Information Leakage by Trace Analysis in QUAIL

Fabrizio Biondi, Jean Quilbeuf, Axel Legay

Information Leakage by Trace Analysis in QUAIL

Fabrizio Biondi[1], Jean Quilbeuf[1], and Axel Legay[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 based on computing exponentially large channel matrices or Markov chains, making them impractical for large programs. We propose a different approach based on abstract trace analysis. By analyzing directly sets of execution traces of the program and computing security measures on the results, we are able to scale down the exponential cost of the problem. Also, we are able to appy statistical simulation techniques, allowing us to obtain significant results even without exploring the full space of traces. We have implemented the resulting algorithms in the QUAIL tool. We compare their effectiveness against the state of the art LeakWatch tool on two case studies: privacy of user consumption in smart grid systems and anonymity of voters in different voting schemes.

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 [14]. 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 do 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 [13]. Different information leakage measures have been introduced, including Shannon leakage [15], min-entropy leakage [21] and the g-leakage [1], encoding different security properties of the system. Among the results in the field, Köpf et al. studied leakage of side-channel attacks [2,16], while Boreale has defined leakage for process calculi [6] and characterized the best attack strategy of an adaptive attacker [7].

In previous work, we have presented a quantitative analysis technique able to precisely quantify information leakage of a system to an arbitrary number of decimal digits after modeling a system-attacker scenario with a Markov chain and analyzing it by

applying results in both information theory and graph theory [4]. The method has been implemented in our QUantitative Analyzer for Information Leakage (QUAIL) tool[1] [5].

To our knowledge, QUAIL is the only tool enabling precise and exact information leakage computation, and later tools by multiple authors have used it as comparison [11,19]. Nonetheless, QUAIL needs to produce a full Markov chain model of the system-attacker scenario to produce a meaningful result. Such chain has an exponential number of states in the size of the secrets and observables, making it cumbersome to compute.

The first contribution of this paper is a new algorithm for precise information leakage computation, which we call Exhaustive Trace Exploration (`Exp`). The algorithm is able to compute information leakage following the same Markovian semantics we introduced previously [4, Section 4] by analyzing the execution traces of the system. This avoids building the full Markov chain model of the system.

The `Exp` algorithm performs an exhaustive depth-first exploration of all system traces, so that every trace is explored exactly once. The probability and information leakage of each trace is computed, and the information leakage of the system is obtained as the expected value of the information leakage over all traces.

We will show the effectiveness of the algorithm by comparing it with the previous QUAIL implementation and the state of the art LeakWatch tool [11]. LeakWatch is the most recent of a family of tools for statistical approximation of information leakage developed by Chothia et al. [10,12]. 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, since the former simulates the full program a large amount of times to produce an approximated result while the latter performs a single full abstract exploration of the traces of the system to produce the exact value of the leakage.

The second contribution of this paper is the presentation of two scalable case studies for quantitative information leakage analysis and the comparison of LeakWatch and QUAIL on these case studies. 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[2] labs [17]. 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 with QUAIL indicates how much of the secret is unveiled through the negotiation phase. We show that increasing the number of prosumers also increases security.

[1] http://project.inria.fr/quail
[2] http://fortiss.org

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 candidate, and the *Preference Ranking* where each voter ranks all candidates from his most to his least favorite. In both case 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 with both QUAIL and LeakWatch with a small number of voters and candidates. We show that the Preference Ranking protocol is more effective in protecting the privacy of the voters, and that QUAIL can prove it significantly faster and with more precision than LeakWatch. In general, the size of the secret is an important parameter for the comparison: LeakWatch perfoms better on systems with a small secret, while QUAIL scales better with the size of the secret.

The rest of the paper is structured as follows. Section 2 introduces standard concepts and notations. In Section 3 we present the algorithms for leakage computation via trace analysis. Section 4 elaborates on the case studies, and finally Section 5 discusses statistical extensions of the algorithms presented.

2 Information Leakage with QUAIL

We want to compute the information leakage of a program, i.e. 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

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

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

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 $\mathtt{o}$ is public, while the secret variable $\mathtt{h}$ 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` | `private int`*n* `h` | `v := f(`L`)` | `v := rand` x |
`skip` | `goto` *label* | `return` | `if g(`L, H`) then goto` *label_a*
`else goto` *label_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 *label_a* if `g` is true and to *label_a* otherwise.

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

3 Information Leakage by Trace Analysis

We present a method to compute information leakage of a program by analyzing the execution traces of the program. We present in Subsection 3.1 how a single trace is produced according to the markovian semantics of our langage. Subsection 3.2 presents how the leakage is computed from a multiset of final states. Subsections 3.3 and 3.4 present the `Exp` algorithm and details about its distributed implementation. The `Exp` algorithm explores all the traces to produce the multiset of final states needed to compute the leakage.

3.1 Single Trace Analysis

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 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 *successor states* of a state (pc, L, H, p) depend on the statement pointed at by the program counter pc, as shown in the semantics in Figure 1.

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

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

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

$$\frac{pc\text{: } \texttt{v := rand } x}{(pc, L, H, p) \to (pc+1, L[^0/_v], H, p \cdot x), (pc+1, L[^1/_v], H, p \cdot (1-x))}$$

$$\frac{pc\text{: } \texttt{goto label}}{(pc, L, H, p) \to (\texttt{label}, L, H, p)} \qquad \frac{pc\text{: } \texttt{return}}{\text{terminate}}$$

$$\frac{pc\text{: } \texttt{if g(L,H) then la: A else lb: B}}{\begin{array}{c}(pc, L, H, p) \to (\texttt{la}, L, H|_{\texttt{g(L,H)}}, p \cdot Pr(\texttt{g(L,H)}|\pi(h))),\\ (\texttt{lb}, L, H|_{\neg\texttt{g(L,H)}}, p \cdot Pr(\neg\texttt{g(L,H)}|\pi(h)))\end{array}}$$

Fig. 1: Execution rules in Markovian trace semantics.

We call a state *final* if 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. We refer to our work in the subject to handle the computation of leakage of non-terminating programs [3].

To analyze a single execution trace, we start from a given Markov state (pc, L, H, p), e.g. the initial state $(1, \emptyset, \emptyset, 1)$. The program counter points to the line in the source code to be executed according to the semantics in Figure 1.

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 positive probability is chosen to be the next step in the analysis.

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, $H|_{\texttt{g}(L,H)}$ (resp. $H|_{\neg\texttt{g}(L,H)}$) 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 $\texttt{g}(L,H)$. Similarly, $Pr(\texttt{g}(L,H)|\pi(h))$ (resp. $Pr(\neg\texttt{g}(L,H)|\pi(h))$) refers to the probability

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

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

When a single trace analysis terminates, a given 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.

3.2 Posterior Uncertainty Computation Algorithm

We show how to compute the posterior uncertainty $U(\pi(h|o))$ of a system with a secret `h` and an observable `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 `Exp` algorithm presented in Section 3.3.

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. $\pi(h|o = \bar{o})$.

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 `o` and set of values of the secret variable `h` are analyzed. The probability of observing the value $\bar{o}$ of the observable variable in the state is increased by p, and the probability of observing each of the n values of the secret variable conditioned on $\bar{o}$ is increased by p/n. 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 $\pi(h|o = \bar{o})$.

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

Data: uncertainty measure U, prior distribution $\pi(h)$, imperative code of the program
Result: leakage of the program according to U

```
Initialize an empty multiset Q;
Put the initial state (1, ∅, ∅, 1) on the stack S;
while stack S is not empty do
    Pop a state s = (pc, L, H, p) from the stack S;
    while s is not a final state do
        Find the successors s_1, ..., s_n of s that have positive probability;
        Put s_2, ...s_n on the stack S;
        Assign s := s_1;
    end
    Add s to Q;
end
Return U(π(h))−ComputePosteriorUncertainty(U, Q);
```

Algorithm 2: `Exp`: Exhaustive depth-first trace exploration

3.3 Exhaustive Trace Exploration Algorithm

We present the Exhaustive Trace Exploration Algorithm (`Exp`) for the efficient computation of information leakage of a program written in an imperative code equipped with Markovian semantics. The algorithm can be used to compute any information-theoretical measure of leakage, including Shannon leakage, min-entropy leakage, guessing entropy and g-leakage. In the algorithm, the prior and posterior distributions of the attacker are computed, and information leakage is computed as the difference between the value of the chosen measure on the prior and posterior distribution, following common practice in the field. The algorithm can be parallelized to take advantage of multicore architectures and is implemented in the current release of the QUAIL tool, available at http://project.inria.fr/quail. Details about the implementation are given in Section 3.4.

The `Exp` algorithm perform a depth-first exhaustive exploration of the execution traces of the system. 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 state 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. We refer back to Section 3.1 for details about the analysis of a single trace. When all traces have been explored, `Exp` has produced the full multiset of final states Q of the system. The algorithm then computes the information leakage of the system using Algorithm 1. The pseudocode of the algorithm is presented in Algorithm 2. The following theorem proves the correctness of the algorithm:

Theorem 2. *Algorithm 2 terminates on a terminating program and when it does it outputs the information leakage according to the given uncertainty measure U.*

3.4 Distributed Implementation

Algorithm `Exp` is highly parallelizable, and can be implemented to take advantage of multicore and distributed systems. To parallelize it, the stack S and the set Q of final

states must be shared between the processes. Different processes can pop states from S, explore the corresponding traces until they find final states, and add those final states to Q. To avoid race conditions, access to both S and Q must be sequential. QUAIL currently implements this strategy on multicore systems.

Algorithm 1 can also be parallelized. Divide Q in subsets $Q_{\bar{o}} = \{s \in Q | o = \bar{o} \text{ in } s\}$ for each output $\bar{o}$. It is then possible to utilize a different process for each output $\bar{o}$ to analyze each set $Q_{\bar{o}}$ and compute the probability and conditional posterior uncertainty of $\bar{o}$ independently from each other.

4 Case Studies

We compare different leakage analysis tools on two case studies. In the first one, we measure information about occupancy of an house based on the information obtained through a Smart Grid. In the second one, we compare the loss of anonymity of the voters due to the publication of the results of an election with different voting protocols.In both case we measure Shannon leakage, setting $U(\pi(h)) = \sum_{\bar{h} \in D(h)} \pi(h = \bar{h}) \log_2 \pi(h = \bar{h})$. The code of all the case studies presented in this paper is available at https://project.inria.fr/quail/tacas15/.

For our both case studies, we compare `Exp` with LeakWatch [11]. On the first case study we also compare `Exp` with the previous algorithm implemented in QUAIL [4]. LeakWatch evaluates the leakage in Java programs by approximating it from a statistical sample of program executions. In order to run our case studies with LeakWatch, we translate the QUAIL code into Java code. This translation is straightforward except for the declaration of private variables. To handle them, we replace each interval in a declaration by an assignment to a random number within that interval. For instance, the declaration

```
secret array [N] of int32 vote:=[0,C-1];
```

becomes

```
for(int i; i<N ; i++) vote[i] = securerandom.nextInt(C);
```

Then we instrument the code to define the secret and observable variables by using the LeakWatch API. In our example, defining the vote as secret and the results as observable is obtained through:

```
for(int i; i<N ; i++) LeakWatchAPI.secret('vote[i]',vote[i]);
for(int j; j<C ; j++) LeakWatchAPI.observe(result[j]);
```

After compilation of the instrumented code, the analysis is run by the command `java -jar leakwatch-0.5.jar SinglePreference`. This executes the `SinglePreference` class several times and records the secrets and observables to compute the leakage. LeakWatch determines at runtime how much simulations are necessary to obtain a good approximation of the leakage.

4.1 Smart Grid

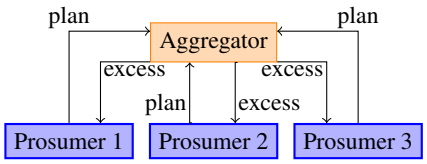

Fig. 2: Smart grid overview

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 [17], is an instance of such a prosumer, whose data can be accessed online[3]. Without coordination between the prosumers, the grid may globally consume or produce too much energy, making it unstable. To stabilize the grid, 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 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 announced plans are feasible and terminates the negotiation. Otherwise, the prosumers then have to adapt their plan accordingly and send the updated version.

Switching to a smart grid structure will probably be required in the coming years. Smart grid and smart sensors raise several security and privacy concerns. The platform can ensure the information cannot flow directly between prosumers [9]. 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 [20]. However, each prosumer receives some information about the consumption of other prosumers through the excess value sent back by the aggregator.

An attacker might try to 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 1	Case 2
Small	1	1
Medium	2	3
Large	3	5

For our analyses, we assume that the attacker observes the global consumption, declared by: **observable int32** global_consumption;. We consider two variants, depending on whether the attacker focuses on a single house or on all the houses. The secret is either the occupancy of one particular house: **secret int1** presence_target:=[0,1]; or the occupancy of all houses: **secret array** [N] **of int1** presence:=[0,1];

Table 2 shows the results of several leakage analyses. The first two columns indicates the case, as presented in Table 1 and the number of houses in the model. For a model with N houses, there is $N/3$ house of each type. The columns S, M and L indicates the

[3] livinglab.fortiss.org

leakage of the variable representing the vacancy of an house of this type. The column "Global leakage" contains the leakage of the whole array of occupancy bits and the column "Global leakage/bit" indicates the average leak for one bit of secret.

Table 2: Leakage of occupancy through the global consumption

Case	Nb of Houses	Single house leakage S	M	L	Global leakage	Global leakage/bit
0	3	0.7500	0.7500	0.7500	2.7500	0.9166
0	6	0.0688	0.1466	0.2944	3.4210	0.5701
0	9	0.0214	0.0768	0.1771	3.7363	0.4151
0	12	0.0135	0.0544	0.1273	3.9479	0.3289
1	3	1.0000	1.0000	1.0000	3.0000	1.0000
1	6	0.1965	0.1965	0.3687	4.0243	0.6707
1	9	0.0241	0.0808	0.2062	4.3863	0.4873
1	12	0.0074	0.0510	0.1443	4.6064	0.3838

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

Case	House Nb	Time MC	Time Exp	Time LW
1	3	0.4s	0.1s	0.3s
1	6	1.6s	0.4s	0.3s
1	9	91.4s	1.0s	0.4s
1	12	timeout	2.6s	0.4s
2	3	0.4s	0.1s	0.3s
2	6	1.6s	0.4s	0.5s
2	9	92.1s	1.0s	0.4s
2	12	timeout	2.7s	0.4s

The average leakage per bit from a global attack is more important that the information obtained by focusing on a single bit. 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. occupancy of a single house. In both cases, the leakage, and thus the anonymity of prosumers, diminishes when the number of houses increases.

In Table 3 we compare the execution time between the previous QUAIL algorithm that builds a global Markov chain from the program, `Exp` and LeakWatch. The `Exp` algorithm is much faster as it needs 3s to perform a computation that requires more than one hour (the timeout value) with the previous algorithm. LeakWatch needs less than 1s on the same example. This experiment validates the fact than LeakWatch performs better than `Exp` when the size of the secret is small (1 bit here).

4.2 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. The voting system also comprehends the additional procedures enforced to guarantee that the voters can vote freely, that they can verify that their vote has been counted and that their vote remains confidential.

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. Since each protocol we model is concerned only about how the votes are

expressed and counted and what results are published, each protocol models a number of electoral formulae. For the same reason, the models are valid both for uninominal and multinominal elections.

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 [18]. 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 secrets and observables are modeled by the following lines of QUAIL code:

```
secret array [N] of int32 vote:=[0,C-1];
observable array [C] of int32 result;
```

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 characterizes the solution computed by QUAIL for this case, it is very hard to find such a characterization for any process, so in general QUAIL is the best way to obtain a result. We run QUAIL with the command `./quail single_preference.quail -v 0 -p 5` and obtain `1.8112` showing that the leakage of the Single Preference protocol for 3 voters and 2 candidates is ≈ 1.8112 bits.

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 [18]. 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 secrets and observables are modeled by the following lines of QUAIL code:

```
secret array [N] of int32 vote:=[0,C!-1];
observable array [C] of int32 result;
```

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

SP	Candidates				
Voters	2	3	4	5	6
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		
Voters	2	3	4
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 Table 4 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 5: Percent error of the leakage obtained by LeakWatch relatively to the one from QUAIL for Single Preference (on the left) and Preference Ranking (on the right)

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

Table 5 shows the percent error of the leakage value obtained with LeakWatch. Indeed, LeakWatch computes an approximation of the leakage based on simulation, whereas QUAIL computes the exact value. For this reason, the leakage computed by LeakWatch for a given program may change at each invocation of the tool. LeakWatch slightly underestimates the leakage.

We compare the execution time of QUAIL and LeakWatch in Table 6 for Single Preference and in Table 7 for Preference Ranking. These execution times have been obtained on a laptop with a i7 quad-core running at 3.3Ghz and 16G of ram. The results show that QUAIL is significantly faster than LeakWatch on these examples. For Single

Preference with 6 candidates and 5 voters, QUAIL is 200 times faster than LeakWatch. This shows that QUAIL performs better than LeakWatch with large secrets.

Table 6: Time in seconds needed to compute the leakage for Single Preference with QUAIL (on the left) and LeakWatch (on the 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	timeout	timeout

Table 7: 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

5 Towards Statistical Estimation: A Perspective

One of the reasons why LeakWatch is faster than the `Exp` algorithm in some examples is that the former just computes an approximation of the leakage, while the latter computes the precise result. This requires `Exp` to explore the totality of the execution traces of the system. However, since the space of traces is generally very large, it would be interesting to be able to analyze only a statistically significant sample of the traces and to estimate the leakage of the system based on the sample.

Statistical computation of information leakage has been explored by the group of Tom Chothia, leading to the development of the LeakWatch tool we compare against, while Boreale proves that no Monte Carlo estimator exists under very mild conditions for the estimation of channel capacity [8], a problem closely related to information leakage.

Data: uncertainty measure U, prior distribution $\pi(h)$, imperative code of the program
Result: leakage of the program according to U
Initialize an empty set Q;
while *termination conditions are not met* **do**
  Let $s := (1, \emptyset, \emptyset, 1)$;
  while s *is not a* `return` *state* **do**
    Find the successors $s_1, ..., s_n$ of s;
    Assign to s one of its successors chosen randomly;
  end
  Add s to Q;
end
Return $U(\pi(h))-$**ComputePosteriorUncertainty**(U, Q);

Algorithm 3: Statistical trace simulation

Statistical estimation of leakage via trace analysis would follow a different path, as explained in Algorithm 3. We call this algorithm `Sim`. The `Sim` algorithm randomly chooses traces to analyze, and terminates when some termination conditions are met. Such conditions can include a certain number or percentage of traces analyzed or a time limit. Contrarily to `Exp`, `Sim` may analyze the same trace twice. In that case, the resulting final state should be added only once to the set Q. This can be obtained by enriching the state s with a hash value depending on the trace producing the final state s. After the analysis, leakage is computed from the data available. Compared to Monte Carlo techniques, `Sim` has the advantage that since we know the probability of each trace explored, the percentage of the probability space actually covered by the simulation is known. In case not all final states are contained in Q, the distribution $\pi(o)$ computed by Algorithm 1 will be a subdistribution. In this case we increase the posterior uncertainty accordingly, assuming that the unanalyzed part of the posterior distribution behaves like the part that has been analyzed. This assumption is valid only if the final states in Q have been produced by statistically independent analyses, so that they represent a meaningful sample of the full set of final states.

Finally, note that in Algorithm 3 we are choosing the successor state to explore randomly and locally, so the probability of analyzing a trace does not correspond with the probability of the trace. This is useful when some of the traces have a very low probability, as it is often the case with authentication protocols. This kind of rare event simulation is a well established approach in probabilistic model checking. In practice we are simulating for a prior distribution $\pi(h)$ but choosing the traces to simulate according to a different prior distribution $\pi'(h)$ that generates more interesting traces with a higher probability. This begs for the question of which prior $\pi'(s)$ allows us to explore the trace space more efficiently, thus providing a more meaningful sample in a shorter time. We leave this question open as future work.

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, B. F. Nielsen, and A. Wasowski. Information leakage of non-terminating programs. In *FSTTCS*, 2014. To appear.
4. 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.
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 F. Pampaloni. Quantitative information flow under generic leakage functions and adaptive adversaries. In E. Ábrahám and C. Palamidessi, editors, *FORTE*, volume 8461 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2014.
8. M. Boreale and M. Paolini. On formally bounding information leakage by statistical estimation. *Unpublished Manuscript*, 2014.
9. D. Bytschkow, J. Quilbeuf, G. Igna, and H. Ruess. Distributed MILS architectural approach for secure smart grids. In *Smart Grid Security - Second International Workshop, SmartGridSec 2014, Munich, Germany, February 26, 2014, Revised Selected Papers*, pages 16–29, 2014.
10. T. Chothia and A. Guha. A statistical test for information leaks using continuous mutual information. In *CSF*, pages 177–190. IEEE Computer Society, 2011.
11. 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.
12. T. Chothia, Y. Kawamoto, C. Novakovic, and D. Parker. Probabilistic point-to-point information leakage. In *CSF*, pages 193–205. IEEE, 2013.
13. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
14. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
15. J. W. G. III. Toward a mathematical foundation for information flow security. In *IEEE Symposium on Security and Privacy*, pages 21–35, 1991.
16. 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.
17. 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.
18. P. Norris. *Electoral Engineering: Voting Rules and Political Behavior*. Cambridge Studies in Comparative Politics. Cambridge University Press, 2004.
19. 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, *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 283–292. ACM, 2014.
20. 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.
21. 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 $\{(pc, L, H, p) \in Q \mid L(\mathtt{o}) = \bar{\mathtt{o}}\}$, 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.

Proof (of Theorem 2). We assume that the program under analysis always terminates, i.e. all traces reach a final state in a finite number of steps. The analysis of a single trace follows the Markovian semantics of the language, as explained in Section 3.1.

The algorithm explores the tree of computation traces following a depth-first strategy. Since all traces terminate, eventually all of them are explored and the set Q contains all of the possible final states of the system. 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.

When the full set Q of final states has been produces the algorithm uses Algorithm 1 to compute the posterior uncertainty and applies the formula

$$Leakage_U = U(\pi(h)) - U(\pi(h|o)) \tag{3}$$

so the proof of correctness is a consequence of Theorem 1.

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 [4]. 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 $o = k_1, \ldots, k_c$ correspond 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. □