the tools studied here are academic tools who are still in their early years, usability is not necessarily the main concern of their developers. However, we have found some important discrepancies in this area.

The most usable tool is LeakWatch, especially if the program to analyze is already written in Java. In that case, it is sufficient to annotate the program in order to declare the secrets and the observable values. Furthermore, LeakWatch has a command line option to display the current results based on the traces collected so far, which is convenient when the analysis time is very long.

QUAIL has its own language, which is an imperative WHILE language with arrays and constants. QUAIL allows the explicitly declaration of variables as observable, public, private or secret, with a specific range of allowed values. Furthermore, QUAIL has a command-line option to change the values of constants declared in a program, which comes in handy when performing batch experimentation.

Using Moped-QLeak has been more problematic because of some issues with the Remopla language.In particular, the range of the secrets cannot be determined, instead the program has to raise an `out_of_domain` exception when the values are not in the expected range. Also, all integer variables have the same length, defined in the `DEFAULT_INT_BITS` constant. Finally, some error messages are misleading, e.g. "The first computed value is not a constant." is output when a variable local to the main module is declared and the `out_of_domain` variable is used. This issue is undocumented.

To conclude, Moped-QLeak is the fastest tool, because it uses a suitable data structure (Algebraic Decision Diagrams) for representing the executions. However, this data structure may become a problem with complex program, as shown by the preference ranking example, which Moped-QLeak cannot analyze, contrarily to the other tools. The

other tools, QUAIL and LeakWatch are more usable. QUAIL, which also has its own dedicated language, provide some specific constructs for declaring the visibility and range of a variable. We believe that reimplementing QUAIL with a better data structure for probability distributions, like the ADDs used in Moped-QLeak, would provide a fast and usable tool for performing leakage analysis.

6 Related Tools

We discuss some security-related automated tools and their relation with the work presented in this paper.

The STA tool developed by Boreale et al. [7] is similar in intent to the algorithms we propose, since it also uses symbolic trace analysis. More recent work by Boreale et al. [8] introduces a semiring-based semantics able to perform compositional quantitative analysis of non-deterministic systems, but no tool is available at the moment.

Efficient tools have been developed by Phan and Malacaria for information-theoretical analysis of systems. The tools squifc [24], QILURA [25], and jpf-qif [26] use SMT solving to perform a symbolic analysis of C or Java code and to compute channel capacity of programs, where the channel capacity is the maximum information leakage achievable for any prior distribution over the secret and randomness of the system. Since the tools compute channel capacity and not Shannon leakage of randomized systems, they have not been included in our comparison.

McCamant et al. have obtained interesting results in detecting leakage of information by implicit flow by applying dynamic and quantitative taint analysis techniques [19,22]. Again, their techniques have not been included in this evaluation since they do not compute information-theoretical leakage measures like Shannon and min-entropy leakage.

References

1. M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring information leakage using generalized gain functions. In S. Chong, editor, *CSF*, pages 265–279, 2012.
2. M. Backes, G. Doychev, and B. Köpf. Preventing side-channel leaks in web traffic: A formal approach. In *NDSS*. The Internet Society, 2013.
3. F. Biondi, A. Legay, P. Malacaria, and A. Wasowski. Quantifying information leakage of randomized protocols. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI*, 2013.
4. F. Biondi, A. Legay, B. F. Nielsen, P. Malacaria, and A. Wasowski. Information leakage of non-terminating processes. In Raman and Suresh [27], pages 517–529.
5. F. Biondi, A. Legay, L.-M. Traonouez, and A. Wasowski. QUAIL: A quantitative security analyzer for imperative code. In N. Sharygina and H. Veith, editors, *CAV*, 2013.
6. M. Boreale. Quantifying information leakage in process calculi. *Inf. Comput.*, 207(6):699–725, 2009.
7. M. Boreale and M. G. Buscemi. Experimenting with STA, a tool for automatic analysis of security protocols. In *SAC*, pages 281–285. ACM, 2002.
8. M. Boreale, D. Clark, and D. Gorla. A semiring-based trace semantics for processes with applications to information leakage analysis. *Mathematical Structures in Computer Science*, 25(2):259–291, 2015.

9. M. Boreale and F. Pampaloni. Quantitative information flow under generic leakage functions and adaptive adversaries. In E. Ábrahám and C. Palamidessi, editors, *FORTE*, 2014.
10. D. Bytschkow, J. Quilbeuf, G. Igna, and H. Ruess. Distributed MILS architectural approach for secure smart grids. In *SmartGridSec*, pages 16–29, 2014.
11. R. Chadha, U. Mathur, and S. Schwoon. Computing information flow using symbolic model-checking. In Raman and Suresh [27], pages 505–516.
12. T. Chothia and A. Guha. A statistical test for information leaks using continuous mutual information. In *CSF*, pages 177–190. IEEE Computer Society, 2011.
13. T. Chothia, Y. Kawamoto, and C. Novakovic. Leakwatch: Estimating information leakage from java programs. In M. Kutylowski and J. Vaidya, editors, *ESORICS*, volume 8713 of *Lecture Notes in Computer Science*, pages 219–236. Springer, 2014.
14. T. Chothia, Y. Kawamoto, C. Novakovic, and D. Parker. Probabilistic point-to-point information leakage. In *CSF*, pages 193–205. IEEE, 2013.
15. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
16. J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:27–56, June 2008. Special Issue on Constraints to Formal Verification.
17. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.
18. J. W. G. III. Toward a mathematical foundation for information flow security. In *IEEE Symposium on Security and Privacy*, pages 21–35, 1991.
19. M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*. The Internet Society, 2011.
20. B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In P. Madhusudan and S. A. Seshia, editors, *CAV*, pages 564–580. Springer, 2012.
21. D. Koss, F. Sellmayr, S. Bauereiß, D. Bytschkow, P. K. Gupta, and B. Schätz. Establishing a smart grid node architecture and demonstrator in an office environment using the SOA approach. In *SE4SG, ICSE*, pages 8–14. IEEE, 2012.
22. 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.
23. P. Norris. *Electoral Engineering: Voting Rules and Political Behavior*. Cambridge Studies in Comparative Politics. Cambridge University Press, 2004.
24. Q. Phan and P. Malacaria. Abstract model counting: a novel approach for quantification of information leaks. In S. Moriai, T. Jaeger, and K. Sakurai, editors, *ASIACCS*, pages 283–292. ACM, 2014.
25. Q. Phan, P. Malacaria, C. S. Pasareanu, and M. d'Amorim. Quantifying information leaks using reliability analysis. In N. Rungta and O. Tkachuk, editors, *SPIN*, pages 105–108. ACM, 2014.
26. Q. Phan, P. Malacaria, O. Tkachuk, and C. S. Pasareanu. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.
27. V. Raman and S. P. Suresh, editors. *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*.
28. C. Rottondi, S. Fontana, and G. Verticale. A privacy-friendly framework for vehicle-to-grid interactions. In J. Cuéllar, editor, *SmartGridSec*, volume 8448 of *Lecture Notes in Computer Science*, pages 125–138. Springer, 2014.
29. G. Smith. On the foundations of quantitative information flow. In L. de Alfaro, editor, *FOSSACS*, volume 5504 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2009.

A Appendix

Proof (of Theorem 1). Termination of Algorithm 1 is trivial if Q is finite. For the soundness, the algorithm computes posterior uncertainty according to the uncertainty measure U using the formula

$$U(\pi(h|o)) = \sum_{\bar{o} \in D(o)} \pi(o = \bar{o}) U(\pi(h|o = \bar{o}))$$

The computation reduces to finding the probability distribution $\pi(o)$ on the observable variable and the conditional probability distribution on the secret $\pi(h|o = \bar{o})$ for each possible value $\bar{o}$ of the observable variable.

Let $Q_{\bar{o}}$ be the set $\{(pc, L, H, p) \in Q \mid L(\circ) = \bar{\circ}\}$, and let $p(Q_{\bar{o}})$ be the sum of the probabilities p of the states in $Q_{\bar{o}}$. Then $\pi(o = \bar{o}) = p(Q_{\bar{o}})$. Since the sets $Q_{\bar{o}}$ for each $\bar{o}$ form a partition of Q, then $\sum_{\bar{o} \in D(o)} p(Q_{\bar{o}}) = \sum_{\bar{o} \in D(o)} \pi(o = \bar{o}) = 1$, proving that $\pi(o)$ is a probability distribution.

Similarly, let $\bar{h} = \{k_1, \ldots, k_n\}$ and $Q_{\bar{o},\bar{h}}$ be the set $\{(pc, L, H, p) \in Q_{\bar{o}} \mid H(\mathtt{h}) = \bar{h}\}$ and let $p(Q_{\bar{o},\bar{h}})$ be the sum of the probabilities p of the states in $Q_{\bar{o},\bar{h}}$. Then each state in $Q_{\bar{o},\bar{h}}$ represents a fragment of the the joint distribution on (o, h) that is uniform on $\bar{h} = \{k_1, ..., k_n\}$, and $\pi(o = \bar{o}, h \in \{k_1, ..., k_n\}) = p(Q_{\bar{o},\bar{h}})$. The conditional probability of each value of h conditioned by a given $\bar{o} \in D(o)$ is computed as $\pi(h|o = \bar{o}) = \frac{\pi(o=\bar{o},h)}{\pi(o=\bar{o})}$ by the normalization in line 9.

Having computed $\pi(o)$ and all $\pi(h|o = \bar{o})$, we compute posterior uncertainty with the formula presented above.

Lemma 1: The information leakage for the Single Preference protocol with n voters and c candidates corresponds to

$$-\sum_{k_1+k_2+\ldots+k_c=n} \frac{n!}{c^n k_1! k_2! \ldots k_c!} \log_2 \left(\frac{n!}{c^n k_1! k_2! \ldots k_c!} \right)$$

Proof. It is known that, since the program is deterministic, its Shannon leakage corresponds to the entropy $H(\pi(o))$ of the distribution over the output [3]. We compute

$$H(\pi(o)) = -\sum_{o \in D(o)} \pi(o) \log_2(\pi(o))$$

Here o corresponds to a possible output, which is, in our case a vote result. A vote result precises, for each candidate j, the number k_j of votes obtained. Furthermore, the total of all votes is the number of voters v. Thus the domain of o is $D(o) = \{k_1, \ldots, k_c \in \mathbb{N} \mid \sum_{j=1}^{c} k_j = n\}$. The number of votes with result $o = k_1, \ldots, k_c$ corresponds to the number of choices of voters for candidate 1 amongst n, i.e. $\binom{n}{k_1}$, times the number of choice of voters for candidate 2 amongst the ones remaining i.e. $\binom{n-k_1}{k_2}$ and so on. The result is

$$\prod_{i=1}^{c} \binom{n - \sum_{j=1}^{i-1} k_j}{k_i}$$

which simplifies to $\frac{n!}{k_1! k_2! \ldots k_c!}$. As the votes are equiprobable, we have $p(k_1, \ldots, k_c) = \frac{n!}{c^n k_1! k_2! \ldots k_c!}$, hence the result.

B Code of our case studies

B.1 Smart Grid Case Study

Guessing Presence in a single House

QUAIL version:

```
// N is the total number of houses
const N:=12;

// indicates the size of the target. Only one of those should
     be one and all the other 0.
const target_is_S := 0 ;
const target_is_M := 0 ;
const target_is_L := 1 ;

// We consider different sizes of houses. S, M and L indicate
     the number of houses of each size, excluding the one
    chosen for the attack
const S:=N/3 - target_is_S ;
const M:=N/3 - target_is_M ;
const L:=N/3 - target_is_L ;

// each size correspond to a different level of consumption
const  small_consumption := 1 ;
const medium_consumption := 3 ;
const  large_consumption := 5 ;

// the observable is the global consumption of the system
observable int32 global_consumption :=0 ;

// The secret is the consumption of one particular house
secret int1 presence_target:=[0,1];
private array [N-1] of int1 presence:=[0,1];

public int32 i := 0;

if  (presence_target == 1) then
  if  (target_is_S == 1) then
    assign global_consumption:= global_consumption +
        small_consumption ;
  elif (target_is_M == 1) then
    assign global_consumption:= global_consumption +
        medium_consumption ;
  else
    assign global_consumption:= global_consumption +
        large_consumption ;
  fi
fi

```

```
38 while ( i < N-1) do
39   if (presence[i] == 1) then
40     if (i<S) then
41       assign global_consumption:= global_consumption +
             small_consumption ;
42     elif (i<S+M) then
43       assign global_consumption:= global_consumption +
             medium_consumption ;
44     else
45       assign global_consumption:= global_consumption +
             large_consumption ;
46     fi
47   fi
48   assign i:= i + 1 ;
49 od
50 return;
```

LeakWatch version:

```
1  import bham.leakwatch.LeakWatchAPI;
2  import java.security.SecureRandom;
3
4  public class TargetInfo {
5
6    // N is the total number of houses
7    static int N=12;
8
9    // indicates the size of the target. Only one of those
         should be one and all the other 0.
10   static int target_is_S = 0 ;
11   static int target_is_M = 0 ;
12   static int target_is_L = 1 ;
13
14   // We consider different sizes of houses. S, M and L
         indicate the number of houses of each size.
15   static int S=N/3 - target_is_S ;
16   static int M=N/3 - target_is_M ;
17   static int L=N/3 - target_is_L ;
18
19   // each size correspond to a different level of consumption
20   static int  small_consumption = 1 ;
21   static int medium_consumption = 3 ;
22   static int  large_consumption = 5 ;
23
24   // the observable is the global consumption of the system
25   static int global_consumption =0 ;
26
27   //  the presence of people in each house
28   static  int[] presence;
29   // the secret is the presence
```

```
30  static int presence_target;
31
32  static void initPublicValues(String[] args) {
33     // parses command line to set parameters
34  }
35
36  static void initPrivateValues() {
37    SecureRandom rand = new SecureRandom();
38    presence = new int[N-1];
39    for(int i =0; i<N-1 ; i++){
40      presence[i]=rand.nextInt(2);
41    }
42    presence_target=rand.nextInt(2);
43    LeakWatchAPI.secret("presence_target",presence_target);
44  }
45
46
47  public static void main(String[] args){
48    initPublicValues(args);
49    // init private values in the intervals defined in QUAIL
        file
50    initPrivateValues();
51    if  (presence_target == 1) {
52      if  (target_is_S == 1) {
53        global_consumption= global_consumption +
              small_consumption ;
54      }
55      else if (target_is_M == 1) {
56        global_consumption= global_consumption +
              medium_consumption ;
57      }
58      else {
59        global_consumption= global_consumption +
              large_consumption ;
60      }
61    }
62
63    int i  = 0;
64    while ( i < N-1) {
65      if (presence[i] == 1) {
66        if (i<S) {
67          global_consumption = global_consumption +
                small_consumption ;
68        }
69        else if (i<S+M) {
70          global_consumption = global_consumption +
                medium_consumption ;
71        }
72        else{
```

```
73            global_consumption = global_consumption +
                  large_consumption ;
74          }
75        }
76        i= i + 1;
77      }
78      LeakWatchAPI.observe(global_consumption);
79    }
80
81  }
```

Moped-QLeak version:

```
1  // we use 16 bits integers
2  define DEFAULT_INT_BITS 16
3
4  // N is the total number of houses
5  define N 12
6
7  // consumptions of houses according to their size
8  define CONS_S  1
9  define CONS_M  3
10 define CONS_L  5
11
12 // We consider different sizes of houses. S, M and L indicate
       the number of houses of each size.
13 define S N/3
14 define M N/3
15 define L N-(S+M)-1
16
17 // The secret is the presence of a house of size L
18 bool presence;
19
20 // the observable is the global consumption of the system
21 int global_consumption;
22
23 module void main(){
24   int i;
25
26   // set global consumption to 0: ignore prior value
27   global_consumption = 0;
28
29   if
30   :: presence -> global_consumption = global_consumption +
         CONS_L;
31   :: else -> skip;
32   fi
33
34   i=0;
35   do
```

```
36   :: i<S -> pchoice
37              :: 0.5 -> global_consumption = global_consumption
                     + CONS_S;
38              :: 0.5 -> skip;
39              choicep
40              i=i+1;
41   od
42
43   i=0;
44   do
45   :: i<M -> pchoice
46              :: 0.5 -> global_consumption = global_consumption
                     + CONS_M;
47              :: 0.5 -> skip;
48              choicep
49              i=i+1;
50   od
51
52   i=0;
53   do
54   :: i<L -> pchoice
55              :: 0.5 -> global_consumption = global_consumption
                     + CONS_L;
56              :: 0.5 -> skip;
57              choicep
58              i=i+1;
59   od
60   // hide the secret input
61   presence = false ;
62 }
```

Guessing Presence in all the Houses

QUAIL version:

```
1 // N is the total number of houses
2 const N:=12;
3
4 // We consider different sizes of houses. S, M and L indicate
       the number of houses of each size.
5 const S:=N/3 ;
6 const M:=N/3 ;
7 const L:=N-S-M ;
8
9 // each size correspond to a different level of consumption
10 const  small_consumption := 1 ;
11 const medium_consumption := 3 ;
12 const  large_consumption := 5 ;
13
```

```
// the observable is the global consumption of the system
observable int32 global_consumption :=0 ;

// The secret is the presence of people in each hose
secret array [N] of int1 presence:=[0,1];

public int32 i := 0;

while ( i < N) do
  if (presence[i] == 1) then
    if (i<S) then
      assign global_consumption:= global_consumption +
          small_consumption ;
    elif (i<S+M) then
      assign global_consumption:= global_consumption +
          medium_consumption ;
    else
      assign global_consumption:= global_consumption +
          large_consumption ;
    fi
  fi
  assign i:= i + 1 ;
od
return;
```

LeakWatch version:

```
import bham.leakwatch.LeakWatchAPI;
import java.security.SecureRandom;

public class GlobalInfo {

  // N is the total number of houses
  static int N=12;

  // We consider different sizes of houses. S, M and L
      indicate the number of houses of each size.
  static int S=N/3 ;
  static int M=N/3 ;
  static int L=N-S-M ;

  // each size correspond to a different level of consumption
  static int  small_consumption = 1 ;
  static int medium_consumption = 3 ;
  static int  large_consumption = 5 ;

  // the observable is the global consumption of the system
  static int global_consumption =0 ;

  // The secret is the presence of people in each hose
```

```
23   static  int[] presence;
24
25   static void initPublicValues(String[] args) {
26     // parse commandline to get parameters
27   }
28
29   static void initPrivateValues() {
30     SecureRandom rand = new SecureRandom();
31     // secret var is used to sumarize the table in an int
32     int secret_var = 0;
33
34     presence = new int[N];
35     for(int i =0; i<N ; i++){
36       presence[i]=rand.nextInt(2);
37       secret_var=2*secret_var + presence[i];
38     }
39     LeakWatchAPI.secret("secret_var",secret_var);
40   }
41
42
43   public static void main(String[] args){
44     initPublicValues(args);
45     // init private values in the intervals defined in QUAIL
         file
46     initPrivateValues();
47
48     int i  = 0;
49     while ( i < N) {
50       if (presence[i] == 1) {
51         if (i<S) {
52           global_consumption = global_consumption +
                 small_consumption ;
53         }
54         else if (i<S+M) {
55           global_consumption = global_consumption +
                 medium_consumption ;
56         }
57         else{
58           global_consumption = global_consumption +
                 large_consumption ;
59         }
60       }
61       i= i + 1;
62     }
63     LeakWatchAPI.observe(global_consumption);
64   }
65
66 }
```

Moped-QLeak version:

```
1  // we use 16 bits integers
2  define DEFAULT_INT_BITS 16
3
4  // N is the total number of houses
5  define N 12
6
7  // consumptions of houses according to their size
8  define CONS_S  1
9  define CONS_M  3
10 define CONS_L  5
11
12 // We consider different sizes of houses. S, M and L indicate
       the number of houses of each size.
13 define S N/3
14 define M N/3
15 define L N-(S+M)
16
17 // The secret is the presence of people in each house
18 bool presence[N];
19
20 // the observable is the global consumption of the system
21 int global_consumption;
22
23 module void main(){
24   int i;
25
26   // set global consumption to 0: ignore prior value
27   global_consumption = 0;
28
29   i=0;
30   do
31   :: i<N -> if
32             :: presence[i] ->
33                 if
34                 :: i<S -> global_consumption =
                       global_consumption + CONS_S;
35                 :: else -> if
36                            ::  i<S+M -> global_consumption =
                                   global_consumption + CONS_M;
37                            :: else -> global_consumption =
                                   global_consumption + CONS_L;
38                            fi
39                 fi
40            :: else -> skip;
41            fi
42            i=i+1;
43   od
44
```

```
45
46   // hide the secret input
47   i=0;
48   do
49   :: i <N -> presence[i] = false;
50      i=i+1;
51   od
52 }
```

B.2 Voting Case Study

Single Preference

QUAIL version:

```
1  // N is the number of voters
2  const N:=3;
3
4  // C is the number of candidates
5  const C:=2;
6
7  // the result is the number of votes of each candidate
8  observable array [C] of int32 result;
9
10 // The secret is the preference of each voter
11 secret array [N] of int32 vote:=[0,C-1];
12
13 // this is just a counter
14 public int32 i:=0;
15 public int32 j:=0;
16 // voting
17 while (i<N) do
18   while (j<C) do
19     if (vote[i]==j) then
20       assign result[j]:=result[j]+1;
21     fi
22     assign j:=j+1;
23   od
24   assign j:=0;
25   assign i:=i+1;
26 od
27
28 return;
```

LeakWatch version:

```
1  import bham.leakwatch.LeakWatchAPI;
2  import java.security.SecureRandom;
3
4  public class MajorityVoting {
```

```
5  // N is the number of voters
6  public static int N=3;
7
8  // C is the number of candidates
9  public static int C=2;
10 // the result is the number of votes of each candidate
11 static int[] result;
12
13 // The secret is the preference of each voter
14 static int[] vote;
15
16
17 static void initPublicValues(String[] args) {
18   N=Integer.parseInt(args[0]);
19   C=Integer.parseInt(args[1]);
20
21   result = new int[C];
22   for (int i=0 ; i<C ; i++) result[i] = 0;
23 }
24
25 static void initPrivateValues() {
26   SecureRandom rand = new SecureRandom();
27
28   vote = new int[N];
29   for(int i =0; i<N ; i++){
30     vote[i]=rand.nextInt(C);
31     LeakWatchAPI.secret("vote[i]",vote[i]);
32   }
33 }
34
35 public static void main(String[] args){
36   initPublicValues(args);
37   // init private values in the intervals defined in QUAIL
         file
38   initPrivateValues();
39
40   int i = 0;
41   int j  = 0;
42   while (i < N) {
43     j=0;
44     while (j < C) {
45 if (vote[i] == j) {
46   result[j] = result[j] + 1 ;
47 }
48 j= j+1;
49     }
50     i= i + 1;
51   }
52   for(j=0; j<C ; j++) LeakWatchAPI.observe(result[j]);
53 }
```

```
54 }
```

Moped-QLeak version:

```
1  define DEFAULT_INT_BITS 8
2
3  // N is the number of voters
4  define N 3
5
6  // C is the number of candidates
7  define C 2
8
9  // the result is the number of votes for each candidate
10 int result[C];
11
12 // The secret is the preference of each voter
13 int vote[N];
14
15 // to restrict the domain
16 bool out_of_domain;
17
18 int i; //loop counter
19 int j; //vote index
20
21 module void main() {
22   if
23   :: E k (0,N-1) vote[k]>=C  -> out_of_domain = true;   //
         discard invalid votes value
24   :: else ->
25   i=0;
26
27   do
28   :: i<C ->
29        result[i] = 0; //set score to zero (result is an
             output)
30        i=i+1;
31   od
32
33   i=0;
34   do
35   :: i<N ->
36      j=vote[i];
37      result[j] = result[j] + 1; //result[vote[i]] does not
           work
38      // hide secret
39      vote[i]=C;
40      i=i+1;
41   od
42
43   i=0;
```

```
44   j=0;
45   fi
46 }
```

Preference Ranking

QUAIL version:

```
1  // N is the number of voters
2  const N:=3;
3
4  // C is the number of candidates
5  const C:=2;
6
7  // CFACT is C factorial (until we implement factorials in
       QUAIL)
8  const CFACT:=4;
9
10 // the result is the number of votes of each candidate
11 observable array [C] of int32 result;
12
13 // The secret is the preference of each voter, from 0 to C!-1
14 secret array [N] of int32 vote:=[0,CFACT-1];
15
16 // these bits represent the votes received by the voting
       machine
17 public array [N] of int32 decl;
18
19 public array [C] of int32 temparray;
20 public int32 pos;
21 // this is just a counter
22
23 public int32 voter:=0;
24 public int32 candidate:=0;
25 public int32 k:=0;
26 public int32 y:=0;
27
28 // voting
29 while (voter<N) do
30   while (candidate<CFACT) do
31     if (vote[voter]==candidate) then
32       assign decl[voter]:=candidate;
33     fi
34     assign candidate:=candidate+1;
35   od
36   assign candidate:=0;
37   assign voter:=voter+1;
38 od
39
```

```
// transform the secret of each voter into the order of the
    preferences
assign voter:=0;
while (voter<N) do

   // build the initial array
  assign candidate:=0;
  while (candidate<C) do
    assign temparray[candidate]:=candidate;
    assign candidate:=candidate+1;
  od

  assign k:=C;
  // find a position
  while (k>0) do
    assign pos := decl[voter]%k;
    assign candidate:=C-k;
    // update the vote of the candidate
    assign result[candidate]:=result[candidate]+temparray[pos
        ];

    // remove the element from the array
    assign y:=pos;
    while (y<C-1) do
      assign temparray[y]:=temparray[y+1];
      assign y:=y+1;
    od

    // update the vote of the voter
    assign decl[voter]:=decl[voter]/k;

    // decrease the counter
    assign k:=k-1;
  od
  assign voter:=voter+1;
od
// calculate the results
return;
```

LeakWatch version:

```
import bham.leakwatch.LeakWatchAPI;
import java.security.SecureRandom;

public class PreferentialVoting {
  // N is the number of voters
  public static int N=3;

  // C is the number of candidates
  public static int C=2;
```

```
10  // the result is the number of votes of each candidate
11  static int[] result;
12
13  // The secret is the preference of each voter
14  static int[] vote;
15
16
17  static int[] decl;
18  static int[] temparray;
19
20  static int fact(int n){
21    if (n <= 1)
22      return 1;
23    else
24      return n*fact(n-1);
25  }
26
27  static void initPublicValues(String[] args) {
28    if (args.length < 2) {
29      System.out.println("Usage: PreferentialVoting <
            nb_voters> <nb_candidates>");
30      System.exit(1);
31    }
32    N=Integer.parseInt(args[0]);
33    C=Integer.parseInt(args[1]);
34
35    result = new int[C];
36    for (int i=0 ; i<C ; i++) result[i] = 0;
37  }
38
39  static void initPrivateValues() {
40    SecureRandom rand = new SecureRandom();
41
42    vote = new int[N];
43    int CFACT= fact(C);
44    for(int i =0; i<N ; i++){
45      vote[i]=rand.nextInt(CFACT);
46      LeakWatchAPI.secret("vote[i]",vote[i]);
47    }
48
49    /*other_votes = new int[N-S];
50    for(int i =0; i<N-S ; i++){
51      other_votes[i]=rand.nextInt(C);
52    }*/
53  }
54
55  public static void main(String[] args){
56    initPublicValues(args);
57    // init private values in the intervals defined in QUAIL
          file
```

```
58    initPrivateValues();
59
60    int voter = 0;
61    int vote_val = 0;
62    decl=new int[N];
63    // voting
64    while (voter<N) {
65      while (vote_val<fact(C)) {
66  if (vote[voter]==vote_val) {
67    decl[voter]=vote_val;
68  }
69  vote_val=vote_val+1;
70      }
71      vote_val=0;
72      voter=voter+1;
73    }
74
75    // transform the secret of each voter into the order of
          the preferences
76    voter=0;
77    while (voter<N) {
78
79      // build the initial array
80      int candidate=0;
81      temparray = new int[C];
82      while (candidate<C){
83  temparray[candidate]=candidate;
84  candidate=candidate+1;
85      }
86
87      int k=C;
88      // find a position
89      while (k>0) {
90  int pos = decl[voter]%k;
91  candidate=C-k;
92  // update the vote of the candidate
93  result[candidate]=result[candidate]+temparray[pos];
94
95  // remove the element from the array
96  int y=pos;
97  while (y<C-1) {
98    temparray[y]=temparray[y+1];
99    y=y+1;
100 }
101
102 // update the vote of the voter
103 decl[voter]=decl[voter]/k;
104
105 // decrease the counter
106 k=k-1;
```

```
107         }
108         voter=voter+1;
109       }
110       for (int j=0; j<C ; j++){
111         LeakWatchAPI.observe(result[j]);
112       }
113     }
114 }
```

The non working version of preferential voting: this program is parsed correctly, but the analysis does not terminate, even after 3 hours.

```
1  define DEFAULT_INT_BITS 8
2
3  // N voters C candidates
4  define C 2
5  define N 2
6  // CFACT is C factorial
7  define CFACT 2
8
9  //votes are secret
10 //results are observable
11 unsigned int vote[N];
12 unsigned int result[C];
13
14 bool out_of_domain;
15
16 // intermediate variables
17 int temparray[C];
18 int i,j;
19 int pos;
20
21 module void main() {
22   if
23   :: E k (0,N-1) vote[k] >= CFACT  -> out_of_domain = true;
           // discard invalid votes value
24   :: else ->
25
26   // set results to 0
27   i=0;
28   do
29   :: i<C -> result[i] = 0;
30             i=i+1;
31   od
32
33   i=0;
34   do
35   :: i< N -> //
36                j=0;
37                do
```

```
38              :: j<C -> temparray[j]=j;
39                 j=j+1;
40              od
41              j=C; //not really necessary
42              do
43              :: j > 0 ->
44                pos = vote[i] - (vote[i]/j)*j; // pos = vote[i
                      ] % j
45                result[C-j] = result[C-j] + temparray[pos];
46                do
47                :: pos < C-1 -> temparray[pos] = temparray[pos
                       + 1];
48                   pos=pos + 1;
49                od
50                vote[i]=vote[i]/j;
51                j=j-1;
52              od
53              vote[i]=0; //forget vote[i]
54              i=i+1;
55    od
56
57    // forget intermediate values
58    pos=0;
59    do
60    :: pos<C -> temparray[pos] = 0;
61       pos=pos+1;
62    od
63    i=0;
64    fi //close the main if
65  }
```